

# Compte Rendu - S3.01

## Méthodologie de Travail et Organisation du Projet

### 1. Un Processus de Développement Itératif

Le projet s'est articulé autour de 6 **itérations** distinctes. Chaque itération suivait un cycle de vie précis, garantissant que chaque ajout de fonctionnalité était réfléchi avant d'être codé :

1. **Planification et Conception** : Chaque cycle débutait par une phase d'analyse. Nous utilisions cette étape pour mettre à jour nos diagrammes UML (diagrammes de classes et de séquence) afin d'anticiper les impacts des nouvelles fonctionnalités sur l'architecture existante.
2. **Identification des Impacts** : Nous listions précisément les classes à créer et celles nécessitant des modifications.
3. **Répartition** : Attribution des tâches aux membres du groupe.
4. **Implémentation (Codage)** : Développement effectif des fonctionnalités.
5. **Validation** : Récapitulatif du travail effectué et tests de fonctionnement pour valider l'itération.

### 2. Outils de Gestion et Documentation

La communication et le suivi de l'avancement ont été centralisés via des outils collaboratifs :

- **Notion (Pilotage des Itérations)** : Nous avons utilisé Notion comme document de référence collaboratif. Il nous servait à définir les objectifs de chaque itération. Nous y avons mis en place un système de "tickets" sous forme de tableaux, précisant pour chaque tâche son état (À faire, En cours, Terminé) et son niveau d'importance (Priorité).

Corrections & Problèmes		
Au Nom	Etat	Importance
Gestion de la suppression : à distinguer de l'archivage	Terminé	important
quand je déplace une sous tache de colonne, elle n'est plus enfant => Impossible de déplacer en dehors de la colonne du parent	Terminé	important
définir la période sous tache en fonction de la tache	Terminé	important
Sous-tâche doit avoir la même colonne que sa tâche mère	Terminé	important
vue archivage & désarchivage	Terminé	important
documentation & commentaires	Terminé	important
Jeu de données ne se crée pas	Terminé	détail
gestion de l'archivage : afficher les tâches archivées	Terminé	détail

- **Trello (Documentation)** : L'outil Trello a été utilisé en complément pour documenter l'avancement et garder une trace visuelle des fonctionnalités validées.

### 3. Stratégie de Répartition des Tâches

Afin d'optimiser le temps de développement, nous avons adopté une stratégie de travail en parallèle sur des modules indépendants.

- **Responsabilité Individuelle** : Pour les tâches standard, chaque étudiant se voyait attribuer une fonctionnalité précise ou une classe spécifique à modifier. Cela a permis à chacun de maîtriser au moins une fonctionnalité majeure de bout en bout.
- **Travail en Binôme (Sub-groupes)** : Lors de l'implémentation de fonctionnalités complexes ou structurelles, nous avons divisé l'équipe en deux sous-groupes pour faciliter la résolution de problèmes.

Bien que la répartition initiale fût segmentée, la phase de correction de bugs ("fix") a permis une rotation des tâches : chaque membre a été amené à intervenir sur différentes parties du code, assurant ainsi une connaissance partagée de l'ensemble du projet par tous les membres de l'équipe.

### 4. Gestion de Version (Workflow Git)

La collaboration technique s'est appuyée sur **GitHub** avec une gestion des branches par feature:

- Chaque nouvelle fonctionnalité faisait l'objet d'une **branche dédiée**, isolée de la branche principale.
- Une fois la fonctionnalité développée et testée localement, nous procédions à la fusion (**merge**) de la branche vers le `main`.

Cette méthode a permis de minimiser les conflits de code et de garder une version stable de l'application sur la branche principale tout au long du développement.

[https://github.com/JassemMT/S3-01\\_TAMOURGH-GILBERT-GOFFIN-CHEBAH](https://github.com/JassemMT/S3-01_TAMOURGH-GILBERT-GOFFIN-CHEBAH)

## Liste des fonctionnalités par itération

## Détail des Itérations de Développement

Le développement de l'application a suivi une progression logique en 6 phases, allant de la mise en place de l'architecture MVC jusqu'aux fonctionnalités avancées de récursivité et d'interface utilisateur.

### Itération 1 : Fondations et Modèle de Données

L'objectif initial était de poser les bases de l'architecture logicielle et du modèle de données ; nous avons principalement implémenté la conception établie lors de la phase Analyse/conception.

- **Architecture MVC** : Nous avons structuré le projet en packages distincts (Modele, Vue, Contrôleur) pour respecter la séparation des préoccupations.
- **Modèle de Données** : Création de la classe abstraite `Tache` définissant les attributs communs (libellé, état, durée, couleur). Nous avons défini les états via des constantes (À faire, En cours, Terminé, Archivé).
- **Pattern Observer** : Mise en place des interfaces `Sujet` et `Observateur`. La classe `Modele` centralise les données et notifie automatiquement les vues lors des opérations CRUD (Création, Lecture, Mise à jour, Suppression).
- **Tests** : Validation du modèle via une interface textuelle (`MainTextuel`) avant de passer à l'interface graphique.

### Itération 2 : Premières Vues et Contrôleurs

Cette phase a marqué le passage à l'interface graphique JavaFX.

- **Vue Kanban** : Développement de l'affichage en colonnes dynamiques. Chaque tâche est représentée par une "carte" colorée affichant ses informations clés.
- **Édition** : Création de la `VueEditeurTache`, une fenêtre modale permettant de modifier tous les attributs d'une tâche (titre, commentaire, état, couleur, durée via Spinner).
- **Interactions** : Implémentation des premiers contrôleurs pour créer une tâche, l'archiver ou ouvrir l'éditeur au double-clic.
- **Drag & Drop** : Première implémentation du glisser-déposer basique pour changer une tâche de colonne.

### Itération 3 : Vue Liste et Améliorations

Nous avons diversifié les modes d'affichage et affiné l'expérience utilisateur.

- **Vue Liste** : Ajout d'un affichage tabulaire organisé par jour. Nous avons introduit une indentation visuelle (→) pour représenter la hiérarchie des tâches.
- **Refonte temporaire** : Pour gérer les dépendances rapidement, nous avons temporairement utilisé des listes simples dans la classe `Tache`, mettant de côté le pattern Composite pour se concentrer sur l'affichage.
- **Drag & Drop (v2)** : Amélioration du déplacement des tâches avec un feedback visuel (mise en surbrillance de la zone de dépôt) pour une meilleure fluidité.

- **Modification patron composite** : Comme toute tâche peut, en principe, accueillir une sous-tâche, nous avons supprimé les classes `TacheSimple` et `TacheComposite`, ne laissant plus que la classe `Tache`, possédant une liste d'objets du même type. \*\*Rollback pendant itération 4 à la demande de l'enseignant.

## Itération 4 : Pattern Composite, Gantt et Persistance

C'est l'itération majeure concernant la structure des données et la visualisation temporelle.

- **Retour au Composite** : Réintégration propre du patron Composite avec la distinction `TacheSimple` (feuille) et `TacheComposite` (conteneur).
- **Mécanisme de Promotion** : Implémentation d'une fonctionnalité clé dans le `Modele` : la méthode `promouvoirEnComposite()`. Elle permet de transformer automatiquement une tâche simple en dossier si l'utilisateur lui ajoute une sous-tâche.
- **Vue Gantt** : Création d'un diagramme temporel. L'affichage s'adapte automatiquement à l'échelle de temps (date min/max) et dessine les barres proportionnellement à la durée des tâches.
- **Sérialisation**: Ajout de la sérialisation (méthodes `save` et `load`) pour sauvegarder l'état du projet dans un fichier binaire `save` à la fermeture et le restaurer au démarrage.

## Itération 5 : Récursivité et Cohérence des dates

Nous nous sommes concentrés sur la logique algorithmique pour gérer la hiérarchie des tâches.

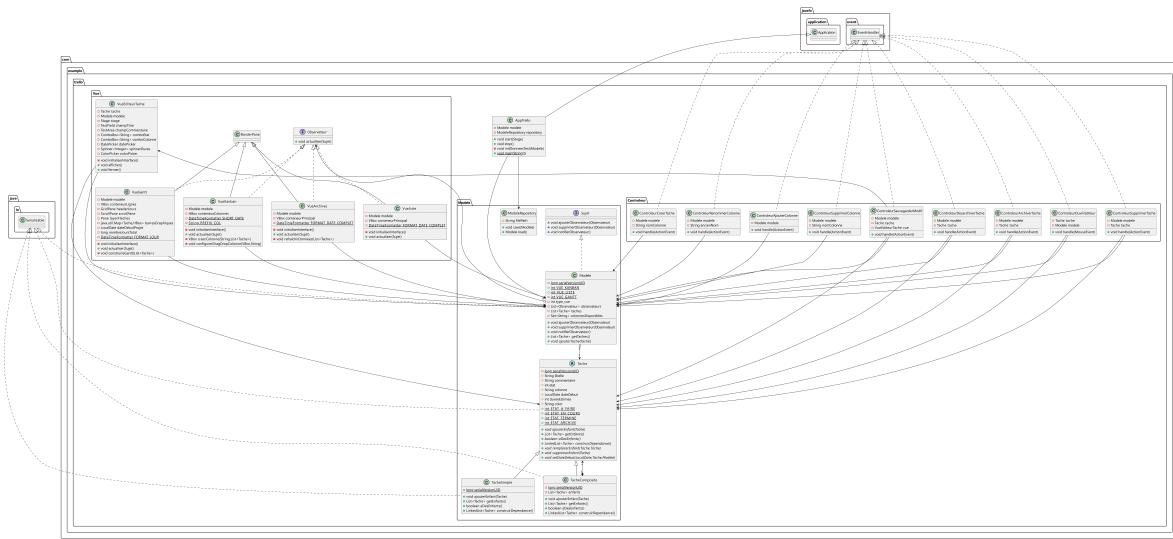
- **Algorithmes Récursifs** : Développement de méthodes pour propager les actions d'un parent vers ses enfants :
  - `supprimerTacheRecursive()` : Supprime toute une branche de l'arbre.
  - `archiverTacheRecursive()` : Archive en cascade.
  - `changerColonneRecursive()` : Déplacer un parent déplace aussi ses sous-tâches.
- **Cohérence des Données** : Ajout de vérifications lors de la création (ex: dates incohérentes) levant des exceptions pour garantir l'intégrité du modèle.

## Itération 6 : Finalisation et Optimisation

La dernière phase a servi à polir l'application et ajouter les dernières fonctionnalités ergonomiques.

- **Propagation Temporelle** : Modification dynamique des dates : changer la date d'une tâche décale automatiquement ses dépendances récursivement.
- **Drag & Drop Colonnes** : Ajout de la possibilité de réorganiser l'ordre des colonnes par glisser-déposer.
- **Vue Archives** : Création d'une vue dédiée (Corbeille) permettant de consulter les tâches archivées, de les restaurer ou de les supprimer définitivement.
- **Documentation** : Finalisation de la Javadoc et commentaire du code pour la maintenabilité.

## Diagramme de classe final



## Évolution Architecturale et Dette Technique

Le développement de l'application n'a pas suivi une ligne droite, mais un processus itératif qui nous a conduits à revoir notre architecture en cours de route. Cela explique certains choix d'implémentation actuels.

### L'Itération "Liste" vs "Composite"

Initialement, nous avions envisagé le **Patron Composite** pour gérer la hiérarchie. Cependant, face à la complexité de mise en œuvre initiale, nous avons temporairement basculé vers une architecture simplifiée : une **liste unique** contenant toutes les tâches, où la parenté était gérée par de simples références.

Rapidement, cette simplification a montré ses limites (difficultés à afficher le Kanban sans doublons, complexité de l'archivage en cascade). Nous avons donc opéré un nouveau changement majeur : revenir à une structure Composite stricte.

### Dette Technique

Ce changement de cap a engendré ce qu'on appelle de la **Dette Technique**. Certaines parties du code, écrites durant la phase "Liste", n'ont pas été entièrement réécrites lors du retour au Composite.

#### L'exemple concret :

Nos contrôleurs de création de tâches (`ControleurCreerTache`) ont conservé la logique de l'étape intermédiaire : ils ajoutent systématiquement la nouvelle tâche à la liste principale du modèle (`modele.ajouterTache`), même s'il s'agit d'une sous-tâche qui devrait être encapsulée dans son parent.

### Une Réponse par la Robustesse (Programmation Défensive)

Conscients de cette incohérence temporaire entre la couche Contrôleur et la couche Modèle, nous avons renforcé la logique du Modèle.

C'est la raison pour laquelle nos algorithmes comme celui de suppression (`supprimerRecuratif`) sont implémentés de manière **défensive** : ils ne présument pas que l'architecture est parfaite. Ils scannent à la fois la structure hiérarchique (parents/enfants) et la liste principale pour nettoyer toute référence résiduelle. Cette approche permet à l'application de rester stable et fonctionnelle malgré les vestiges des itérations précédentes.

### Fonctionnalité dont chacun est fier :

## Rapport Jassem : Architecture et Conception

### 1. Le Patron de Conception Composite

Pour répondre à l'exigence fonctionnelle de gérer des tâches pouvant contenir des sous-tâches (et potentiellement des sous-sous-tâches à l'infini), nous avons opté pour le **Pattern Composite**.

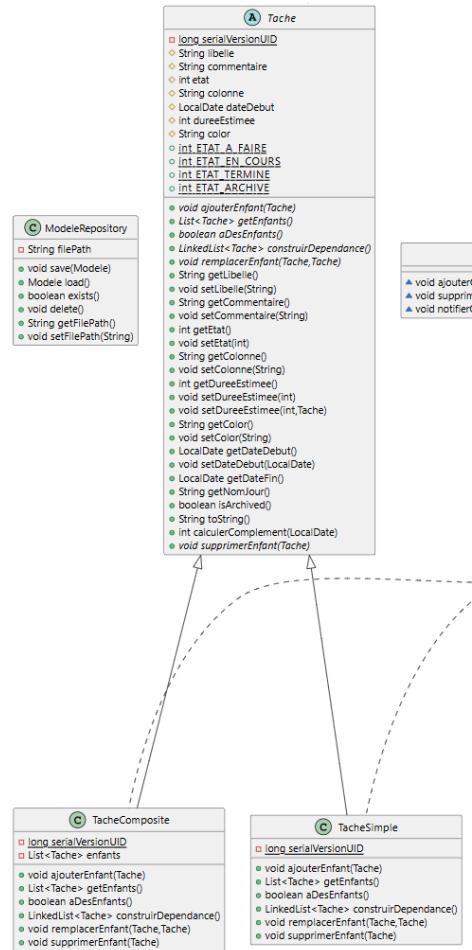
## Le Problème

Sans ce patron, nous aurions dû gérer deux listes distinctes dans le Modèle (une pour les tâches simples, une pour les dossiers) et multiplier les conditions `if/else` pour traiter chaque cas.

## La Solution

Le pattern Composite permet de traiter un objet individuel (feuille) et une composition d'objets (nœud) de manière uniforme.

- **Composant (Abstraction)** : La classe abstraite `Tache`.
  - **Feuille** : La classe `TacheSimple` (ne contient pas d'enfants).
  - **Composite** : La classe `TacheComposite` (contient une liste d'enfants).



## Implémentation dans le Code

Dans la classe abstraite `Tache`, nous définissons les méthodes communes, y compris celles de gestion des enfants (qui sont abstraites).

## Extrait de Tache.java (Le contrat) :

```
public abstract class Tache implements Serializable {  
    // ... Attributs communs (libelle, dateDebut, etat...)  
  
    // Méthodes abstraites que les sous-classes DOIVENT implémenter  
    public abstract void ajouterEnfant(Tache t);  
    public abstract List<Tache> getEnfants();  
    public abstract boolean aDesEnfants();  
  
    // Indispensable pour la suppression dans un Composite
```

```
    public abstract void supprimerEnfant(Tache t);
}
```

Dans `TacheComposite`, ces méthodes manipulent la liste interne, alors que dans `TacheSimple`, elles ne font rien ou renvoient des listes vides.

## 2. Le Polymorphisme

Le polymorphisme est la capacité du système à manipuler des objets de types différents (`TacheSimple` ou `TacheComposite`) à travers leur type parent commun (`Tache`).

### Application Concète : La Suppression

L'exemple le plus frappant du polymorphisme dans notre projet se trouve dans la méthode `supprimerTache` du `Modele`.

Lorsque nous voulons supprimer une tâche, nous devons d'abord la détacher de son parent. Le modèle récupère le parent via `getParentDirect(tache)`. Ce parent est typé comme `Tache`. Le modèle ne sait pas (et n'a pas besoin de savoir) si ce parent est une `TacheSimple` ou `TacheComposite`.

Extrait de `Modele.java` :

```
public void supprimerTache(Tache tache) {
    if (tache != null) {
        // 1. Récupération du parent (Polymorphisme : le retour est de type Tache)
        Tache parent = getParentDirect(tache);

        if (parent != null) {
            // 2. Appel Polymorphe :
            // À la compilation, on appelle Tache.supprimerEnfant()
            // À l'exécution, c'est TacheComposite.supprimerEnfant() qui est exécuté
            parent.supprimerEnfant(tache);
        }

        // 3. Suppression récursive
        supprimerRecursif(tache);
        notifierObservateur();
    }
}
```

Grâce à cette architecture, le code du Modèle reste propre et découplé des implémentations spécifiques des tâches.

## 3. La Promotion Dynamique d'Objet

Une contrainte ergonomique forte était la possibilité de transformer une tâche simple en liste de tâches à la volée (par exemple, en glissant une tâche sur une autre). En Java, on ne peut pas changer la classe d'une instance une fois créée. Nous avons donc implémenté un mécanisme de **Promotion**.

Une contrainte technique majeure de Java est l'immutabilité du type d'une instance : une fois instancié comme `TacheSimple`, un objet ne peut pas devenir `TacheComposite`.

Pour pallier cette rigidité sans abandonner le **Pattern Composite**, nous avons implémenté un mécanisme de **Promotion**.

### Le Principe

La promotion consiste à :

1. Créer une nouvelle instance de `TacheComposite`.

2. Lui transférer toutes les données de l'ancienne `TacheSimple` (titre, dates, couleur...).
3. Remplacer la référence de l'ancienne tâche par la nouvelle dans la structure de données (le parent ou la liste racine).

## Implémentation

Nous utilisons un "constructeur de copie" spécial dans `TacheComposite` et une méthode de remplacement dans le Modèle.

**Extrait de `Modele.java` :**

```
public TacheComposite promouvoirEnComposite(TacheSimple ancienneTache) {
    // 1. Clonage des données dans le nouveau type
    TacheComposite nouvelleTache = new TacheComposite(ancienneTache);

    // 2. Remplacement dans la liste racine (si elle est à la racine)
    int index = taches.indexOf(ancienneTache);
    if (index != -1) {
        taches.set(index, nouvelleTache);
    } else {
        // 3. Remplacement dans le parent (si elle est une sous-tâche)
        Tache parent = getParentDirect(ancienneTache);
        if (parent != null) {
            // Le parent met à jour sa liste interne
            parent.replacerEnfant(ancienneTache, nouvelleTache);
        }
    }

    notifierObservateur();
    return nouvelleTache;
}
```

**Extrait de `Modele.java` :**

```
// constructeur pour passer une tâche simple en tâche composite
public TacheComposite(TacheSimple t) {
    super(t.getLibelle(), t.getCommentaire(), t.getDateDebut(), t.getColonne(), t.getDureeEstimee());
    this.enfants = new ArrayList<>();
    this.setEtat(t.getEtat());
    this.setColor(t.getColor());
}
```

Cette méthode garantit l'intégrité de l'arbre des tâches : aucun lien n'est brisé lors de la transformation de l'objet.

## 4. Algorithmique : Recherche Récursive et Problématique du Lien Unilatéral

Une des contraintes majeures de notre architecture réside dans la modélisation des relations entre les tâches.

### Le Problème : Le Lien Unilatéral

Nous avons choisi une structure d'arbre où la relation est **unidirectionnelle** (Top-Down) :

- **Le Parent connaît ses Enfants** (via la liste `enfants`).
- **L'Enfant ne connaît pas son Parent** (pas d'attribut `parent`).

Cette conception allège la sérialisation (évite les références cycliques infinies lors de la sauvegarde) mais pose un problème critique lors de la suppression ou du déplacement : **comment une sous-tâche peut-elle se détacher de son parent si elle ne sait pas qui il est ?**

## La Solution : Le Parcours en Profondeur (DFS)

Puisque nous ne pouvons pas remonter directement (`tache.getParent()` est impossible), nous devons lancer une recherche depuis la racine du projet pour retrouver "qui possède cette tâche".

Nous avons implémenté un algorithme de **Parcours en Profondeur (Depth First Search)**.

### Implémentation

L'algorithme fonctionne en deux temps :

1. **Itération sur les Racines** : On parcourt les tâches principales.
2. **Récursion (Backtracking)** : Si une racine a des enfants, on descend dans l'arbre jusqu'à trouver le parent recherché.

Extrait de `Modele.java` :

```
/**  
 * Retrouve le parent d'une tâche cible.  
 * Nécessaire car le lien est unilatéral (Parent → Enfant).  
 */  
public Tache getParentDirect(Tache cible) {  
    if (cible == null) return null;  
  
    // 1. On parcourt toutes les tâches racines (niveau 0)  
    for (Tache racine : taches) {  
        // Si la racine est la cible, elle n'a pas de parent  
        if (racine == cible) continue;  
  
        // 2. On lance la sonde récursive dans les profondeurs  
        Tache parentTrouve = chercherParentDirectRecuratif(racine, cible);  
  
        if (parentTrouve != null) {  
            return parentTrouve; // On a trouvé le père, on arrête tout et on le renvoie.  
        }  
    }  
    return null; // Orphelin ou non trouvé  
}  
  
/**  
 * Moteur de recherche récursif (DFS)  
 */  
private Tache chercherParentDirectRecuratif(Tache parentActuel, Tache cible) {  
    if (parentActuel.aDesEnfants()) {  
  
        // A. Vérification immédiate : Est-ce que je suis le père ?  
        // On regarde si la cible est dans ma liste directe d'enfants.  
        if (parentActuel.getEnfants().contains(cible)) {  
            return parentActuel;  
        }  
  
        // B. Appel Récursif : Je demande à mes enfants de chercher dans leurs propres enfants.  
        for (Tache enfant : parentActuel.getEnfants()) {  
            Tache res = chercherParentDirectRecuratif(enfant, cible);  
            if (res != null) {  
                return res;  
            }  
        }  
    }  
    return null;  
}
```

```

        // Si la recherche dans la sous-branche est fructueuse, on remonte le résultat.
        if (res != null) return res;
    }
}
return null; // Pas trouvé dans cette branche
}

```

## Conclusion

L'architecture du projet repose sur une utilisation rigoureuse de la Programmation Orientée Objet. L'utilisation du **Pattern Composite** a simplifié la gestion de la récursivité des tâches. Le **Polymorphisme** a permis d'écrire un code générique et maintenable dans le Modèle. Enfin, la **Promotion d'objet** a permis de contourner les limitations du typage statique de Java pour offrir une expérience utilisateur fluide.

---

## Thomas :

Pour ma part, je vais parler de l'implémentation des flèches dans la vue Gantt pour représenter les relations entre fils et parent, ainsi que comment j'ai pu réaliser cela dans javaFX.

Avant tout l'utilisation de ces flèches répond à une demande spécifique, à savoir le fait de mieux voir les relations de dépendances sur le diagramme de Gantt permettant de voir plus facilement les relations dans un visuel propre et cohérent.

Dans un premier temps, nous devons récupérer les barres de la tache fille et de la tache parent ( en tant que HBox).

```

HBox bEnfant = barresGraphiques.get(enfant);
HBox bMere = barresGraphiques.get(mere);

```

De plus nous devons récupérer les coordonnées de ces barres grâce à l'objet Bounds afin de gérer le point d'arrivée et le point de départ de la flèche :

```

Bounds boundsEnfant = bEnfant.localToScene(bEnfant.getBoundsInLocal());
Bounds boundsMere = bMere.localToScene(bMere.getBoundsInLocal());

```

La méthode `localToScene` permet de récupérer la position absolue des barres de taches présent dans la vue Gantt.

Ces positions nous servirons à définir les points d'encrages de la flèche. Les points d'encrages sont les suivants permettant de définir le point de départ et le point d'arrivée de la flèche.

## Comment sont calculées ces coordonnées ?

```

// point de départ
double xSortie = boundsEnfant.getMaxX() - boundsLayer.getMinX();
double ySortie = boundsEnfant.getMinY() + (boundsEnfant.getHeight() / 2) - boundsLayer.getMinY();

// point d'arrivé
double xEntree = boundsMere.getMinX() - boundsLayer.getMinX();
double yEntree = boundsMere.getMinY() + (boundsMere.getHeight() / 2) - boundsLayer.getMinY();

```

### Le calcul de l'axe X (Horizontal)

- **xSortie** (**Côté droit de l'enfant**) : On utilise `getMaxX()` pour que la flèche commence à l'extrémité droite de la barre de la tâche fille.
- **xEntree** (**Côté gauche de la mère**) : On utilise `getMinX()` pour que la flèche arrive à l'extrémité gauche de la barre de la tâche mère.
- **Le décalage** (`boundsLayer.getMinX()`) : Indispensable pour convertir une position "écran" en une position relative à ton calque de dessin.

### Le calcul de l'axe Y (Vertical) **ySortie** et **yEntree** :

- On prend le haut de la barre (`getMaxY()`) et on y ajoute la moitié de sa hauteur (`getHeight() / 2`).
- **Résultat** : Cela permet de centrer parfaitement les points d'ancre verticalement au milieu de chaque barre.

Ce bloc de code définit deux points :

- **Point A** : Centre-Droit de la tâche précédente (Fille).
- **Point B** : Centre-Gauche de la tâche suivante (Mère).

Cela garantit que la flèche relie toujours la **fin** d'une étape au **début** de la suivante, peu importe où elles se situent dans la fenêtre.

Dans l'objectif de créer la flèche, nous devons créer deux objets distincts , la **courbe** et la **pointe**.

### Comment créer la courbe ?

La courbe est dessiné avec un objet CubicCurve défini par 4 points d'encrages distincts.

```
CubicCurve courbe = new CubicCurve();
courbe.setStartX(xSortie);
courbe.setStartY(ySortie);
courbe.setEndX(xEntree);
courbe.setEndY(yEntree);
```

Ce bloc de code initialise les **points d'ancre** de la flèche.

En bref :

- `new CubicCurve()` : Crée un objet courbe (qui par défaut est invisible et sans forme).
- `setStartX/Y` : Fixe l'origine de la flèche sur le point de **sor**te (le bord droit de la tâche fille).
- `setEndX/Y` : Fixe la destination de la flèche sur le point d'**en**trée (le bord gauche de la tâche mère).

À ce stade, sans les points de contrôle (`ControlX/Y`), la courbe se comporte comme une simple ligne droite entre les deux tâches.

### Comment créer cet effet d'ondulation ?

```
double distance = Math.abs(xEntree - xSortie) * 0.5;
courbe.setControlX1(xSortie + distance);
courbe.setControlY1(ySortie);
courbe.setControlX2(xEntree - distance);
courbe.setControlY2(yEntree);

// implémentation d'un style à la flèche
courbe.setStroke(Color.web("#666666", 0.6));
courbe.setStrokeWidth(1.5);
courbe.setFill(null);
```

## Création de l'effet "S"

Une courbe de Bézier est définie par le fait de ne pas suivre ses points de contrôle, elle est "attirée" par eux comme par des aimants.

- `distance * 0.5` : On calcule la moitié de l'écart horizontal entre les deux tâches.
- `ControlX1 / ControlY1` : Ce point tire la courbe horizontalement vers la droite à partir du départ. Cela force la flèche à sortir bien droite de la première barre.
- `ControlX2 / ControlY2` : Ce point tire la courbe horizontalement vers la gauche juste avant l'arrivée. Cela force la flèche à entrer bien droite dans la barre mère.

## Style visuel

- `setStroke` : Définit la couleur (gris foncé `#666666`) avec une opacité de 60% (`0.6`) pour que la flèche soit visible mais discrète.
- `setStrokeWidth(1.5)` : Règle l'épaisseur du trait pour qu'il soit fin.
- `setFill(null)` : Cela empêche JavaFX de remplir l'intérieur de la courbe avec une couleur.

## Comment créer la pointe de la flèche ?

Pour créer la pointe on utilise un objet `Polygon` que l'on va placer et orienter d'une façon à former la pointe.

### Création de la pointe ( `Polygon` )

Un `Polygon` se définit par une suite de coordonnées (x, y).

Ici, les trois points `(0,0)`, `(-6,-4)` et `(-6,4)` dessinent un petit triangle isocèle :

- **(0,0)** : C'est la pointe de la flèche (le sommet).
- **(-6,-4) et (-6,4)** : Ce sont les deux coins de la base, situés 6 pixels en arrière et 4 pixels de chaque côté.

```
Polygon pointe = new Polygon(0,0, -6,-4, -6,4);
pointe.setFill(Color.web("#666666", 0.6));
pointe.setTranslateX(xEntree);
pointe.setTranslateY(yEntree);

layerFleches.getChildren().addAll(courbe, pointe);
```

### Positionnement ( `setTranslate` )

Par défaut, le polygone est créé en haut à gauche du calque (coordonnées 0,0).

`setTranslateX(xEntree)` et `setTranslateY(yEntree)` : Ces lignes déplacent le triangle pour que son sommet `(0,0)` coïncide exactement avec le point d'entrée de la tâche mère.

### Affichage final ( `getChildren().addAll` )

Le calque `layerFleches` est un conteneur transparent qui recouvre tout le diagramme de Gantt.

`addAll(courbe, pointe)` : Cette instruction injecte simultanément le corps de la flèche (la courbe) et sa pointe dans le calque pour les rendre visibles à l'utilisateur.

## Conclusion : Système de Liaisons Dynamiques

L'implémentation des flèches de dépendance repose sur une **couche graphique indépendante** ([Pane](#)), garantissant une séparation nette entre le contenu (tâches) et la structure (relations).

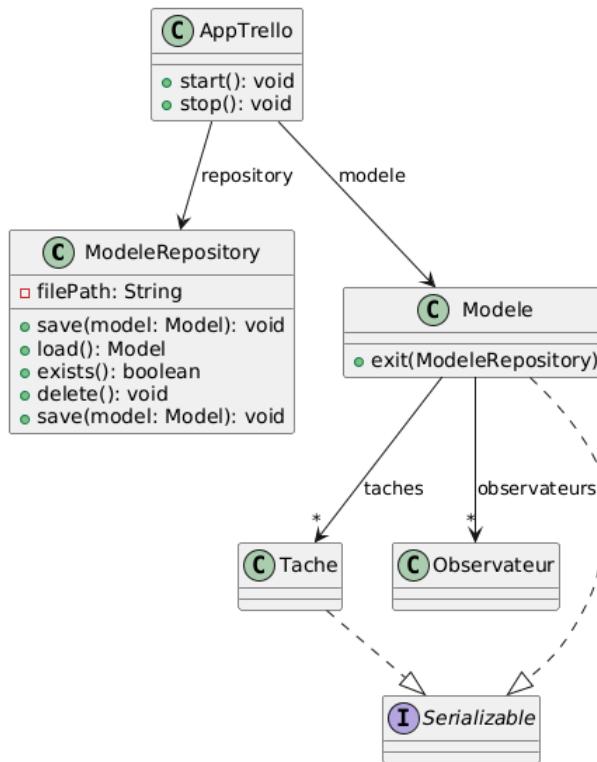
- **Agencement Spatial** : Grâce à la conversion des coordonnées locales en coordonnées de scène, les liens s'adaptent en temps réel au redimensionnement de l'interface et au défilement.
- **Ergonomie Visuelle** : L'usage de **courbes de Bézier cubiques** fluidifie la lecture du chemin critique. La forme en "S" permet d'identifier instantanément les liens de type "Fin-à-Début" tout en évitant la surcharge visuelle des lignes droites sécantes.
- **Réactivité Automatisée** : Le couplage avec le moteur de dépendances du Modèle assure que chaque flèche reflète toujours l'état réel des contraintes du projet, transformant un simple graphique en un outil de planification interactif et fiable.

## Ambroise :

Je choisis de parler de la serialisation, et de la façon dont je l'ai incorporée à l'application déjà existante et fonctionnelle. L'architecture utilisée, fondée sur la séparation entre le [Modèle](#) et la persistance des données via le pattern Repository, présente plusieurs avantages majeurs en termes de qualité logicielle, d'évolutivité et de maintenabilité.

En isolant la logique métier dans de la classe [Modèle](#) et en déléguant toute la responsabilité de l'accès aux données à une classe dédiée ([ModeleRepository](#)), on respecte strictement le principe de responsabilité unique (SRP) : chaque classe n'a qu'un seul rôle.

Il subsiste des points d'améliorations : Une abstraction du repository aurait permis de changer la technologie de persistance (passer d'une sérialisation Java à du JSON, une base relationnelle ou une API distante), simplement en fournissant une nouvelle implémentation du repository. De plus le patron Singleton aurait été pertinent dans la mesure où il n'y a besoins que d'un et un seul Repository dans l'application. De manière générale, quand il s'agit de repository (classe dédié à l'accès et à la modification de donnée) il apparaît pertinent de mettre en place le patron Singleton



```
@startuml

class ModeleRepository {
    - filePath: String
    + save(model: Model): void
    + load(): Model
    + exists(): boolean
    + delete(): void
    + save(model: Model): void
}
```

```
class Modele {
    +exit(ModeleRepository)
}
```

```
class Tache {
}
```

```
class Observateur {
}
```

```
interface Serializable {
}
```

```
class AppTrello {
    + start(): void
    + stop(): void
}
```

```
AppTrello --> ModeleRepository : repository
AppTrello --> Modele : modele
Modele --> "*" Tache : taches
Modele --> "*" Observateur : observateurs
Modele ..|> Serializable
Tache ..|> Serializable
```

```
@enduml
```

## Problèmes rencontrés

### Observateurs non serializable :

Il n'y a aucun intérêt à serialiser les vues puisqu'elles peuvent être reconstruites à partir du `Modele`. Pour serialiser le `Modele`, il faut deux choses, que le `Modele` implémente `Serializable` et que tous ses attribut soit `Serializable`. Or les observateurs ne sont pas `Serializable` ce qui renvoie une erreur `java.io.NotSerializableException`. Pour palier à ce problème j'ai fait une petite méthode dans le `Modele` qui s'appelle `exit` et qui vide la liste des observateurs juste avant de serialiser avec le repository passer en paramètre de la méthode `exit`

```
public void exit(Object repo) {
    this.observateurs = new ArrayList<>();
    ((ModeleRepository) repo).save(this);
}
```

## UID de serialisation :

Au début, l'absence de définition explicite du champ `serialVersionUID` a posé un problème de compatibilité lors de la déserialisation des objets. En effet, lorsque ce champ n'est pas déclaré, la JVM génère automatiquement un identifiant de version à partir de la structure de la classe (attributs, méthodes, modificateurs, hiérarchie). Le problème est qu'à chaque fois que la classe est modifiée (ce qui arrive souvent étant donné que le `Modele` est central dans l'application) cet identifiant change et provoque une exception de type `InvalidClassException` lors de la lecture d'objets précédemment sauvegardés. Ainsi il fallait supprimer la sauvegarde et repartir avec une application sans tâche. Le `serialVersionUID` correspond à un identifiant de version explicite utilisé par le mécanisme de sérialisation Java pour vérifier la compatibilité entre la classe ayant servi à sérialiser un objet et la classe présente au moment de la déserialisation. En le définissant manuellement dans `Modele` et `Tache`, il devient possible de contrôler cette compatibilité et d'autoriser la déserialisation d'objets anciens tant que les évolutions de la classe restent compatibles. L'ajout explicite du `serialVersionUID` a donc permis de stabiliser le mécanisme de persistance, d'éviter les erreurs de déserialisation et de rendre l'évolution du modèle plus maîtrisée.

```
private static final long serialVersionUID = 1L;
```

## Fonctionnement technique

la classe `ModeleRepository` repose principalement sur les deux méthodes `save` et `load` qui permettent respectivement de sauvegarder dans un fichier dont le nom est le seul attribut de la classe `ModeleRepository` et de charger un objet `Modele` à partir de ce même fichier.

Ainsi `save` prend en paramètre un `Modele` et `load` renvoie un `Modele`  
dans les deux cas j'utilise la syntaxe `try` avec en argument la déclaration d'une variable (`ObjectOutputStream` ou `ObjectInputStream`) se qui permet de fermer le flux automatiquement  
puis on a la serialisation/deserialisation (`out.writeObject` et `in.readObject`)

```
//save :  
try ( ObjectOutputStream out = new ObjectOutputStream(FileOutputStream(filePath))) {  
    out.writeObject(modele);  
}  
  
//load :  
try ( ObjectInputStream in = new ObjectInputStream(FileInputStream(filePath))) {  
    Modele modele = (Modele) in.readObject();  
}
```

## Avantages de la serialisation

La serialisation permet de sauvegarder l'état de l'application de manière simple et rapide. Elle a également l'avantage de pouvoir enregistrer plusieurs sauvegardes de l'application, j'ai d'ailleurs réfléchis à pouvoir choisir depuis l'application, à son démarrage, quelle sauvegarde choisir parmi toutes celles disponibles. Mais cette fonctionnalité n'a jamais vu le jour et pour changer de sauvegarde on modifiait directement dans la déclaration du repository dans le main (`AppTrello`) la valeur de l'attribut `filePath`.

## Yanis :

### 1. Contexte et Problématique

Dans le cadre de ce projet, nous avons implémenté une structure où chaque tâche peut contenir des sous-tâches, sans limitation de profondeur. Cela nous oblige à passer d'une simple liste linéaire à une **structure en Arbre** (chaque nœud peut avoir N enfants).

#### Le défi technique :

Nous avons opté pour une **relation unilatérale** : les parents connaissent leurs enfants, mais les enfants ne connaissent pas leurs parents. Cela simplifie la sérialisation mais complique les opérations de suppression et de déplacement, nécessitant l'usage d'algorithmes récursifs.

## 2. Architecture de Données : Le Patron Composite

L'architecture repose sur le **Patron Composite**. Le Modèle ne stocke dans sa liste principale (`taches`) que les **Racines** de l'arbre (les tâches de niveau 0). Les sous-tâches sont encapsulées à l'intérieur des objets parents.

Cela implique deux modes d'accès aux données :

1. **Accès Direct** : Pour les tâches racines (via `modele.taches`).
2. **Accès Récursif** : Pour n'importe quelle sous-tâche (en parcourant l'arbre).

## 3. Implémentation de la Recherche

Puisque l'enfant ignore qui est son père, nous devons retrouver cette information par déduction. Nous utilisons un algorithme de recherche de parcours en Profondeur.

**Logique** : On interroge chaque racine. Si ce n'est pas le père, la racine interroge ses enfants, qui interrogent leurs enfants, et ainsi de suite.

**Extrait du code** (`Modele.java`) :

```
private Tache chercherParentDirectRécursif(Tache parentActuel, Tache cible) {  
    // 1. Condition d'arrêt (Succès) : Le parent actuel possède la cible  
    if (parentActuel.getEnfants().contains(cible)) {  
        return parentActuel;  
    }  
  
    // 2. Appel Récursif (Propagation)  
    for (Tache enfant : parentActuel.getEnfants()) {  
        // L'appel s'empile ici tant qu'on ne trouve pas  
        Tache res = chercherParentDirectRécursif(enfant, cible);  
        if (res != null) return res; // On fait remonter le résultat  
    }  
  
    // 3. Condition d'arrêt (Échec) : Fin de la branche sans succès  
    return null;  
}
```

*Justification* : La récursivité est ici indispensable car nous ne connaissons pas la profondeur de l'arbre à l'avance. Une boucle `for` imbriquée serait impossible à écrire pour une profondeur infinie.

## 4. Gestion de la Suppression

La suppression d'une tâche est une opération critique car elle doit gérer la mémoire et l'intégrité des données. Elle se décompose en deux temps.

### Étape A : Maintien de la cohérence (Détachement)

Avant de supprimer la tâche, il faut supprimer le lien avec son parent.

```

Tache parent = getParentDirect(tache); // Utilisation de l'algorithme précédent
if (parent != null) {
    parent.supprimerEnfant(tache); // Le parent "lâche" l'enfant
}

```

## Étape B : Nettoyage Récuratif

Une fois détachée, nous lançons une suppression en cascade pour garantir qu'aucune trace de la tâche ou de ses descendants ne subsiste dans la liste principale (`taches`).

**Extrait du code ( `Modele.java` ) :**

```

private void supprimerRecuratif(Tache t) {
    // 1. Tentative de suppression dans la liste racine (Sécurité)
    taches.remove(t);

    // 2. Propagation aux enfants
    if (t.aDesEnfants()) {
        // Copie de la liste pour éviter ConcurrentModificationException
        List<Tache> enfants = new ArrayList<>(t.getEnfants());
        for (Tache enfant : enfants) {
            supprimerRecuratif(enfant); // Appel récuratif
        }
    }
}

```

### Analyse technique :

- Si `t` est une sous-tâche, `taches.remove(t)` ne fait rien (car elle n'est pas dans la liste racine).
- Si `t` est une racine, elle est supprimée.
- Cette méthode unique gère donc tous les cas de figure (Racine, Enfant, ou "Tâche Fantôme" mal placée).

Initialement, notre projet gérait les tâches via une liste linéaire simple. La suppression récursive était alors impérative pour identifier et supprimer manuellement chaque descendant.

Lorsque nous avons reimplémenté le patron composite, nous avons techniquement délégué la gestion du cycle de vie des objets au *Garbage Collector* de Java (la suppression de la racine entraînant la libération de la branche).

Cependant, nous avons fait le choix de **maintenir volontairement** l'algorithme de suppression récursive comme mécanisme de **programmation défensive**.

## 5. Propagation d'État

La récursivité est également utilisée pour propager des changements visuels ou logiques.

### Exemple : Déplacement de colonne

Si on déplace une tâche "Projet Web" vers la colonne "Terminé", tous ses enfants doivent suivre.

```

private void deplacerColonneRecuratif(Tache t, String nouvelleColonne) {
    // 1. Action locale
    t.setColonne(nouvelleColonne);

    // 2. Propagation
}

```

```

if (t.aDesEnfants()) {
    for (Tache enfant : t.getEnfants()) {
        deplacerColonneRecursif(enfant, nouvelleColonne);
    }
}
}

```

Nous avons également implémenté une contrainte forte : avant de lancer cette récursivité, nous vérifions via `getParentDirect` si la tâche est une sous-tâche. Si oui, nous interdisons le déplacement si la colonne cible diffère de celle du parent.

## Conclusion

Pour conclure, ce module de gestion de tâches hiérarchiques a constitué le défi technique majeur du projet. Il a nécessité de passer d'une vision linéaire des données à une logique structurelle en arbre.

J'ai particulièrement retenu l'importance de la **récursivité**, non seulement pour l'affichage ou la recherche, mais aussi comme outil de maintenance des données.

[Rapport Yanis : Gestion Hiérarchique des Tâches par Récursivité](#)

## Liste des itération

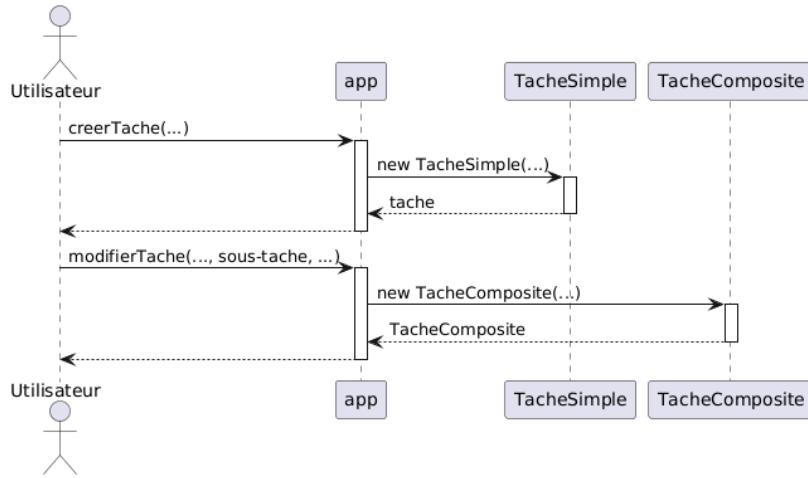
### Itération 1 : Fondations & Conception durable

**Objectif :** Avoir une application qui se lance, stocke des données en mémoire et permet la gestion de base.  
Puisque vous avez le diagramme, c'est le moment de le coder.

- **Fonctionnalités :**
  - Implémentation Vue : observateur → ambroise
  - **Implémentation du Modèle :** Codage des classes métier (Tâche, Liste) d'après votre diagramme. → jassem
  - **CRUD Tâche (Partiel) :** Créer et Supprimer une tâche simple (Titre, Description, Date).
  - **Vue Liste :** Affichage simple des tâches sous forme de liste textuelle ou tableau.
- **Technique :**
  - Mise en place du projet JavaFX et Git. → ambroise
  - Mise en place du pattern **MVC**.
  - Pattern **Factory** pour la création des tâches (si prévu).
- *Livrable* : Une fenêtre qui affiche une liste où l'on peut ajouter/supprimer une ligne.

### Resultat :

- Classes fonctionnelles
- Main Textuel → jassem
- Création des tâches
- Déplacer les tâches

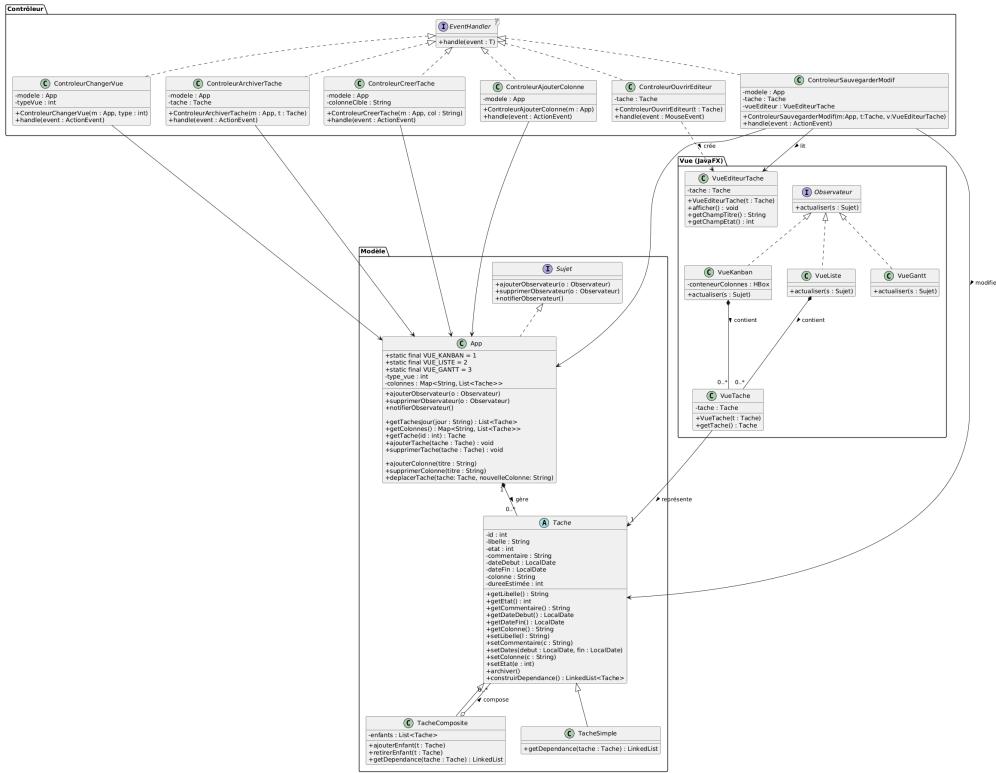


- **Architecture MVC** : Nous avons structuré le projet en packages distincts (Modele, Vue, Controleur) pour respecter la séparation des préoccupations.
- **Modèle de Données** : Création de la classe abstraite `Tache` définissant les attributs communs (libellé, état, durée, couleur). Nous avons défini les états via des constantes (À faire, En cours, Terminé, Archivé).
- **Pattern Observer** : Mise en place des interfaces `Sujet` et `Observateur`. La classe `Modele` centralise les données et notifie automatiquement les vues lors des opérations CRUD (Création, Lecture, Mise à jour, Suppression).
- **Tests** : Validation du modèle via une interface textuelle (`MainTextuel`) avant de passer à l'interface graphique.

## Itération 2 : Controleurs & vue Kanban

**Objectif** : Ajouter l'intelligence de l'application (règles de gestion).

- **Fonctionnalités :**
  - **Gestion des Dépendances** : Lier une tâche A à une tâche B (B bloque A).
  - **Gestion des États** : Une tâche passe automatiquement en "Bloquée" ou change de couleur si sa dépendance n'est pas finie.
  - **CRUD Tâche (Complet)** : Modification complète (dates, description détaillée) et "Cocher" une tâche comme finie.
- **Technique :**
  - Pattern **State** : Pour gérer proprement les transitions (En cours → Fini, Bloqué → En cours).
  - Validation logique lors du Drag & Drop (interdire le déplacement si bloqué).
- **Livrable** : Impossible de finir une tâche si la précédente n'est pas faite.



- **Vue Kanban** : Développement de l'affichage en colonnes dynamiques. Chaque tâche est représentée par une "carte" colorée affichant ses informations clés.
- **Édition** : Création de la `VueEditeurTache`, une fenêtre modale permettant de modifier tous les attributs d'une tâche (titre, commentaire, état, couleur, durée via Spinner).
- **Interactions** : Implémentation des premiers contrôleurs pour créer une tâche, l'archiver ou ouvrir l'éditeur au double-clic.
- **Drag & Drop** : Première implémentation du glisser-déposer basique pour changer une tâche de colonne.

## Itération 3 : Vue Kanban + Dépendances + Gestion des jours + gestion de l'état de la tâche + Vue Liste

**Vue Liste** : Ajout d'un affichage tabulaire organisé par jour. Nous avons introduit une indentation visuelle (`↪`) pour représenter la hiérarchie des tâches.

- **Refonte temporaire** : Pour gérer les dépendances rapidement, nous avons temporairement utilisé des listes simples dans la classe `Tache`, mettant de côté le pattern Composite pour se concentrer sur l'affichage.
- **Drag & Drop (v2)** : Amélioration du déplacement des tâches avec un feedback visuel (mise en surbrillance de la zone de dépôt) pour une meilleure fluidité.

### Travail dans la branche `Kanban` & `VueListe`

Pour tester, on ouvre `MainKanban` qui lance une vue Kanban.

#### CRUD Tache

- **Lecture/Affichage** : Mise à jour de `VueKanban` pour afficher les nouvelles informations (Jour, État sous forme de pastille).
- **Mise à jour (Édition)** : Refonte de `VueEditeurTache` (ajout sélecteurs Colonne, Jour, État) et mise à jour de `ControleurSauvegarderModif` pour persister les changements dans l'objet.
- **Archivage** : Utilisation de `ControleurArchiverTache` et filtrage dans `Model` (`getTaches()` ignore les archivées).

## CRUD Colonne

- **Modèle :** Ajout des méthodes `renommerColonne` et `supprimerColonne` dans `Modele.java` (avec gestion de sécurité : les tâches d'une colonne supprimée vont dans "À faire").
- **Contrôleurs :** Création de `ControleurAjouterColonne`, `ControleurRenommerColonne` et `ControleurSupprimerColonne`.
- **Vue :** Adaptation de l'en-tête dans `VueKanban` pour inclure les boutons d'actions (Renommer/Supprimer) à côté du titre.

## Gestion des dépendances :

- **Architecture :** Suppression du Pattern Composite strict (suppression des fichiers `TacheSimple` et `TacheComposite`). La classe `Tache` devient concrète et intègre une liste `enfants` et les méthodes de gestion (`ajouterEnfant`, `aDesEnfants`).
- **Création :** Ajout d'une `ComboBox` dans la `Dialog` de `ControleurCreerTache` listant toutes les tâches existantes pour rattacher la nouvelle tâche à un parent.
- **Affichage :** Modification de la méthode `creerCarteTache` dans `VueKanban` pour itérer et afficher visuellement les sous-tâches dans la carte du parent.

## Dates → Jours de semaine

Remplacement de la gestion complexe `LocalDate` par une approche catégorielle (Jours de la semaine).

- **Modèle :** Dans `Tache`, remplacement des attributs `dateDebut` / `dateFin` par un String `jour` (validé par un Set static `JOURS_AUTORISES`).
- **Contrôleurs & Vue :** Adaptation de `VueEditeurTache` (remplacement `DatePicker` par `ComboBox`), de `ControleurSauvegarderModif` et de l'affichage dans `VueKanban`.

## Début de drag & drop (fonctionne seulement en interaction avec une autre Tâche)

Interaction : `VueKanban` gère les événements `DragDetected`, `DragOver` et `DragDropped` pour transférer la référence de l'objet `Tache`.

Ajout d'une colonne par défaut appellée `Principale`

## Travail dans la branche `VueListe`

### Objectif

Fournir une vue analytique des tâches, structurée **chronologiquement** (par jour) et **hiérarchiquement** (parents/enfants), en complément de la vue Kanban.

### Algorithmes et Logique d'Affichage

- **Gestion des Doubles (Parent/Enfant) :**
  - *Problème* : La liste des tâches étant plate, les sous-tâches risquaient d'apparaître deux fois (à la racine et sous leur parent).
  - *Solution* : Utilisation d'un **pré-filtrage**. Avant l'affichage, tous les enfants sont identifiés et stockés dans un `HashSet`. Lors du rendu des jours, un filtre (`stream().filter()`) exclut ces tâches de l'affichage racine.
- **Rendu Hiérarchique (Récursivité) :**
  - L'affichage repose sur une méthode récursive `ajouterLigneTache`.
  - Elle accepte un paramètre de profondeur (`niveauIndent`).
  - À chaque appel récursif pour un enfant, ce niveau s'incrémentera, augmentant le **Padding gauche** (décalage de 20px par niveau) et ajoutant le symbole "`↳`".

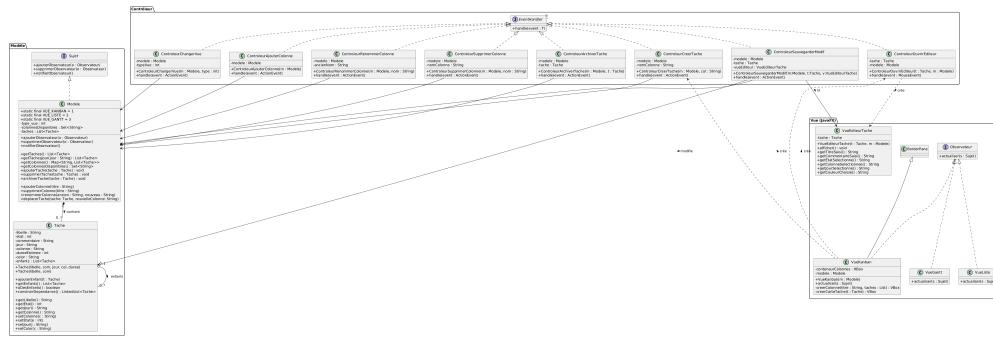
- **Alignement Strict ( GridPane ) :**

- Utilisation d'un `GridPane` unique par jour plutôt que des `HBox` imbriquées.
- Cela garantit que les colonnes (État, Durée, Commentaire) restent parfaitement alignées verticalement, quelle que soit la profondeur de la tâche dans l'arborescence.

## Architecture et Réutilisation

- **Intégration Directe** : La vue n'introduit aucun nouveau contrôleur. Elle instancie directement les classes existantes (`ControleurCreerTache`, `ControleurArchiverTache`, `ControleurOuvrirEditeur`) sur les composants graphiques.
- **Pattern Observateur** : La vue s'abonne au modèle (`implements Observateur`) pour se rafraîchir automatiquement lors des modifications (CRUD), garantissant la synchronisation avec le Kanban.

## Diagramme de classe



## Itération 4 : Retablir le patron + Serialisation + Composite + Vue Gantt

### Rétablissement du Pattern Composite : La Promotion Dynamique

Dans l'itération précédente, nous avons supprimé le patron composite dans un soucis de simplicité : puisque chaque tâche peut en théorie accueillir une sous-tâche.

Mais à la demande du professeur, nous devons le rétablir.

#### Le Problème : Immutabilité du Type

En Java, le type d'un objet est défini à l'instanciation. Une `TacheSimple` ne peut pas se transformer en `TacheComposite` pour accueillir des enfants.

**Solution :** Remplacer l'objet original par un clone du bon type, tout en conservant ses données (titre, état, couleur).

#### Le Mécanisme : "Copier-Remplacer"

Le processus se déroule en trois étapes clés, transparentes pour l'utilisateur :

- **Détection** : Le Contrôleur repère si le parent cible est une tâche simple.
- **Promotion** : Le Modèle crée une `TacheComposite` identique à l'ancienne tâche, échange les références dans la liste principale et notifie les vues.
- **Ajout** : La sous-tâche est ajoutée au nouveau parent.

### 3. Implémentation Technique

#### Côté Modèle (La Promotion) :

La méthode crée le "jumeau" Composite et effectue la substitution en mémoire.

```

public TacheComposite promouvoirEnComposite(TacheSimple ancienneTache) {
    // 1. Duplication des données dans le nouveau type
    TacheComposite nouvelleTache = new TacheComposite(ancienneTache);

    // 2. Substitution dans la liste principale
    int index = taches.indexOf(ancienneTache);
    if (index != -1) taches.set(index, nouvelleTache);

    // 3. Notification des vues
    notifierObservateur();
    return nouvelleTache;
}

```

#### Côté Contrôleur (Le Déclencheur) :

Le contrôleur gère la logique de décision avant d'ajouter l'enfant.

```

// Dans le Contrôleur lors de la création
if (parentSelectionne instanceof TacheSimple) {
    // Promotion dynamique nécessaire
    TacheComposite nouveauParent = modele.promouvoirEnComposite((TacheSimple) parentSelectionne);
    nouveauParent.ajouterEnfant(nouvelleTache);
} else {
    // Ajout standard
    parentSelectionne.ajouterEnfant(nouvelleTache);
}

```

#### Pourquoi cette approche ?

- **Intégrité** : Aucune perte de données lors de la transition.
- **Architecture** : Respect strict du Pattern Composite (séparation Feuille/Nœud).
- **Transparence** : L'utilisateur transforme une tâche en projet sans s'en rendre compte.

## La Persistance des Données (Sérialisation)

**Objectif** : Permettre à l'utilisateur de retrouver ses tâches, colonnes et configurations après avoir fermé et relancé l'application.

#### Classes concernées :

- `Modele/ModeleRepository.java` (Nouvelle classe : Gestionnaire E/S)
- `Modele/Modele.java` (Modification : Rendue sérialisable)
- `Modele/Tache.java` (Modification : Rendue sérialisable)
- `AppTrello.java` (Modification : Chargement au démarrage, Sauvegarde à la fermeture)

## Le Concept : Java Sérialisation

Nous avons utilisé le mécanisme natif de Java pour transformer nos objets en flux d'octets. Cela nécessite de "marquer" les classes racines avec l'interface `Serializable`.

**Code** (`Modele.java` & `Tache.java`) :

```
// L'interface Serializable agit comme un marqueur pour la JVM
public class Modele implements Sujet, Serializable {
    // Versionnage pour garantir la compatibilité lors de la désérialisation
    private static final long serialVersionUID = 1L;
    // ...
}
```

Note : Tous les sous-objets (comme `LocalDate` dans `Tache` ou les `ArrayList`) sont déjà sérialisables nativement en Java, ce qui facilite le processus.

### Le Gestionnaire : Pattern Repository

Pour ne pas polluer le `Modele` avec des détails techniques d'écriture de fichiers, nous avons créé une classe dédiée : `ModeleRepository`. Elle encapsule la complexité des `ObjectOutputStream` et `ObjectInputStream`.

**Code ( `ModeleRepository.java` ) :**

```
public void save(Modele modele) {
    try (FileOutputStream fileOut = new FileOutputStream(filePath);
        ObjectOutputStream out = new ObjectOutputStream(fileOut)) {
        // Écriture de l'objet entier (et de toute son arborescence) dans le fichier
        out.writeObject(modele);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

### Le Défi Technique :

Le `Modele` contient une liste d'`observateurs` (les Vues JavaFX). Or, les composants graphiques JavaFX (comme `VueListe`) ne sont **pas sérialisables**. Tenter de les sauvegarder provoquerait un crash (`NotSerializableException`).

**Solution :**

Vider la liste des observateur (à ignorer lors de la sauvegarde).

**Code :**

```
public void exit(Object repo) {
    this.observateurs = new ArrayList<>();
    ((ModeleRepository) repo).save(this);
}
```

Le `Modele` contient une liste de `Tache`. Or une tache est une classe abstraite qui ne peux pas être reconstruite à la lecture du fichier de sauvegarde.

**Solution :**

créer un constructeur vide dans la classe `Tache` (en suivant les conseils d'un autre groupe)

**Code :**

```
protected Tache() {
    this.libelle = "";
    this.commentaire = "";
    this.etat = this.ETAT_A_FAIRE;
    this.colonne = "";
    this.jour = "";
```

```

        this.dureeEstimee = 0;
        this.color = "#C5D3D0";
    }
}

```

### Intégration au Cycle de Vie ( [AppTrello](#) )

L'application charge les données au lancement ( [start](#) ) et sauvegarde automatiquement à l'arrêt ( [stop](#) ).

#### Code ( [AppTrello.java](#) ) :

```

@Override
public void start(Stage primaryStage) {
    // Tentative de chargement
    repository = new ModeleRepository("Sauvegarde/app.save");
    modele = repository.load();

    // Si premier lancement (fichier inexistant), on injecte des données de test
    if (modele == null) {
        modele = new Modele();
        initDonneesTest(modele);
    }
    // ... Lancement des vues
}

@Override
public void stop() throws Exception {
    // Sauvegarde automatique quand on clique sur la croix rouge
    if (repository != null && modele != null) {
        modele.exit(repository);
    }
    super.stop();
}

```

### Vue Gantt

**Vue Gantt** : Création d'un diagramme temporel. L'affichage s'adapte automatiquement à l'échelle de temps (date min/max) et dessine les barres proportionnellement à la durée des tâches.

## Iteration 5 : Gestion des dates + Propagation des actions sur les enfants

### Changement majeur: Remplacement des jours par des dates

Le professeur nous a initialement suggéré de remplacer nos LocalDates par des Jours (en String).

Ensuite, lors de la soutenance concernant la [VueGantt](#), nous avons constaté la pertinence de l'approche des attributs dates de type [LocalDate](#).

#### Classe [Tache](#) (Abstraite)

- **Attributs modifiés/ajoutés :**

- [protected LocalDate dateDebut](#) : (Nouveau) Remplace [String jour](#). Stocke la date réelle.
- [protected int dureeEstimee](#) : (Modifié) Représente maintenant une durée en **jours** (et non plus en heures).

```

public abstract class Tache implements Serializable {
    //...
    protected LocalDate dateDebut;
}

```

```

protected int dureeEstimee;
//...
}

```

- **Méthodes principales :**

- `getDateFin()` : (Nouvelle) Calcule automatiquement `dateDebut + dureeEstimee`.
- `getNomJour()` : (Nouvelle) Retourne le nom du jour (ex: "Lundi") à partir de la date.
- `ajouterEnfant(Tache t)` : (Abstraite) Force les sous-classes à définir le comportement d'ajout.

## Package Controleur

Adaptation de la logique pour manipuler des objets temporels.

### Classe ControleurCreerTache

- **Modifications :**

- Ajout d'un `DatePicker` dans la boîte de dialogue.
- Logique de détection : Si le parent choisi est `TacheSimple`, appel automatique de `modele.promouvoirEnComposite()`.

### Classe ControleurSauvegarderModif

- **Modifications :**

- Récupère `LocalDate` depuis la vue via `vue.getDateSelectionnee()`.
- Met à jour le modèle avec `tache.setDateDebut()`.

## 3. Package Vue

Refonte de l'affichage pour gérer le temps dynamique.

### Classe VueListe

- **Méthodes principales :**

- `rafrachirDonnees(List<Tache>)` : (Refondue)
  - Calcule `minDate` et `maxDate` de tout le projet.
  - Boucle `while` du premier au dernier jour pour générer les sections dynamiquement.
  - Sécurisée contre les dates `null`.

### Classe VueGantt

- **Méthodes principales :**

- `calculerLimitesTemporelles(List<Tache>)` : (Nouvelle) Détermine le début et la fin du graphique Gantt + marge.
- `construireHeader()` : (Nouvelle) Génère les colonnes de dates dynamiquement (plus de "Lundi...Dimanche" fixe).
- `ajouterLigneTache(...)` : Calcule la position X et la largeur de la barre en pourcentage (Binding) basé sur l'écart en jours (`ChronoUnit.DAYS`).

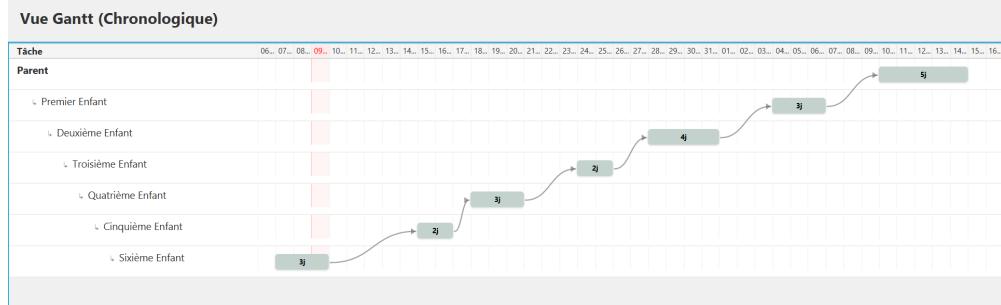
## Iteration 6: Derniers fix + peauffinage

### Cohérence et hiérarchie des dates

Ce que je vais appeler cohérence fait référence au fait qu'une tâche parente ne puisse pas commencer tant que toutes ses tâches filles n'ont pas fini. Ce principe de fonctionnement s'inspire du "jeu" de gestion CESIM. Jusqu'ici, la vérification de la cohérence ne se faisait qu'à la création de la tâche (fonctionnalité implémentée dans l'itération précédente). Avec les méthodes récursives implémentées précédemment, il m'a été possible de faire un parcours de toute l'arborescence des tâches pour appliquer un décalage récursif. La façon dont ça fonctionne concrètement d'un point de vue utilisateur est que lorsqu'une tâche est déplacée, toutes les tâches dont elles dépendent ou qui sont dépendantes d'elle, sont déplacées pour respecter la règle de cohérence.

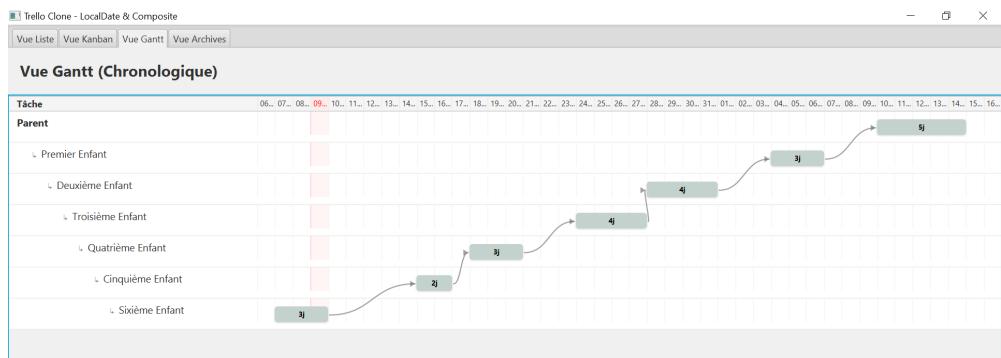
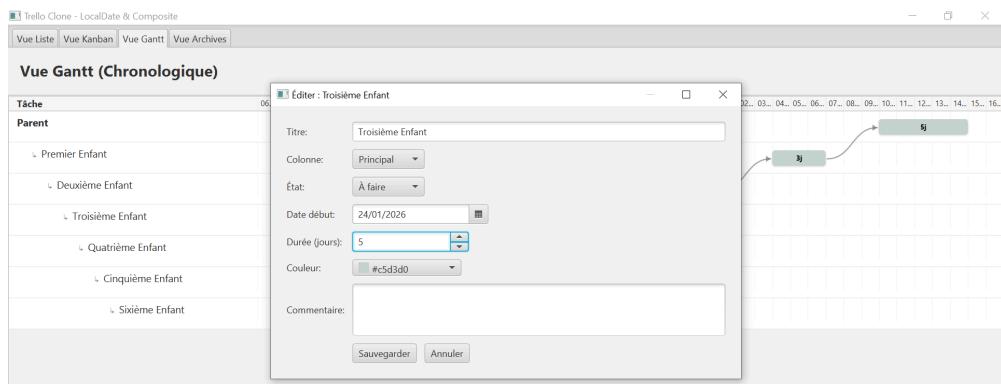
### exemples :

voici la situation initiale

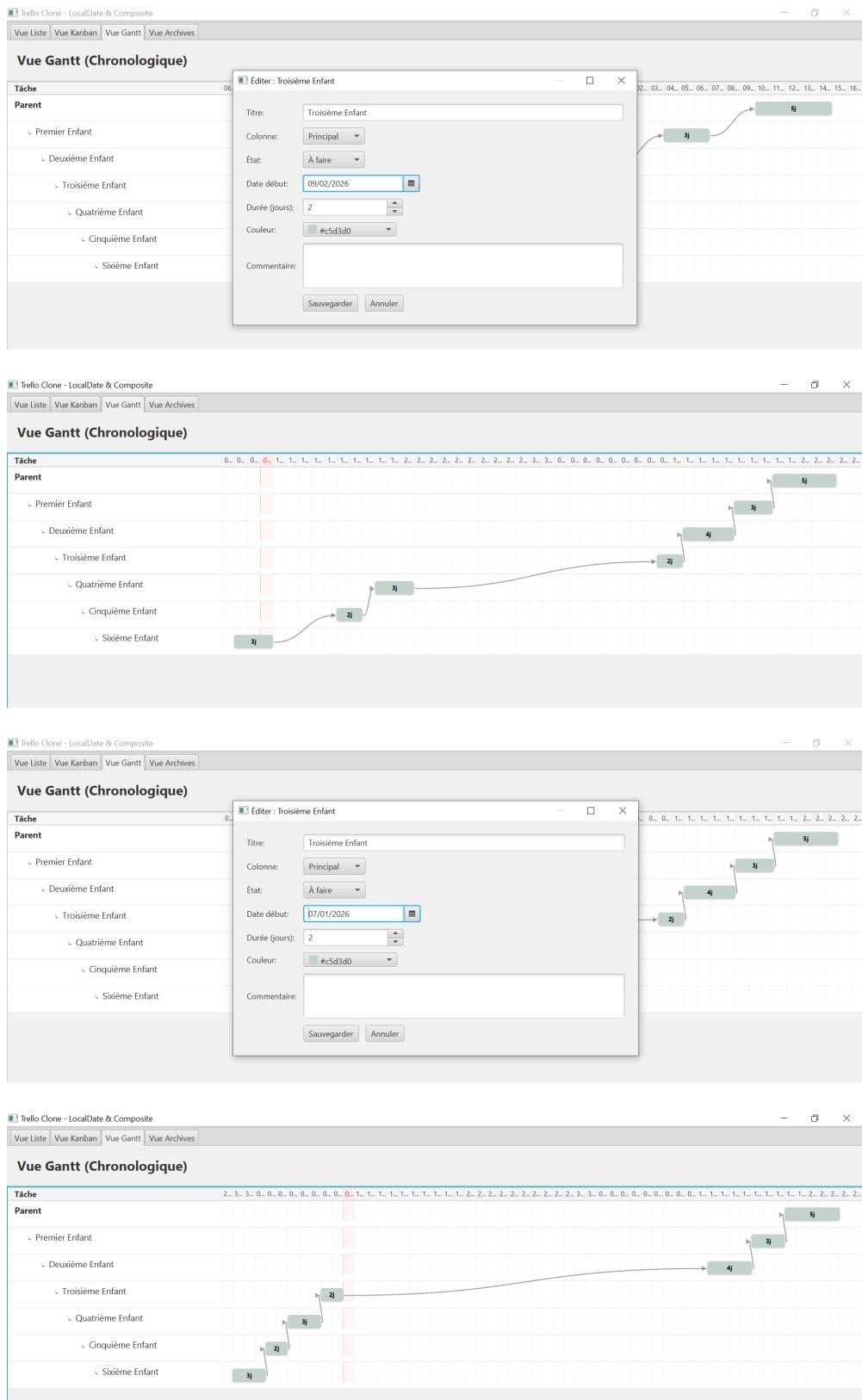


prenons la tâche **Troisième Enfant**

- quand on modifie sa durée au-delà du début de la tâche **Deuxième Enfant** (qui est parente de **Troisième Enfant**) l'application nous en empêche. On voit dans l'exemple que la tâche **Troisième Enfant** est bloqué à quatre jour



- En repartant de la situation initiale et en changeant la date de début de la tâche **Troisième Enfant** on se rend compte que toutes les autres tâche qui dépendent de **Troisième Enfant** et toutes celles dont **Troisième Enfant** dépend se déplace pour conservé une cohérence globale.



## Drag and Drop pour les colonnes

### Objectif et Problème Technique

L'objectif est de permettre la réorganisation des colonnes (ex: intervertir "En cours" et "Terminé").

Le défi technique est que la vue gère déjà le déplacement des tâches. Il fallait donc un moyen fiable pour que le système distingue le déplacement d'une **Tâche** de celui d'une **Colonne**.

## **Solution : Le Marquage par Préfixe**

Nous avons utilisé un "tag" (préfixe) dans le presse-papier lors du glisser-déposer.

- Si le texte commence par `COL`, c'est une colonne.
- Sinon, c'est une tâche.

## **Implémentation**

La logique repose sur trois événements clés dans `VueKanban` :

1. **Départ** (`setOnDragDetected`) : On ajoute le préfixe au nom de la colonne.

```
private static final String PREFIX_COL = "COL|";
// ...
content.putString(PREFIX_COL + titreColonne); // Ex: "COL|En cours"
```

1. **Feedback Visuel** (`setOnDragEntered`) : Si le préfixe est détecté, on affiche une bordure bleue pour indiquer à l'utilisateur où la colonne sera déposée.

2. **Arrivée** (`setOnDragDropped`) : On vérifie le tag avant d'appeler le modèle.

```
if (data.startsWith(PREFIX_COL)) {
    String source = data.replace(PREFIX_COL, ""); // On récupère le vrai nom
    if (!source.equals(target)) {
        modele.deplacerColonneOrdre(source, target); // Réorganisation
    }
}
```

Cela assure que le déplacement des colonnes n'interfère jamais avec celui des tâches.

## **Vue Archivage**

Le point central de cette fonctionnalité est l'utilisation de la **récursivité** dans le Modèle (`archiverRecuratif`).

Lorsque l'utilisateur archive une tâche :

1. Son état passe à `ETAT_ARCHIVE`.
2. L'algorithme vérifie si elle possède des sous-tâches.
3. Si c'est le cas, l'archivage est propagé automatiquement à tous les enfants.

Cela assure l'intégrité des données : on ne peut pas avoir une sous-tâche active dont le parent est archivé.

## **Implémentation des Contrôleurs**

Nous avons ajouté des contrôleurs spécifiques pour gérer ces actions :

- `ControleurArchiverTache` : Déclenché depuis la vue principale (Kanban ou Liste), il appelle la méthode d'archivage du modèle.
- `ControleurDesarchiverTache` : Permet de restaurer une tâche vers l'état "À faire".
- `ControleurSupprimerTache` : Permet la suppression irréversible depuis la corbeille.

## **La Vue dédiée : `VueArchives`**

Nous avons créé une nouvelle classe `VueArchives` pour visualiser ce contenu spécifique. Elle fonctionne différemment des autres vues :

- **Filtrage** : Elle ne demande au modèle que la liste des tâches archivées via `getTachesArchives()`, isolant ces données du reste de l'application.
- **Affichage Chronologique** : Nous avons réutilisé la logique de tri par date (comme dans la Vue Liste) pour regrouper les archives par jour.
- **Actions** : Chaque ligne de tâche propose deux boutons exclusifs à cette vue : la restauration (○) et la suppression définitive (☒).

## Documentation

Commentaires détaillés et documentation (javadoc)

## Cohérence et hiérarchie des dates

Ce que je vais appeler cohérence fait référence au fait qu'une tâche parent ne puisse pas commencer tant que toutes ses tâches filles n'ont pas fini. Ce principe de fonctionnement s'inspire du "jeu" de gestion CESIM. Jusqu'ici la vérification de la cohérence ne se faisait qu'à la création de la tâches (fonctionnalité implémentée dans l'itération précédente). Avec les méthodes récursives implémentées précédemment il m'a été possible de faire un parcours de toute l'arborescence des tâche pour appliquer un décalage récursif. La façon dont ça fonctionne concrètement d'un point de vu utilisateur est que lorsqu'une tâche est déplacé, toutes les tâches dont elles dépendent ou qui sont dépendantes d'elle, sont déplacées pour respecter la règle de cohérence.

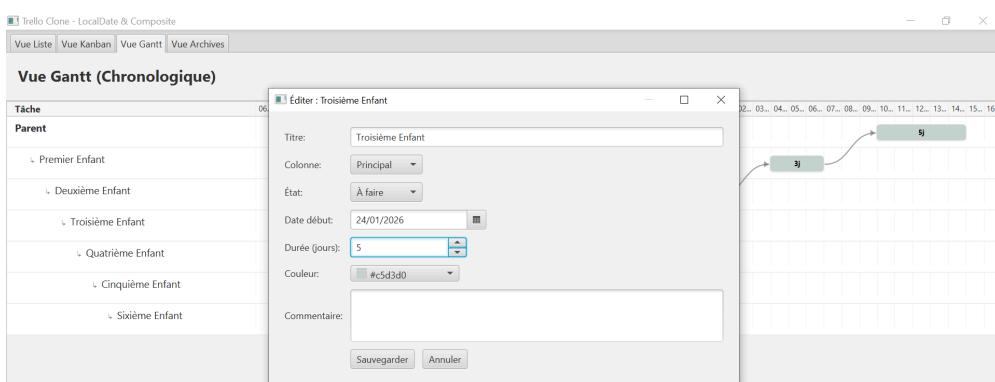
### exemples :

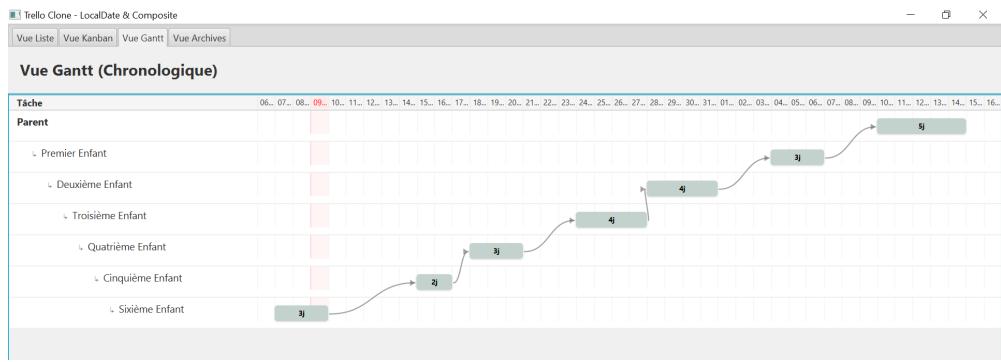
voici la situation initiale



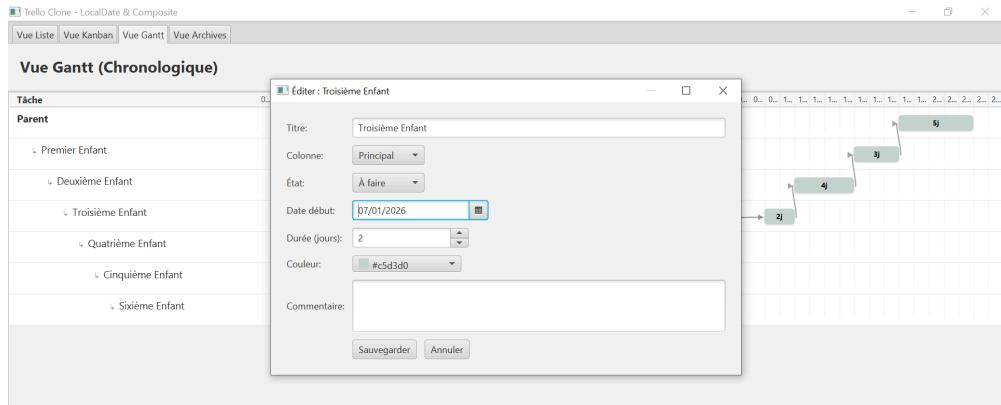
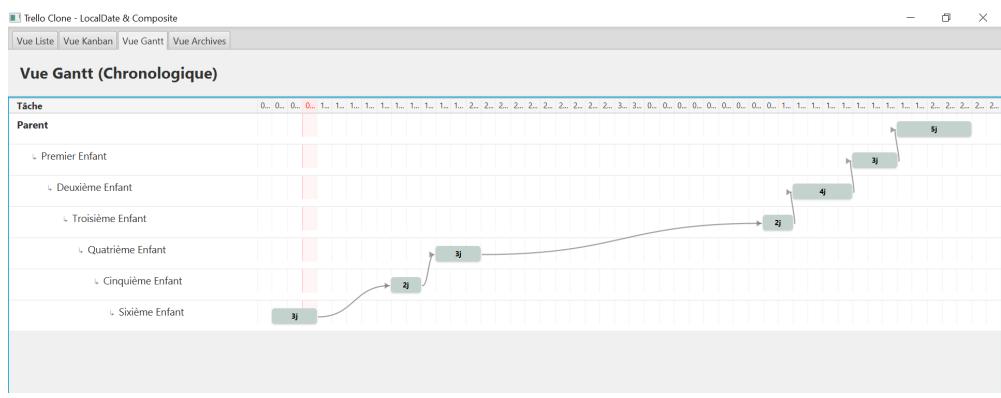
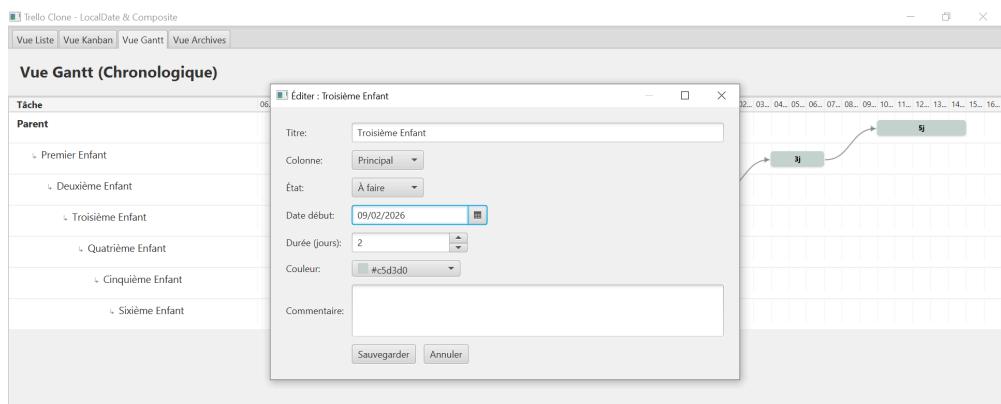
prenons la tâche `Troisième Enfant`

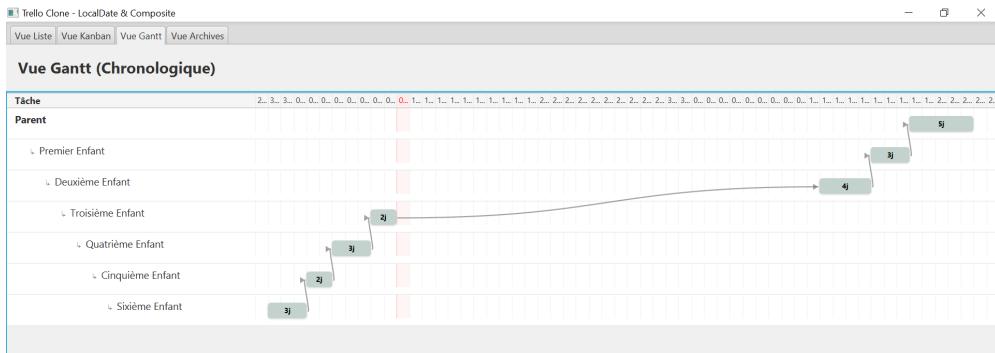
- quand on modifie sa durée au-delà du début de la tâche `Deuxième Enfant` (qui est parent de `Troisième Enfant`) l'application nous en empêche. On voit dans l'exemple que la tâche `Troisième Enfant` est bloqué à quatre jour





- En repartant de la situation initiale et en changeant la date de début de la tâche **Troisième Enfant** on se rend compte que toutes les autres tâche qui dépendent de **Troisième Enfant** et toutes celles dont **Troisième Enfant** dépend se déplace pour conservé une cohérence globale.





## Drag and Drop pour les colonnes

### Objectif et Problème Technique

L'objectif est de permettre la réorganisation des colonnes (ex: intervertir "En cours" et "Terminé").

Le défi technique est que la vue gère déjà le déplacement des tâches. Il fallait donc un moyen fiable pour que le système distingue le déplacement d'une **Tâche** de celui d'une **Colonne**.

### Solution : Le Marquage par Préfixe

Nous avons utilisé un "tag" (préfixe) dans le presse-papier lors du glisser-déposer.

- Si le texte commence par `COL`, c'est une colonne.
- Sinon, c'est une tâche.

### Implémentation

La logique repose sur trois événements clés dans `VueKanban` :

1. **Départ** (`setOnDragDetected`) : On ajoute le préfixe au nom de la colonne.

```
private static final String PREFIX_COL = "COL|";
// ...
content.putString(PREFIX_COL + titreColonne); // Ex: "COL|En cours"
```

1. **Feedback Visuel** (`setOnDragEntered`) : Si le préfixe est détecté, on affiche une bordure bleue pour indiquer à l'utilisateur où la colonne sera déposée.
2. **Arrivée** (`setOnDragDropped`) : On vérifie le tag avant d'appeler le modèle.

```
if (data.startsWith(PREFIX_COL)) {
    String source = data.replace(PREFIX_COL, ""); // On récupère le vrai nom
    if (!source.equals(target)) {
        modele.deplacerColonneOrdre(source, target); // Réorganisation
    }
}
```

Cela assure que le déplacement des colonnes n'interfère jamais avec celui des tâches.

## Vue Archivage

Le point central de cette fonctionnalité est l'utilisation de la **récursivité** dans le Modèle (`archiverRecuratif`). Lorsque l'utilisateur archive une tâche :

1. Son état passe à `ETAT_ARCHIVE`.
2. L'algorithme vérifie si elle possède des sous-tâches.

3. Si c'est le cas, l'archivage est propagé automatiquement à tous les enfants.

Cela assure l'intégrité des données : on ne peut pas avoir une sous-tâche active dont le parent est archivé.

### **Implémentation des Contrôleurs**

Nous avons ajouté des contrôleurs spécifiques pour gérer ces actions :

- **ControleurArchiverTache** : Déclenché depuis la vue principale (Kanban ou Liste), il appelle la méthode d'archivage du modèle.
- **ControleurDesarchiverTache** : Permet de restaurer une tâche vers l'état "À faire".
- **ControleurSupprimerTache** : Permet la suppression irréversible depuis la corbeille.

### **La Vue dédiée : [VueArchives](#)**

Nous avons créé une nouvelle classe [VueArchives](#) pour visualiser ce contenu spécifique. Elle fonctionne différemment des autres vues :

- **Filtrage** : Elle ne demande au modèle que la liste des tâches archivées via `getTachesArchives()`, isolant ces données du reste de l'application.
- **Affichage Chronologique** : Nous avons réutilisé la logique de tri par date (comme dans la Vue Liste) pour regrouper les archives par jour.
- **Actions** : Chaque ligne de tâche propose deux boutons exclusifs à cette vue : la restauration (○) et la suppression définitive (☒).

### **Documentation**

Commentaires détaillés et documentation (javadoc)

## **Eléments qui ont été modifiés par rapport à l'étude préalable**

Ajout de méthodes (pour bon fonctionnement et optimisation du code)

Classe [ModeleRepository](#) : Pour gérer la sérialisation

Classes de contrôleur auxquelles nous n'avons pas pensé.

Affichage différent de la maquette.

Contrairement à l'analyse, les tâches doivent d'abord être archivées pour être supprimées.

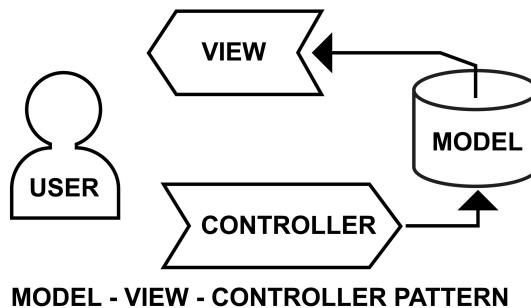
## **Patrons de Conception et Architecture**

Pour garantir modularité et maintenabilité, le projet repose sur trois piliers architecturaux majeurs.

### **1. Architecture MVC (Modèle-Vue-Contrôleur)**

**Rôle** : Séparation des préoccupations pour découpler la logique métier de l'interface graphique.

- **Modèle** ([package Modele](#)) : Cœur de l'application. Il encapsule les données ([Tache](#), [Colonne](#)) et la logique métier complexe (règles de déplacement, calcul de dates, algorithmes récursifs). Il est totalement indépendant de l'interface graphique.
- **Vue** ([package Vue](#)) : Couche de présentation (JavaFX). Elle est "passive" : elle observe le modèle pour se dessiner mais ne modifie jamais les données directement.
- **Contrôleur** ([package Controleur](#)) : Intermédiaire transactionnel. Il intercepte les événements utilisateur (clics, Drag & Drop) et les traduit en actions sur le Modèle.



## 2. Patron Observateur (Observer)

**Rôle :** Synchronisation temps réel entre le Modèle et ses multiples représentations (Vues), assurant un découplage fort.

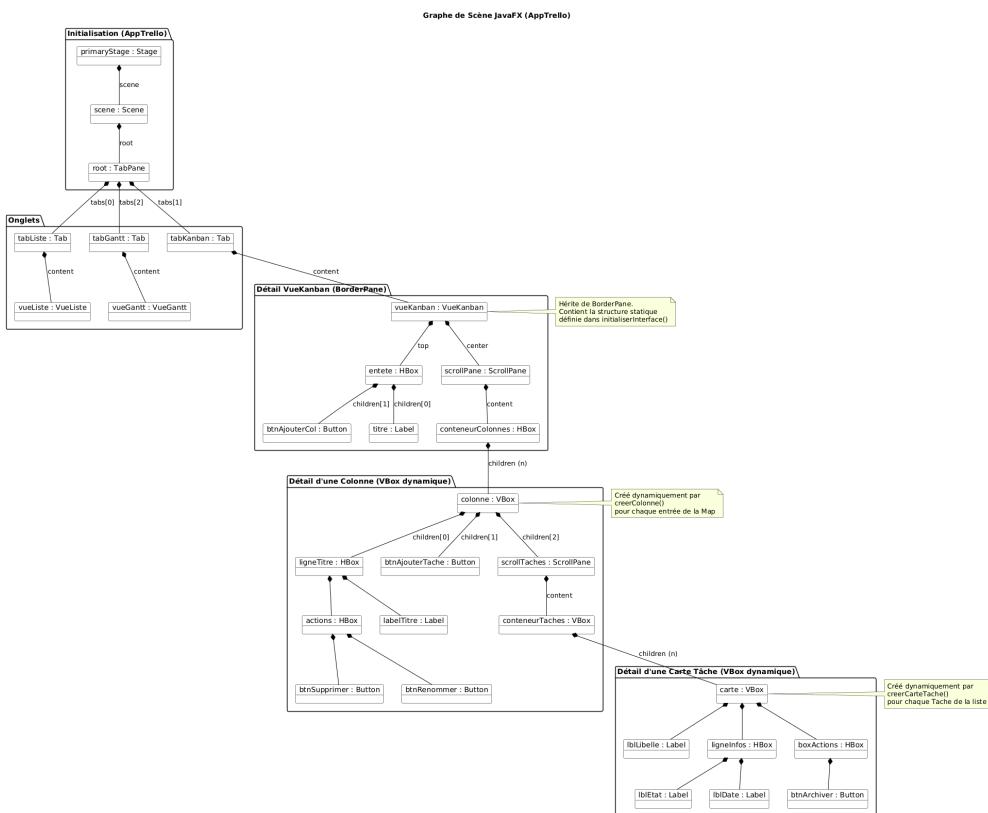
- **Mécanisme :**
  - **Sujet ( `Modele` )** : Maintient une liste d'observateurs via l'interface `Observateur`. À chaque mutation d'état (CRUD), il déclenche `notifierObservateur()`.
  - **Observateur ( `VueKanban` , `VueGantt` , etc.)** : Implémente la méthode `actualiser()`.
- **Avantages Techniques :**
  - **Principe Open/Closed (SOLID)** : L'ajout d'une nouvelle vue (ex: Gantt) ne nécessite aucune modification du Modèle.
  - **Indépendance** : Le modèle n'a aucune dépendance forte vers les vues concrètes, facilitant les tests et le développement parallèle.

## 3. Patron Composite

**Rôle :** Gestion uniforme de la structure hiérarchique des tâches (Arborescence).

- **Structure :**
  - **Composant ( `Tache` - Abstraite )** : Définit le contrat commun (méthodes métier, getters/setters).
  - **Feuille ( `TacheSimple` )** : Unité de travail indivisible.
  - **Composite ( `TacheComposite` )** : Conteneur possédant une liste d'enfants ( `List<Tache>` ).
- **Implémentation et Bénéfices :**
  - **Polymorphisme** : Le code client manipule des objets `Tache` sans se soucier de leur type réel, simplifiant l'affichage dans les Vues.
  - **Puissance Récursive** : Les opérations complexes (suppression, archivage, propagation de dates) sont implémentées une seule fois et ce propagent naturellement dans tout l'arbre via la récursivité.
  - **Promotion Dynamique** : Flexibilité permettant à une `TacheSimple` de devenir `TacheComposite` à la volée lors de l'ajout d'un enfant.

## Graphe de scène Pour la vue Kanban



## 🛠️ Guide d'Installation et d'Exécution sur IntelliJ IDEA

Ce document détaille la procédure pour cloner, configurer et lancer l'application de gestion de projet (Trello) sur l'environnement de développement IntelliJ IDEA.

### 1. Prérequis Techniques

Avant de commencer, assurez-vous que votre environnement dispose des éléments suivants :

- **IDE** : IntelliJ IDEA (Community ou Ultimate).
- **JDK** : Java 21 (Version 21.0.x).
- **Git** : Installé et configuré sur votre machine.

### 2. Récupération du Code Source (Clonage)

Le projet est hébergé sur GitHub.

1. Ouvrez IntelliJ IDEA.
  2. Allez dans le menu **File > New > Project from Version Control**.
  3. Dans la fenêtre qui s'ouvre, collez l'URL SSH suivante :
- ```
git@github.com:JassemMT/S3-01_TAMOURGH-GILBERT-GOFFIN-CHEBAH.git
```
4. Cliquez sur **Clone**.

### 3. Configuration du Projet (JDK et Librairies)

Une fois le projet ouvert, il est normal que le code apparaisse en rouge (erreurs "module not found" dans `module-info.java`). Il faut configurer le SDK et télécharger manuellement les bibliothèques manquantes.

#### Étape A : Configurer le SDK (Java 21)

1. Allez dans **File > Project Structure...** (Raccourci : `Ctrl+Alt+Shift+S`).

2. Cliquez sur l'onglet **Project** (à gauche).
3. Dans le champ **SDK**, assurez-vous que **21** est sélectionné.
  - Si vous ne l'avez pas : Cliquez sur "Add SDK" > "Download JDK" > Version: 21 > Vendor: Oracle OpenJDK (ou autre) > Download.
4. Dans le champ **Language Level**, sélectionnez **SDK Default (21 - ...)**.

## Étape B : Ajouter les librairies JavaFX manuellement

Le fichier `module-info.java` indique qu'il manque `javafx`, `controlsfx` et `bootstrapfx`. Voici comment les ajouter :

1. Toujours dans **Project Structure**, cliquez sur l'onglet **Libraries** (à gauche).
2. Cliquez sur le bouton (en haut de la liste centrale) et choisissez **From Maven....**
3. Une barre de recherche s'ouvre. Vous devez chercher et ajouter les 4 librairies suivantes une par une :
  - **Pour JavaFX Controls** : Tapez `org.openjfx:javafx-controls:21` → Cochez "Download to..." → OK.
  - **Pour JavaFX FXML** : Tapez `org.openjfx:javafx-fxml:21` → Cochez "Download to..." → OK.
  - **Pour ControlsFX** : Tapez `org.controlsfx:controlsfx:11.1.2` (ou version plus récente) → OK.
  - **Pour BootstrapFX** : Tapez `org.kordamp.bootstrapfx:bootstrapfx-core:0.4.0` → OK.
4. Une fois les 4 librairies ajoutées dans la liste, allez dans l'onglet **Modules** (à gauche).
5. Sélectionnez votre module (le nom du projet), puis l'onglet **Dependencies** (à droite).
6. Vérifiez que les 4 bibliothèques que vous venez d'ajouter sont bien listées ici et cochées (Scope: Compile).
7. Cliquez sur **Apply** puis **OK**. Les erreurs rouges dans le code doivent disparaître.

## 4. Lancement de l'Application

Pour démarrer l'application :

1. Dans l'arborescence du projet (à gauche), naviguez vers :
   
`src > main > java > com.example.trello`
2. Faites un clic droit sur la classe `AppTrello` (ou `HelloApplication`).
3. Sélectionnez **Run 'AppTrello.main()'** (l'icône verte ▶).

## 5. Accès aux Fonctionnalités

- **Changer de vue** : Boutons dans l'en-tête
- **Créer une tâche** : Bouton Ajouter tâche en haut d'une colonne.
- **Modifier une tâche** : Double-clic sur une carte.
- **Déplacer une tâche** : Glisser-déposer d'une colonne à une autre.
- **Déplacer une colonne** : Glisser-déposer par l'entête de la colonne (le curseur change et une bordure bleue apparaît).
- **Sous-tâches** : Lors de la création, choisissez un "Parent" pour transformer une tâche en dossier.
- **Archiver** : Bouton "Archiver" sur la carte.
- **Gérer la Corbeille** : Changer de vue vers "Archives" pour restaurer ou supprimer définitivement.