Rapport de projet LAOA

KIRA
Key Intelligent Resource & Activity manager

Auteurs : Dabbebi Mohamed et Djebbi Yesmine

Année : 2025
Cours / Projet : LAOA

Lien GitHub :
https://github.com/mohameddabbebi/Project_QT_CPP-.git

Table des matières

1. Introduction et Modèle de données

1.1. Introduction

1.1.1. Contexte

Le projet **KIRA** s'inscrit dans le cadre de la gestion de projets et de tâches hiérarchiques. Aujourd'hui, la complexité des projets nécessite des outils permettant de visualiser et de gérer efficacement les tâches, leurs dépendances et l'avancement global. L'objectif de KIRA est de fournir une application intuitive qui aide les utilisateurs à organiser leurs projets, suivre le progrès des tâches et gérer les interdépendances entre celles-ci.

1.1.2. Objectifs du projet

Les principaux objectifs de KIRA sont :
— Afficher la structure hiérarchique des tâches et éléments (partie gauche de l'interface).
— Afficher les détails d'un élément sélectionné (partie droite de l'interface).
— Permettre l'édition d'un élément sélectionné.
— Créer de nouveaux éléments et tâches.
— Supprimer un élément sélectionné.
— Rechercher et filtrer un ou plusieurs éléments spécifiques dans la liste.
— Enregistrer les données du modèle dans un fichier (JSON).
— Lire ou ajouter des données dans un modèle depuis un fichier existant.

1.2. Modèle de données

1.2.1. Graphe de classes

Le modèle de données repose sur une architecture orientée objets, hiérarchique et conforme au paradigme MVC de Qt. Il est basé sur le patron de conception *Composite*, permettant de représenter des tâches simples ainsi que des tâches composées de sous-tâches. Le modèle est encapsulé dans une classe dérivée de `QAbstractItemModel`, afin de permettre son affichage dans une vue hiérarchique de type `QTreeView`.

Le diagramme de classes met en évidence les principales entités du modèle :
— **TodoModel** : modèle Qt responsable de la gestion globale des tâches.
— **TodoItem** : représente une tâche élémentaire.
— **Composite** : spécialisation de `TodoItem` permettant de contenir d'autres tâches.
— **TodoState** : énumération représentant l'état d'une tâche.
— **TodoSerializer** : assure l'importation et l'exportation des données.

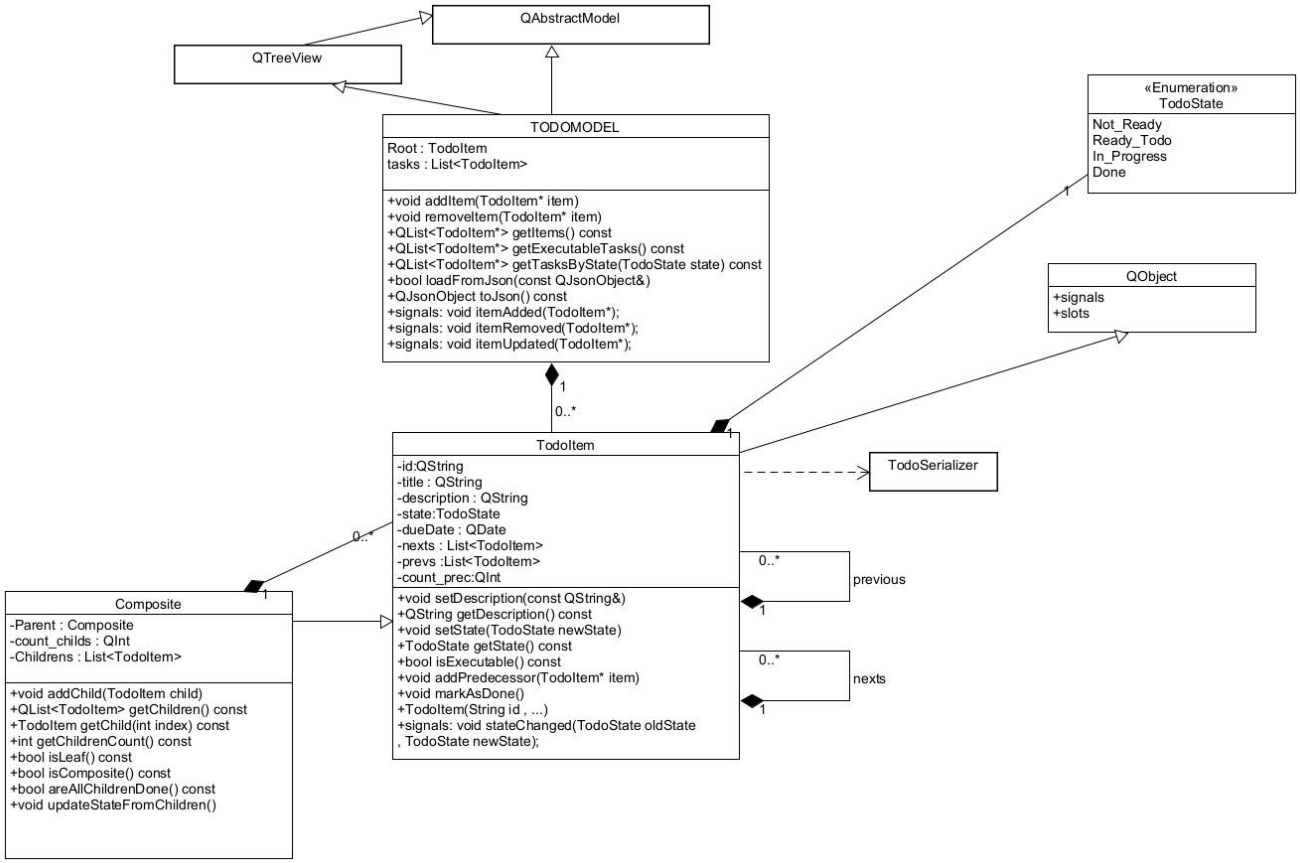

FIGURE 1.1 – Diagramme de classes du modèle de données de l'application KIRA

1.2.2. Description des classes

— **TodoItem** : Cette classe représente une tâche élémentaire du projet. Une tâche est caractérisée par :
  — un identifiant unique (`id`),
  — un titre et une description,
  — une date limite (`dueDate`),
  — un état (`TodoState`),
  — des relations de dépendance avec d'autres tâches (tâches précédentes et suivantes).

  La classe fournit des méthodes permettant de modifier son état, d'ajouter des dépendances, de vérifier si la tâche est exécutable et d'émettre des signaux lors des changements d'état afin de synchroniser le modèle avec l'interface graphique.

— **Composite** : La classe `Composite` hérite de `TodoItem` et implémente le patron de conception Composite. Elle permet de représenter une tâche complexe composée de plusieurs sous-tâches. Elle contient :
  — une référence vers une tâche parente,
  — une liste de sous-tâches (`children`).

  Cette classe fournit des méthodes pour ajouter ou supprimer des sous-tâches, déterminer si une tâche est feuille ou composite, et calculer l'état global à partir de l'état de ses sous-tâches (par exemple, vérifier si toutes les sous-tâches sont terminées).

— **TodoModel** : Cette classe hérite de `QAbstractItemModel` et constitue le cœur du

modèle Qt. Elle est responsable :
— de la gestion de la structure hiérarchique des tâches,
— de la création et suppression des éléments du modèle,
— de la communication avec les vues Qt (`QTreeView`),
— de l'émission des signaux nécessaires à la mise à jour automatique de l'interface.
Elle fournit également des fonctionnalités avancées telles que le filtrage des tâches par état et la récupération des tâches exécutables.

— **TodoState** : Il s'agit d'une énumération définissant les différents états possibles d'une tâche : `Not_Ready`, `Ready_Todo`, `In_Progress` et `Done`. Cette énumération permet de standardiser la gestion du cycle de vie des tâches.

— **TodoSerializer** : Cette classe est chargée de la sérialisation et désérialisation du modèle de données. Elle permet l'importation et l'exportation des tâches au format JSON, assurant ainsi la persistance des données et l'échange de projets.

2. Fonctionnalités et Utilisation

2.1. Fonctionnalités de l'application

L'application **KIRA** est un outil de gestion de tâches hiérarchiques permettant d'organiser, de planifier et de suivre l'avancement d'un projet. Elle repose sur une architecture MVC fournie par Qt et propose un ensemble de fonctionnalités couvrant l'ensemble du cycle de vie des tâches, de leur création jusqu'à leur achèvement.

2.1.1. Gestion des tâches

La gestion des tâches constitue la fonctionnalité centrale de l'application. Les tâches sont organisées sous forme d'une structure arborescente, permettant de représenter des projets complexes composés de plusieurs niveaux de sous-tâches.

Cette fonctionnalité permet notamment :

— **Affichage hiérarchique des tâches** : Les tâches et sous-tâches sont affichées dans une vue de type `QTreeView`, offrant une visualisation claire de la structure du projet et des relations parent–enfant entre les éléments.

— **Création de tâches et de sous-tâches** : L'utilisateur peut créer de nouvelles tâches racines ou ajouter des sous-tâches à une tâche existante. Chaque tâche est intégrée dynamiquement au modèle de données et apparaît immédiatement dans l'interface grâce au mécanisme de signaux et slots de Qt.

— **Édition des propriétés d'une tâche** : Les propriétés d'une tâche (titre, description, date limite et état) peuvent être modifiées directement depuis l'interface graphique. L'édition s'effectue sans boîte de dialogue, via des composants intégrés, conformément aux recommandations de Qt, garantissant une interaction fluide et intuitive.

— **Suppression des tâches** : L'application permet la suppression d'une tâches sélectionnée. Lorsqu'une tâche composite est supprimée, l'ensemble de ses sous-tâches est également supprimé, conformément au patron de conception Composite utilisé dans le modèle.

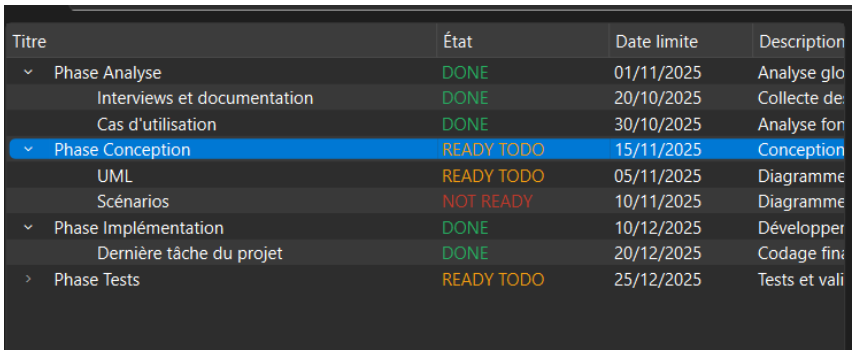

FIGURE 2.1 – tache

2.1.2. Gestion des dépendances

En plus de la hiérarchie des tâches, l'application prend en charge des relations de dépendance entre celles-ci, permettant de modéliser un ordre logique d'exécution.

Cette fonctionnalité inclut :

— **Définition des dépendances entre tâches** : Une tâche peut dépendre d'une ou plusieurs autres tâches (relations de type précédente / suivante). Ces dépendances sont gérées directement dans le modèle de données et reflètent les contraintes d'exécution du projet.

— **Prévention des cycles de dépendances** : Afin de garantir la cohérence du modèle et d'éviter toute situation de blocage, l'application interdit la création de cycles de dépendances entre les tâches. Lors de l'ajout ou de la modification d'une dépendance, un contrôle automatique est effectué pour vérifier que la relation ne génère pas de cycle dans le graphe des tâches. Ainsi, le graphe des dépendances reste acyclique, assurant un ordre d'exécution valide et déterministe.

— **Blocage des tâches non exécutables** : Une tâche ne peut être considérée comme exécutable que lorsque toutes ses tâches prérequises sont terminées. L'application vérifie automatiquement ces conditions et adapte l'état de la tâche en conséquence.

— **Calcul automatique de l'avancement des tâches composites** : L'état d'une tâche composite est déterminé à partir de l'état de ses sous-tâches. Par exemple, une tâche composite est considérée comme terminée lorsque toutes ses sous-tâches le sont. Ce calcul est effectué automatiquement par le modèle, garantissant la cohérence des données et une mise à jour dynamique de l'interface graphique.

2.1.3. Recherche et filtrage

L'application intègre des fonctionnalités de recherche et de filtrage afin de faciliter l'accès rapide aux tâches et d'améliorer l'expérience utilisateur, en particulier dans le cas de projets contenant un grand nombre d'éléments.

Cette fonctionnalité inclut :

— **Barre de recherche par préfixe** : Une barre de recherche permet à l'utilisateur de saisir un texte afin de filtrer les éléments affichés. Seules les tâches dont le nom commence par le préfixe saisi par l'utilisateur sont conservées dans l'affichage, les autres étant temporairement masquées. Ce mécanisme de filtrage par préfixe permet une recherche rapide et efficace, tout en restant simple à utiliser.
Remarque : si la tâche correspondant au préfixe recherché est un composite, l'affichage conserve non seulement le composite lui-même, mais également tous ses

enfants. Cela permet à l'utilisateur de visualiser immédiatement la hiérarchie complète des sous-tâches associées à un composite trouvé, améliorant ainsi la compréhension et la navigation dans les projets complexes.

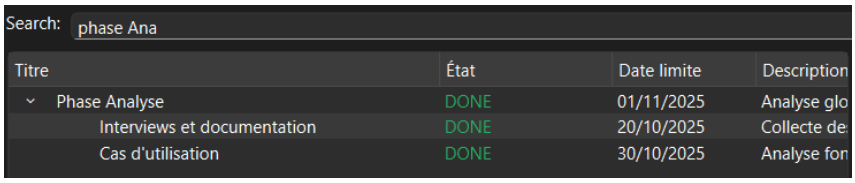

FIGURE 2.2 – exemple de recherche

— **Mise à jour dynamique des résultats** : Le filtrage est appliqué en temps réel lors de la saisie de l'utilisateur. L'interface graphique est automatiquement mise à jour pour refléter les résultats correspondants, sans nécessiter de rechargement manuel.

2.1.4. Import et export

L'application offre des fonctionnalités d'importation et d'exportation permettant à l'utilisateur de sauvegarder, partager ou réutiliser des projets existants.

Cette fonctionnalité inclut :

— **Exportation des projets** : L'utilisateur peut exporter un projet existant sous forme de fichier au format `JSON`. Ce format structuré permet de conserver l'ensemble des informations du projet, notamment les tâches, leur hiérarchie, leurs dépendances ainsi que leurs états.

— **Importation des projets** : L'application permet également l'importation de projets existants à partir de fichiers au format `JSON`. Lors de l'importation, les données sont automatiquement analysées et intégrées dans le modèle de l'application, garantissant la restitution fidèle de la structure et des contraintes du projet initial.

2.1.5. Détails d'une tâche

Dans l'application, une tâche est représentée par la classe `TodoItem`, qui constitue l'élément fondamental du modèle. Chaque tâche regroupe des informations descriptives, temporelles et structurelles permettant de définir son comportement au sein du projet.

Les principales caractéristiques d'une tâche sont les suivantes :

— **Identifiant, titre et description** : Chaque tâche possède un identifiant unique, un titre ainsi qu'une description textuelle permettant de préciser son rôle et son contenu.

— **Date d'échéance** : Une date limite est associée à chaque tâche afin d'encadrer son exécution dans le temps.

— **État de la tâche** : L'état d'une tâche est représenté par un type `TodoState`. Toute modification de cet état déclenche des signaux permettant la mise à jour automatique de l'interface graphique.

— **Relations de dépendance (précédentes / suivantes)** : Une tâche peut être liée à d'autres tâches par des relations de dépendance, stockées sous forme de deux listes :

  — les tâches **précédentes** (`prevs`), qui doivent être terminées avant que la tâche courante ne puisse être exécutée ;

— les tâches **suivantes** (`nexts`), qui dépendent de l'exécution de la tâche courante. Ces relations permettent de modéliser un graphe orienté des tâches et de représenter les contraintes d'ordre d'exécution du projet.

— **Vérification de l'exécutabilité** : Grâce aux dépendances définies, l'application peut déterminer automatiquement si une tâche est prête à être exécutée. Une tâche est considérée comme exécutable uniquement lorsque toutes ses tâches précédentes sont terminées.

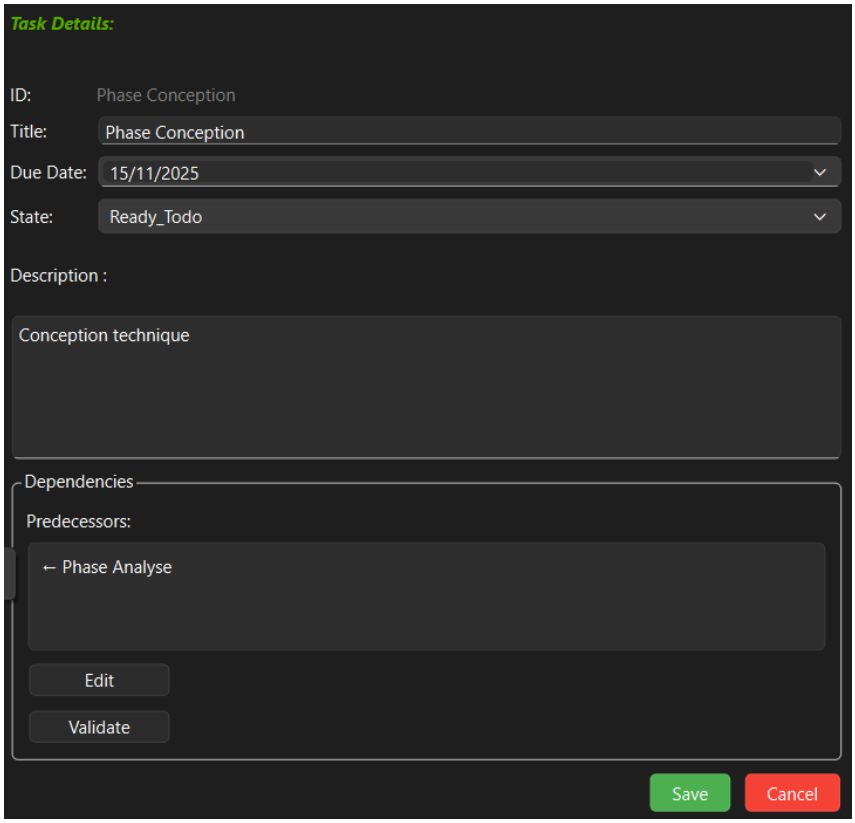

FIGURE 2.3 – détails Tache

2.2. Fichier de test

Afin de tester et valider le bon fonctionnement de l'application **KIRA**, un fichier de test au format `JSON` a été conçu. Ce fichier permet de vérifier l'importation et l'exportation des données, la reconstruction correcte de la hiérarchie des tâches ainsi que la gestion des dépendances entre celles-ci.

2.2.1. Format et organisation générale

Le fichier de test est structuré sous forme d'un objet `JSON` contenant deux éléments principaux :

— **Name** : représente le nom du projet.
— **root** : correspond à la racine du modèle de données.

La racine (`root`) est une tâche composite invisible utilisée uniquement pour structurer l'arbre des tâches. Elle contient une liste de tâches enfants définies dans le champ `children`.

2.2.2. Description d'une tâche

Chaque élément du tableau `children` représente une tâche, simple ou composite, décrite par les attributs suivants :

— **title** : titre de la tâche.
— **description** : description textuelle de la tâche.
— **dueDate** : date limite d'exécution de la tâche.
— **state** : état courant de la tâche (`NOT READY`, `READY TODO`, etc.).
— **isComposite** : indique si la tâche est composite ou simple.
— **children** : liste des sous-tâches (présente uniquement pour les tâches composites).
— **nexts** : liste des titres des tâches suivantes dépendant de la tâche courante.

Cette structure permet de représenter aussi bien des tâches élémentaires que des tâches composites imbriquées sur plusieurs niveaux.

2.2.3. Gestion de la hiérarchie

La hiérarchie des tâches est définie de manière récursive à travers le champ `children`. Une tâche composite peut contenir plusieurs sous-tâches, elles-mêmes composites ou simples, ce qui permet de représenter des projets complexes avec plusieurs niveaux de profondeur.

Dans le fichier de test fourni, certaines tâches composites contiennent un nombre important de sous-tâches, ce qui permet de tester :

— la navigation dans l'arbre des tâches,
— l'affichage hiérarchique dans la vue,
— le calcul automatique de l'état des tâches composites.

2.2.4. Gestion des dépendances

Les dépendances entre tâches sont représentées à l'aide du champ `nexts`. Ce champ contient une liste des titres des tâches qui dépendent de la tâche courante.

Le champ `count_prec` indique le nombre de dépendances entrantes d'une tâche, permettant à l'application de déterminer automatiquement si une tâche est exécutable ou non.

Cette représentation permet de tester :
— la reconstruction correcte du graphe de dépendances,
— la détection des tâches exécutables,
— le respect des contraintes d'ordre d'exécution.

2.2.5. Objectif du fichier de test

Ce fichier de test constitue un cas d'étude complet permettant de valider :
— l'importation correcte des projets,
— la cohérence du modèle de données,
— la gestion simultanée de la hiérarchie et des dépendances,
— la persistance des données via l'exportation au format `JSON`.

Il garantit ainsi la robustesse et la fiabilité du mécanisme de sérialisation implémenté dans l'application **KIRA**.

2.3. Utilisation de l'application

2.3.1. Navigation dans le menu

L'application contient un menu principal **Menu** qui regroupe les actions suivantes :
— **New Project** : permet de créer un nouveau projet avec une structure vide.
— **Open** : ouvre un projet existant à partir d'un fichier JSON.
— **Save** : enregistre les modifications du projet courant dans le fichier associé.
— **Save As** : permet d'enregistrer le projet sous un nouveau nom ou emplacement.
— **Import** : importe des données d'un fichier JSON externe et les ajoute au projet courant.
— **Export** : exporte l'ensemble du projet et sa hiérarchie sous forme de fichier JSON.
— **Quit** : ferme l'application après avoir demandé la sauvegarde des modifications non enregistrées.

Un autre menu intitulé **Edit** est disponible et contient les options suivantes :
— **Create Task** : ajoute une nouvelle tâche simple au niveau sélectionné dans l'arborescence.
— **Create SubTask** : ajoute une sous-tâche à la tâche ou au composite actuellement sélectionné.
— **Add SubGroupTask** : permet de regrouper plusieurs tâches existantes dans un nouveau composite.
— **Delete Task** : supprime la tâche ou le composite sélectionné, avec gestion des relations entre les tâches.
— **Create Group Task** : crée un nouveau composite vide auquel l'utilisateur pourra ajouter des sous-tâches.

L'application dispose également d'un menu **About** qui contient l'option suivante :
— **Info** : affiche des informations sur l'application, telles que la version et l'auteur.

En plus des menus, une **barre d'outils** est présente et contient les boutons correspondant à la plupart de ces actions, afin de faciliter l'accès rapide aux fonctionnalités les plus utilisées. Cela permet à l'utilisateur d'exécuter les actions sans passer par le menu déroulant.

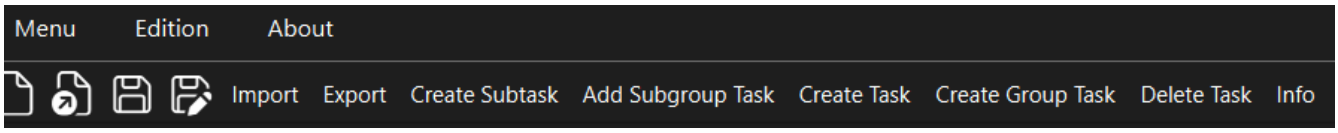

FIGURE 2.4 – Menu de navigation et barre d'outils

2.3.2. Interaction avec les boutons et widgets

L'interface permet à l'utilisateur de renseigner les informations d'une tâche ou d'une tâche composite. Il peut notamment définir :

— un **Title**,
— une **Due Date**,
— un **State**,
— une **Description**.

L'utilisateur peut également modifier la liste des **predecesseurs** d'une tâche ou d'une composite de tâche à travers les widgets disponibles dans l'interface.

3. Conclusion

Ce projet a permis de concevoir et de développer **KIRA**, une application de gestion de tâches hiérarchiques reposant sur une architecture orientée objets et le framework Qt. L'objectif principal était de proposer un outil capable de modéliser des projets complexes en combinant une structure arborescente de tâches avec des relations de dépendance explicites entre celles-ci.

À travers l'utilisation du patron de conception **Composite** et l'intégration du modèle dans un `QAbstractItemModel`, l'application assure une séparation claire entre les données, la logique métier et l'interface graphique, conformément au paradigme MVC. Cette approche garantit la cohérence du modèle, la réactivité de l'interface et la facilité d'extension du système.

Les fonctionnalités implémentées couvrent l'ensemble du cycle de vie des tâches : création, édition, suppression, gestion des dépendances, calcul automatique des états, recherche, filtrage ainsi que l'importation et l'exportation des projets au format `JSON`. L'interface utilisateur, basée sur des menus, une barre d'outils et des widgets dédiés, permet une interaction fluide et intuitive avec le modèle de données.

En conclusion, **KIRA** constitue une solution fonctionnelle et extensible pour la gestion de projets structurés. Des améliorations futures pourraient inclure l'ajout d'une visualisation graphique de l'avancement (diagrammes de Gantt), la gestion collaborative multi-utilisateurs ou encore l'intégration de notifications automatiques, afin d'enrichir davantage l'expérience utilisateur et les capacités de l'application.