Rapport de projet

Application de gestion de tâches

ToDoApp

Cours : Langages et Applications Orientés Objets Avancés (LAOA)

Auteurs :

Simon Bélier
Wissal Jalfa

17 décembre 2025

Table des matières

1 Introduction **3**

2 Architecture générale **3**

3 Classes et diagramme de classes **5**
3.1 Inventaire des classes principales 5
3.2 Diagramme de classes détaillé 6
3.3 Relations entre les classes 6

4 Modèle de données **8**
4.1 La classe Task 8
4.2 Extension Git : GitIssueTask 8
4.3 Le modèle Qt : TaskModel 9
4.3.1 Choix de QAbstractItemModel 9
4.3.2 Structure interne 9
4.3.3 Colonnes affichées 9
4.4 Filtrage : TaskFilterProxyModel 9
4.5 Signaux et slots pour la synchronisation 10

5 Vues graphiques et widgets **10**
5.1 Organisation de l'interface 10
5.2 Fenêtre principale : MainWindow 10
5.3 Vue arborescente (QTreeView) 11
5.4 Panneau de détails : TaskDetailWidget 11
5.5 Vue Kanban 12

6 Contrôleurs et logique d'interaction **13**
6.1 MainWindow comme contrôleur central 13
6.2 Délégué d'édition : TaskItemDelegate 13
6.3 Système Undo/Redo 13

7 Fonctionnalités de l'application **14**
7.1 Création, modification, suppression 14
7.2 Filtrage et recherche 14
7.3 Statistiques et visualisations 14
7.3.1 Vue de distribution des tâches 15
7.3.2 Vue Heatmap 15
7.3.3 Vue Calendrier 16
7.3.4 Vue d'avancement 16
7.3.5 Timer Pomodoro 17
7.4 Persistance et export 17
7.5 Thèmes visuels 17
7.5.1 Thème clair (Light) 18

7.5.2 Thème sombre (Dark) 18
7.5.3 Thème chaleureux (Warm) 19
7.6 Intégration Git 20
7.7 Internationalisation et traductions 21

8 Justification des choix techniques **22**
8.1 QAbstractItemModel pour la hiérarchie 22
8.2 Séparation MVC stricte 22
8.3 Édition in-place 22
8.4 Proxy model pour le filtrage 22
8.5 Intégration Git 23

9 Conclusion **23**

1 Introduction

Ce rapport présente le projet ToDoApp, une application de gestion de tâches hiérarchiques développée en C++ avec le framework Qt 6. L'application implémente l'architecture MVC (Modèle-Vue-Contrôleur) et utilise les mécanismes de Qt pour synchroniser automatiquement les données et l'interface graphique.

Le projet démontre l'utilisation des concepts de Qt : modèles personnalisés dérivés de `QAbstractItemModel`, vues arborescentes (`QTreeView`), modèles proxy de filtrage (`QSortFilterProxyModel`), délégués d'édition (`QStyledItemDelegate`), et système de signaux/slots pour la communication inter-composants. L'application intègre des fonctionnalités avancées : système d'annulation/rétablissement (Undo/Redo) avec `QUndoStack`, intégration avec les plateformes Git (GitHub, GitLab, Gitea), vues métier (Kanban, Timeline, Burndown), et widgets de visualisation des statistiques.

L'architecture suit la séparation stricte des responsabilités : les classes de modèle (`Task`, `Category`, `TaskModel`) gèrent la logique métier et la persistance des données, les vues (`MainWindow`, `TaskDetailWidget`, `KanbanView`) gèrent l'affichage, et les contrôleurs coordonnent les interactions utilisateur.

2 Architecture générale

L'architecture de ToDoApp suit le patron MVC avec une séparation entre le modèle de données, les vues graphiques et la logique de contrôle. La figure 1 présente l'organisation des composants principaux et leurs interactions.

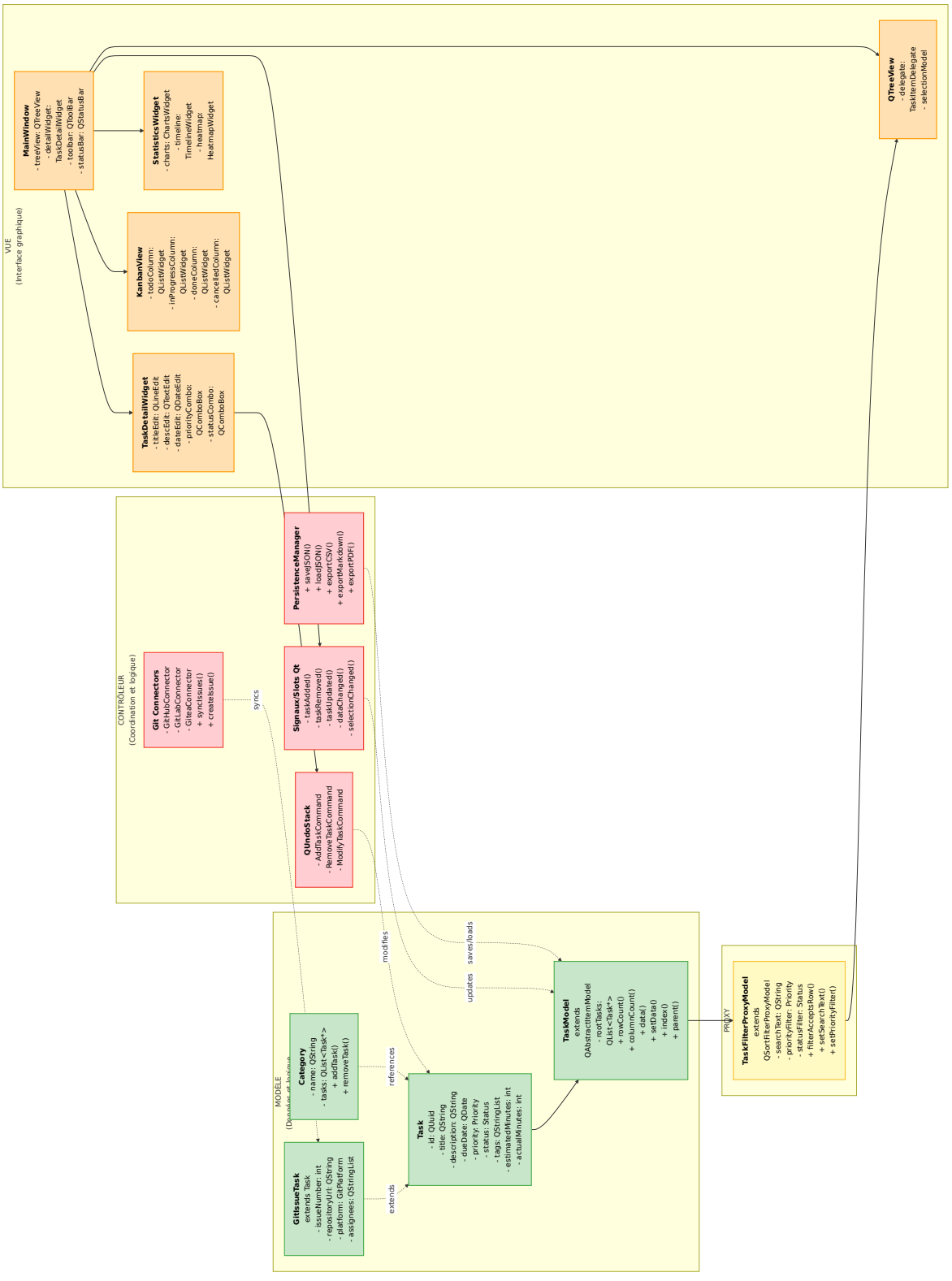

FIGURE 1 – Architecture MVC de l'application ToDoApp

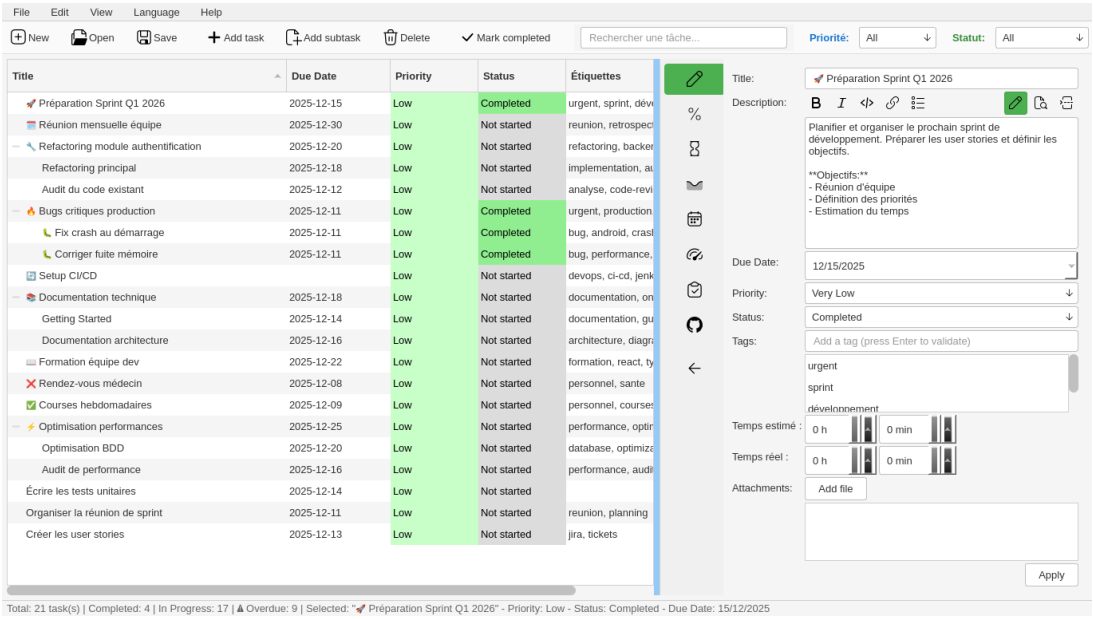

FIGURE 2 – Vue d'ensemble de l'interface principale avec tâches

Le modèle (`TaskModel`) hérite de `QAbstractItemModel` pour gérer la structure arborescente des tâches. Un proxy de filtrage (`TaskFilterProxyModel`) s'intercale entre le modèle et la vue (`QTreeView`) pour implémenter la recherche et le filtrage sans altérer les données sources. La fenêtre principale (`MainWindow`) orchestre les interactions et contient tous les widgets de visualisation.

3 Classes et diagramme de classes

3.1 Inventaire des classes principales

Le projet ToDoApp est structuré en quatre catégories de classes :

Classes du modèle métier (répertoire `src/models/`)

- `Task` : représentation d'une tâche avec hiérarchie parent-enfant
- `GitIssueTask` : extension de `Task` pour synchronisation Git
- `Category` : regroupement thématique de tâches
- `TaskModel` : modèle Qt arborescent (`QAbstractItemModel`)
- `TaskFilterProxyModel` : proxy de filtrage et recherche
- `TaskItemDelegate` : délégué pour édition in-place

Classes de vue (répertoire `src/widgets/`)

- `MainWindow` : fenêtre principale et contrôleur central
- `TaskDetailWidget` : panneau d'édition exhaustive d'une tâche
- `KanbanView` : vue Kanban par colonnes de statut
- `StatisticsWidget` : affichage des métriques agrégées
- `ChartsWidget` : graphiques de répartition

— `TimelineWidget` : vue chronologique
— `BurndownWidget` : graphique de progression
— `HeatmapWidget` : carte de chaleur d'activité
— `PomodoroTimer` : timer Pomodoro

Classes utilitaires (répertoire `src/utils/`)

— `PersistenceManager` : sauvegarde/chargement JSON, CSV, Markdown
— `AddTaskCommand`, `RemoveTaskCommand`, `ModifyTaskCommand` : commandes Undo/Redo
— `ThemesManager` : gestion des thèmes visuels

Classes Git (répertoire `src/git/`)

— `RepositoryManager` : gestionnaire de dépôts Git
— `GitRepository` : représentation d'un dépôt distant
— `GitHubConnector`, `GitLabConnector` : connecteurs API REST
— `GitProjectWidget` : interface de gestion des dépôts

3.2 Diagramme de classes détaillé

La figure 3 présente le diagramme de classes complet mettant en évidence les relations d'héritage, de composition, et d'association entre les classes principales.

3.3 Relations entre les classes

Héritage. Toutes les classes métier (`Task`, `Category`) héritent de `QObject` pour bénéficier du système de signaux/slots. `GitIssueTask` étend `Task` pour ajouter les métadonnées Git. `TaskModel` hérite de `QAbstractItemModel` pour implémenter un modèle arborescent compatible avec les vues Qt.

Composition. La classe `Task` implémente une relation de composition récursive : chaque tâche possède une liste de sous-tâches (`m_subtasks`), permettant de construire des arbres de profondeur arbitraire. Le `TaskModel` possède également les tâches racines (`m_rootTasks`) et est responsable de leur cycle de vie.

Agrégation. La classe `Category` maintient une liste de références vers des tâches (`m_tasks`) sans en avoir la propriété. La destruction d'une catégorie n'entraîne pas la destruction des tâches associées.

Association. `TaskFilterProxyModel` observe `TaskModel` via les signaux Qt standard (`dataChanged`, `rowsInserted`). `MainWindow` orchestre l'ensemble des widgets et leur fournit une référence au modèle. Les widgets de visualisation (`KanbanView`, `ChartsWidget`) observent également le modèle pour se rafraîchir automatiquement.

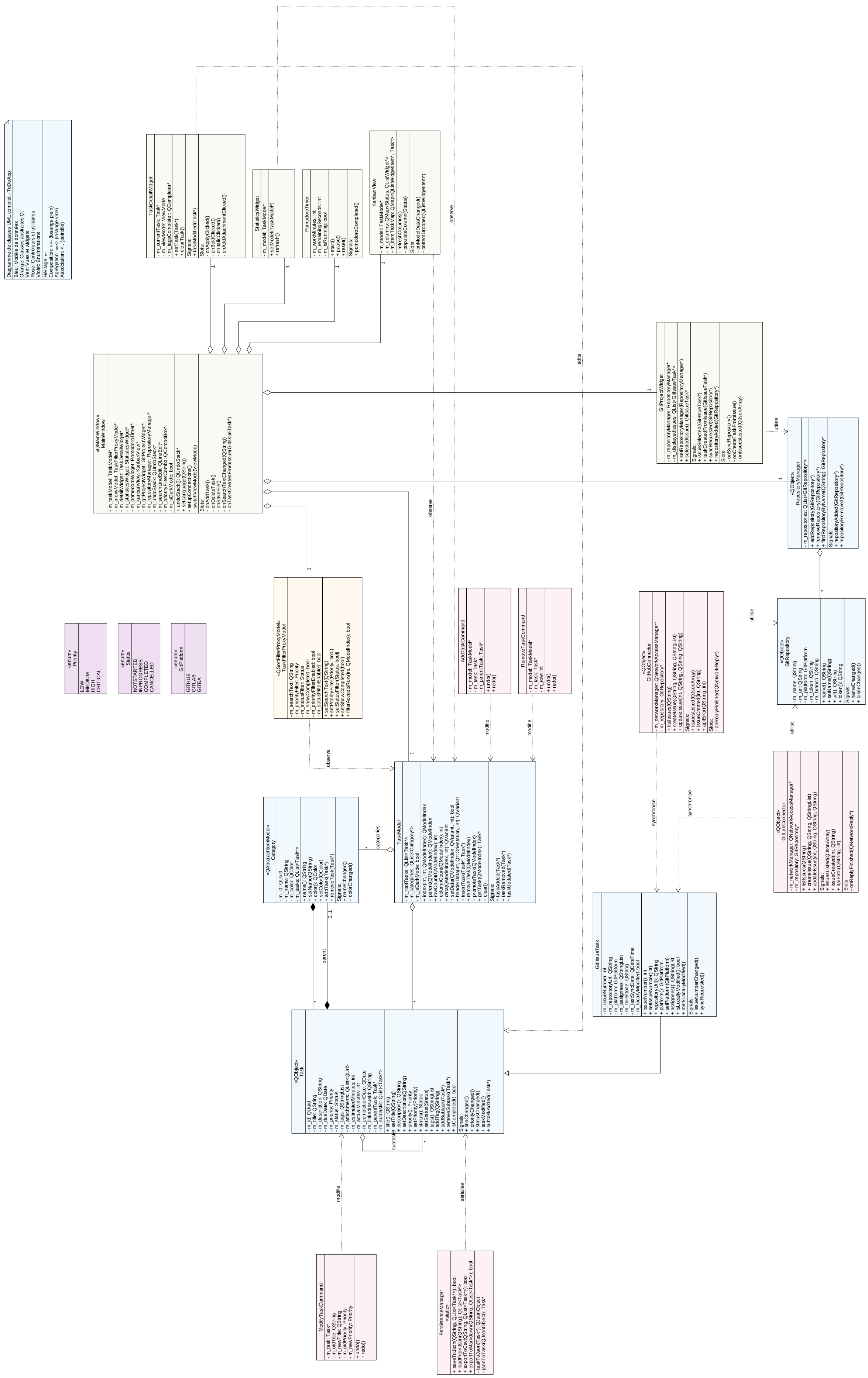

Figure 3 – Diagramme de classes UML de ToDoApp

4 Modèle de données

4.1 La classe Task

La classe `Task`, définie dans `task.h` et `task.cpp`, hérite de `QObject` et représente une tâche individuelle. Elle encapsule les attributs suivants :

```
class Task : public QObject {
    Q_OBJECT
private:
    QUuid m_id;
    QString m_title;
    QString m_description;
    QDate m_dueDate;
    Priority m_priority;          // enum: LOW, MEDIUM, HIGH, CRITICAL
    Status m_status;              // enum: NOTSTARTED, INPROGRESS, COMPLETED,
        CANCELLED
    QStringList m_tags;
    QList<QUrl> m_attachments;
    int m_estimatedMinutes;
    int m_actualMinutes;
    QDate m_completionDate;
    QString m_linkedIssueId;      // format: "owner/repo#123"

    Task *m_parentTask;
    QList<Task*> m_subtasks;      // Composition
};
```

Chaque tâche possède un identifiant unique (`QUuid`) généré automatiquement à la construction. La hiérarchie parent-enfant est implémentée par composition : chaque tâche maintient un pointeur vers son parent (`m_parentTask`) ainsi qu'une liste de pointeurs vers ses sous-tâches (`m_subtasks`). Cette structure permet de représenter des arbres de profondeur arbitraire.

La classe `Task` expose l'ensemble de ses attributs via des propriétés Qt (`Q_PROPERTY`), ce qui permet une liaison automatique avec les widgets de l'interface. Toute modification d'une propriété déclenche l'émission d'un signal dédié (`titleChanged()`, `priorityChanged()`, etc.), assurant ainsi la cohérence entre le modèle et les vues.

4.2 Extension Git : GitIssueTask

La classe `GitIssueTask`, définie dans `src/git/gitissuetask.h`, hérite de `Task` et ajoute les attributs nécessaires à la synchronisation avec les issues Git (GitHub, GitLab, Gitea) :

```
class GitIssueTask : public Task {
private:
    int m_issueNumber;
    QString m_repositoryUrl;
    GitPlatform m_platform;       // enum: GitHub, GitLab, Gitea
    QStringList m_assignees;
    QString m_milestone;
    QDateTime m_lastSyncDate;
};
```

Cette extension permet de synchroniser bidirectionnellement les tâches locales avec les issues distantes. Les modifications apportées à une `GitIssueTask` peuvent être propagées vers la plateforme Git correspondante via les classes `GitHubConnector` et `GitLabConnector`.

4.3 Le modèle Qt : TaskModel

4.3.1 Choix de QAbstractItemModel

La classe `TaskModel`, définie dans `taskmodel.h` et `taskmodel.cpp`, hérite de `QAbstractItemModel`. Ce choix architectural est motivé par la nécessité de représenter une structure hiérarchique complexe tout en restant compatible avec les vues Qt standards (`QTreeView`, `QListView`). Contrairement à `QAbstractListModel` (modèle plat) ou `QAbstractTableModel` (modèle tabulaire), `QAbstractItemModel` permet de modéliser des arbres de profondeur arbitraire en implémentant la méthode `parent()` qui retourne l'index du parent d'un élément donné.

4.3.2 Structure interne

Le `TaskModel` maintient une liste des tâches racines (`m_rootTasks`). Les sous-tâches sont accessibles récursivement via la méthode `Task::subtasks()`. Chaque `QModelIndex` retourné par le modèle utilise son pointeur interne (`internalPointer()`) pour référencer directement l'objet `Task` correspondant. Cette technique évite les recherches coûteuses lors de l'accès aux données.

```
QModelIndex TaskModel::index(int row, int col, const QModelIndex &parent) const
    {
    Task *parentTask = parent.isValid()
        ? static_cast<Task*>(parent.internalPointer())
        : nullptr;

    Task *child = parentTask
        ? parentTask->subtasks().at(row)
        : m_rootTasks.at(row);

    return createIndex(row, col, child);
}
```

4.3.3 Colonnes affichées

Le modèle expose cinq colonnes (`columnCount() == 5`) correspondant aux propriétés essentielles d'une tâche : Titre, Date d'échéance, Priorité, Statut, et Étiquettes. La méthode `data()` retourne les valeurs appropriées selon le rôle Qt demandé (`Qt::DisplayRole`, `Qt::EditRole`, `Qt::BackgroundRole`, `Qt::ForegroundRole`). Les colonnes de priorité et de statut sont colorées dynamiquement selon le thème actif (clair ou sombre), et les dates d'échéance dépassées sont affichées en rouge.

4.4 Filtrage : TaskFilterProxyModel

La classe `TaskFilterProxyModel` hérite de `QSortFilterProxyModel` et s'intercale entre le `TaskModel` et la `QTreeView` pour filtrer dynamiquement les tâches affichées sans modifier les

données sous-jacentes. Le proxy implémente la méthode `filterAcceptsRow()` qui détermine si une tâche doit être affichée selon plusieurs critères : recherche textuelle classique, recherche avancée avec préfixes (`tag:`, `priority:`, `status:`, `date:`), filtre par priorité, filtre par statut, et affichage/masquage des tâches terminées.

4.5 Signaux et slots pour la synchronisation

Le système de signaux et slots de Qt assure la communication asynchrone entre les composants. Les signaux définis dans `Task` notifient les changements d'état (`titleChanged()`, `priorityChanged()`, `statusChanged()`, `taskModified()`). Le `TaskModel` émet également des signaux personnalisés (`taskAdded()`, `taskRemoved()`, `taskUpdated()`). Ces signaux sont connectés dans `MainWindow::setupConnecti` pour déclencher la mise à jour de la barre d'état et des widgets statistiques.

5 Vues graphiques et widgets

5.1 Organisation de l'interface

L'interface graphique de ToDoApp est structurée selon le schéma présenté en figure 4. La fenêtre principale est divisée horizontalement par un `QSplitter`, séparant la vue arborescente (gauche) du panneau de visualisation (droite).

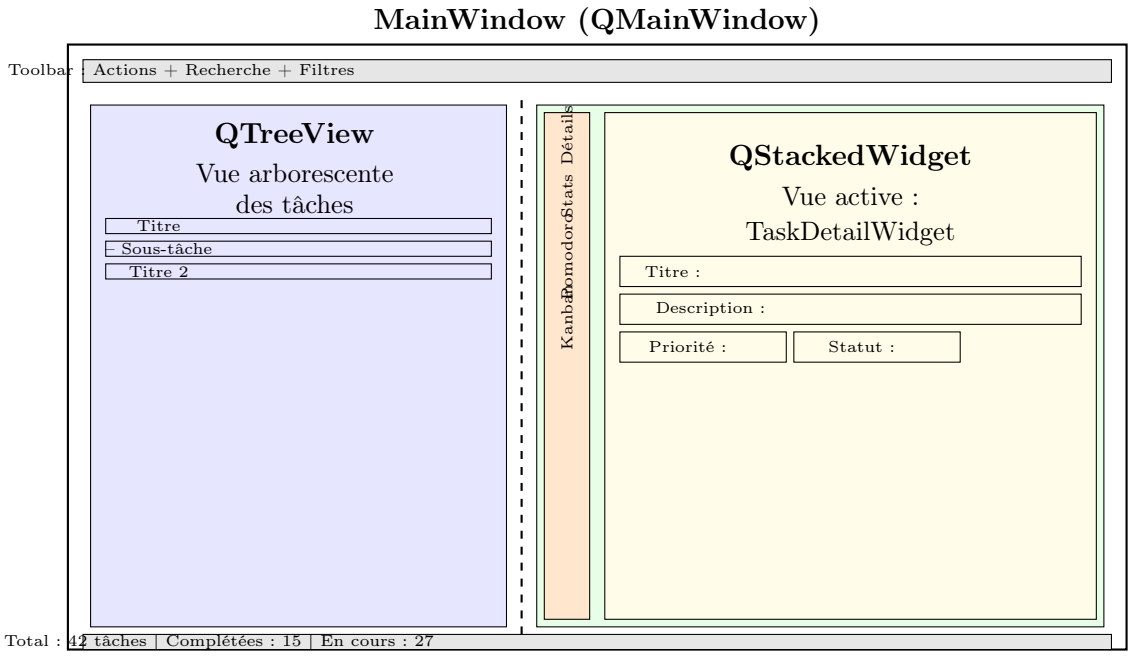

FIGURE 4 – Organisation spatiale de l'interface graphique

5.2 Fenêtre principale : MainWindow

La classe `MainWindow` hérite de `QMainWindow` et orchestre l'ensemble de l'interface. Elle instancie le `TaskModel`, le `TaskFilterProxyModel`, et tous les widgets de visualisation. La toolbar principale contient des widgets intégrés : un `QLineEdit` pour la recherche, deux `QComboBox` pour les filtres de priorité et de statut.

5.3 Vue arborescente (QTreeView)

La `QTreeView` est configurée pour supporter l'édition in-place, la sélection multiple, et le glisser-déposer. Un délégué personnalisé `TaskItemDelegate` fournit des éditeurs contextuels selon la colonne : `QLineEdit` pour le titre, `QDateEdit` avec calendrier pour la date, `QComboBox` pour la priorité et le statut.

5.4 Panneau de détails : TaskDetailWidget

`TaskDetailWidget` est un widget pour l'édition complète d'une tâche sélectionnée. Il contient des champs pour le titre, la description (avec éditeur Markdown et 3 modes de visualisation), la date d'échéance, la priorité, le statut, les temps estimé et réel, les tags (avec autocomplétion), et les pièces jointes. Les modifications sont appliquées uniquement après un clic sur le bouton "Appliquer".

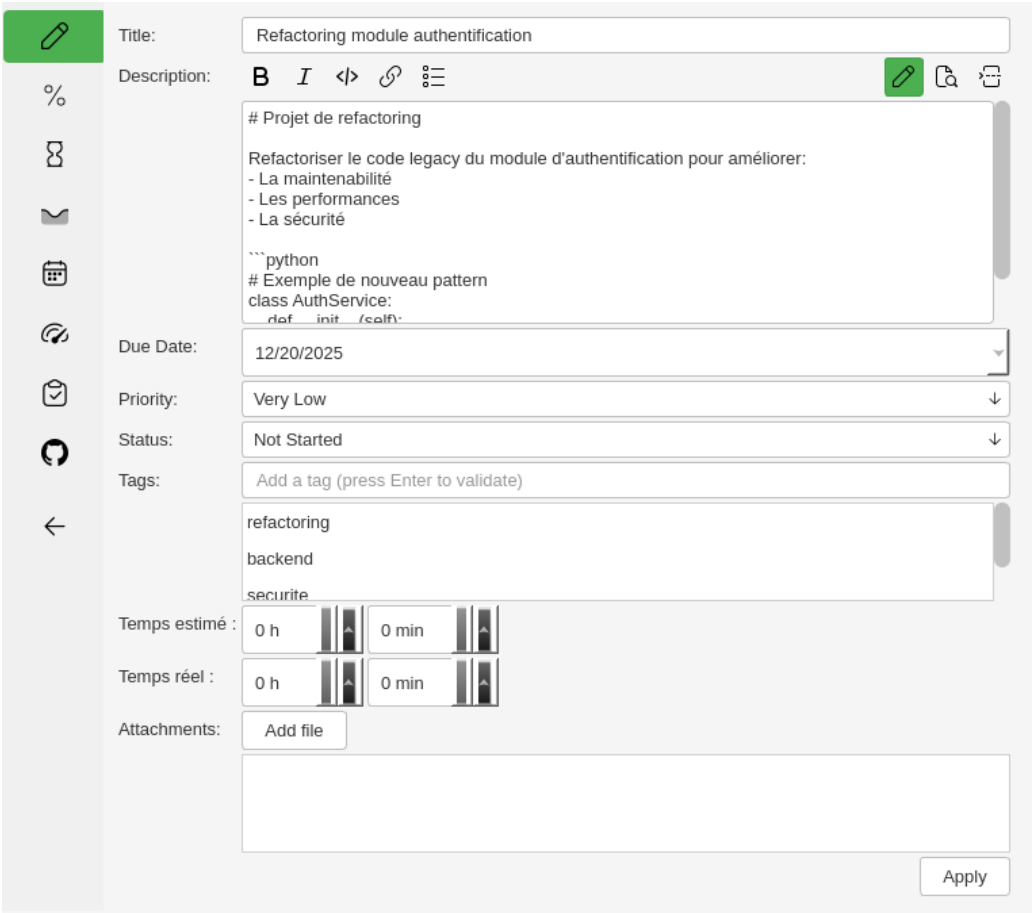

FIGURE 5 – Panneau de détails TaskDetailWidget (vue 1)

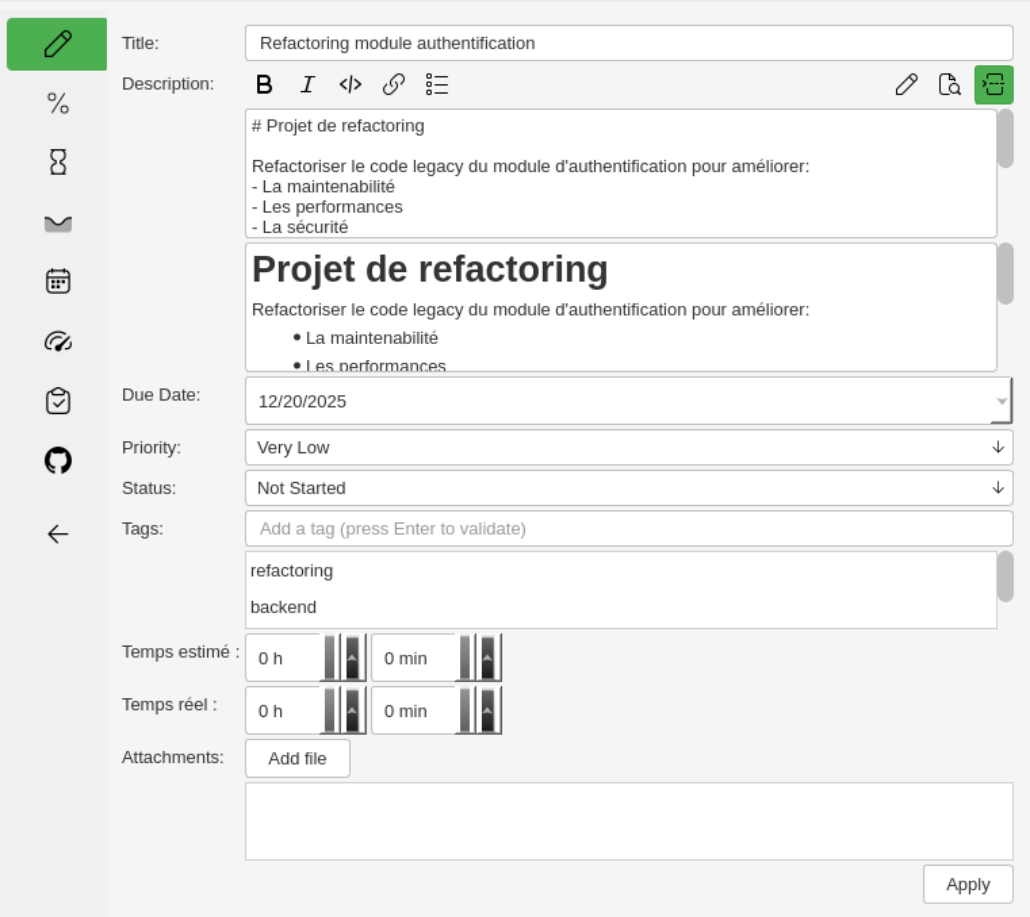

FIGURE 6 – Panneau de détails TaskDetailWidget (vue 2)

5.5 Vue Kanban

La classe `KanbanView` fournit une visualisation alternative avec quatre colonnes organisées par statut (À faire, En cours, Terminé, Annulé). Chaque colonne est un `QListWidget` qui supporte le glisser-déposer pour changer le statut d'une tâche. Le `KanbanView` observe le `TaskModel` et se met à jour automatiquement. Cette vue complète la vue arborescente : la vue arborescente affiche la décomposition hiérarchique des tâches, tandis que la vue Kanban les organise selon leur statut.

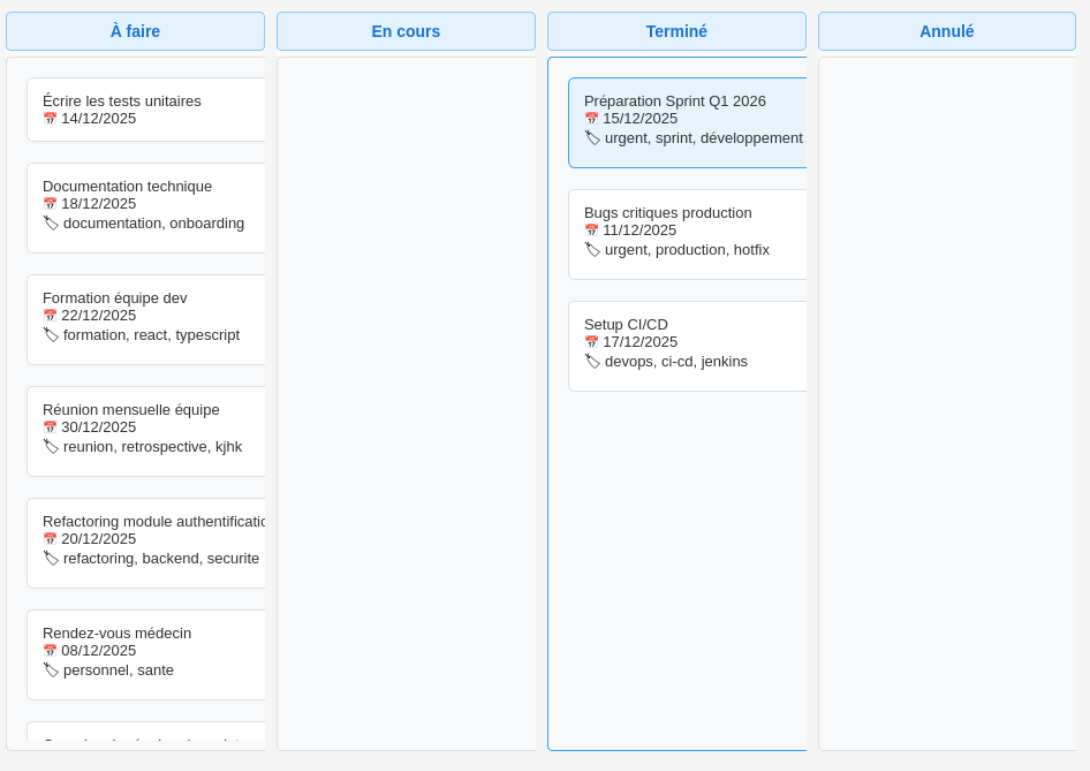

FIGURE 7 – Vue Kanban avec colonnes par statut

6 Contrôleurs et logique d'interaction

6.1 MainWindow comme contrôleur central

La classe `MainWindow` agit comme le contrôleur central de l'application. La méthode `setupConnections()` établit l'ensemble des connexions signaux/slots : actions du menu aux slots correspondants, modèle de sélection de la `QTreeView` à `onTaskSelectionChanged()`, widgets de recherche au proxy model, `TaskDetailWidget` au modèle, et signaux du `TaskModel` à la barre d'état.

6.2 Délégué d'édition : TaskItemDelegate

`TaskItemDelegate` hérite de `QStyledItemDelegate` et personnalise l'édition in-place. Elle implémente `createEditor()` pour instancier le widget approprié, `setEditorData()` pour initialiser le widget avec la valeur actuelle, et `setModelData()` pour transmettre la nouvelle valeur au modèle via `setData()`.

6.3 Système Undo/Redo

L'application implémente un système complet basé sur `QUndoStack`. Trois commandes sont définies : `AddTaskCommand`, `RemoveTaskCommand`, et `ModifyTaskCommand`. Chaque commande hérite de `QUndoCommand` et implémente `undo()` et `redo()`. La pile d'annulation est accessible via Ctrl+Z et Ctrl+Shift+Z.

7 Fonctionnalités de l'application

7.1 Création, modification, suppression

L'utilisateur peut créer des tâches racines (Ctrl+N) ou des sous-tâches (Ctrl+Shift+N). L'édition se fait soit directement dans la `QTreeView` (double-clic), soit dans `TaskDetailWidget` (modifications appliquées via bouton "Appliquer"). La suppression (touche Suppr) est réversible via Undo/Redo, avec confirmation optionnelle.

7.2 Filtrage et recherche

Le `TaskFilterProxyModel` implémente plusieurs mécanismes : recherche textuelle classique, recherche avancée avec préfixes (`tag:urgent`, `priority:high`, `status:completed`, `date:2024-12`), filtres par priorité et statut via `QComboBox`, et affichage/masquage des tâches terminées.

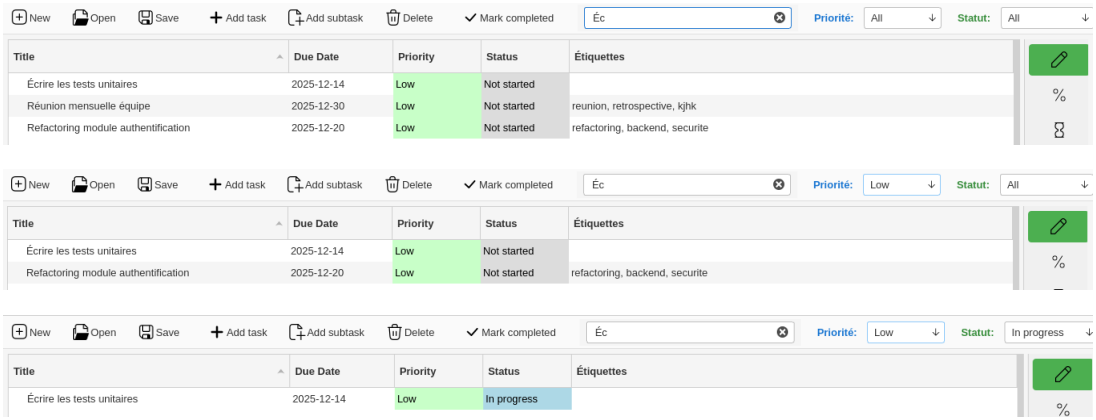

FIGURE 8 – Interface de recherche et filtrage avancé (trois exemples)

7.3 Statistiques et visualisations

L'application intègre un `StatisticsWidget` qui affiche des métriques agrégées en temps réel : nombre total de tâches, taux de complétion, répartition par priorité et par statut, temps estimé vs temps réel, productivité quotidienne. Les widgets de visualisation incluent `ChartsWidget` (graphiques en camembert et barres), `TimelineWidget` (chronologie des tâches), `BurndownWidget` (courbe de progression), et `HeatmapWidget` (carte de chaleur d'activité hebdomadaire).

7.3.1 Vue de distribution des tâches

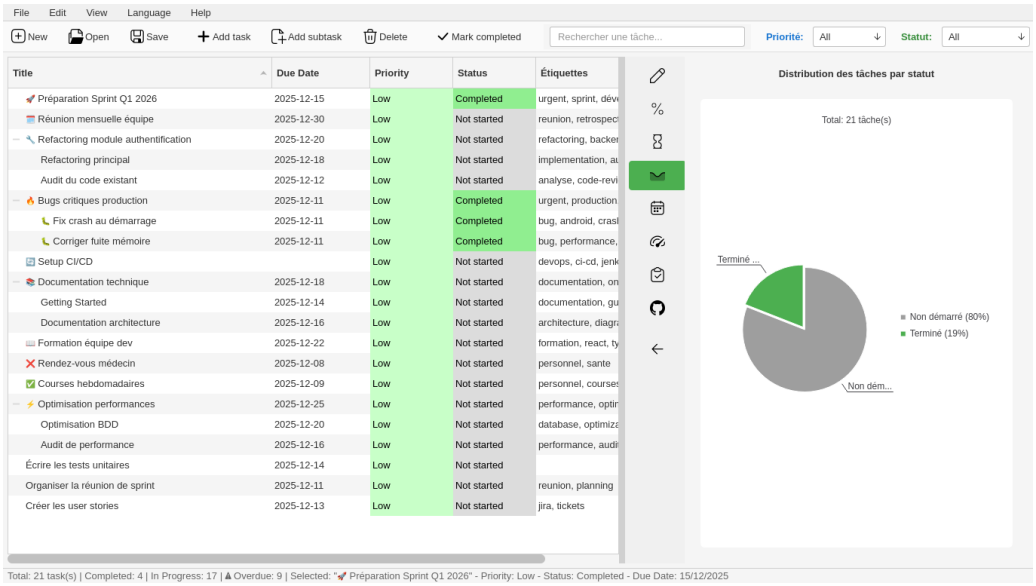

FIGURE 9 – Graphiques de distribution des tâches par priorité et statut

7.3.2 Vue Heatmap

La carte de chaleur (Heatmap) permet de visualiser l'activité hebdomadaire avec une représentation colorée de l'intensité de travail sur chaque jour.

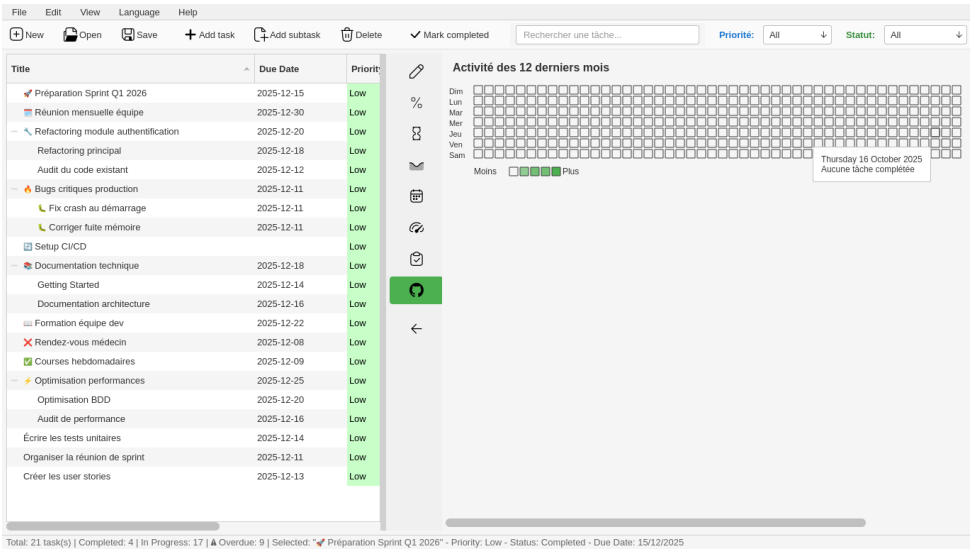

FIGURE 10 – Carte de chaleur d'activité hebdomadaire

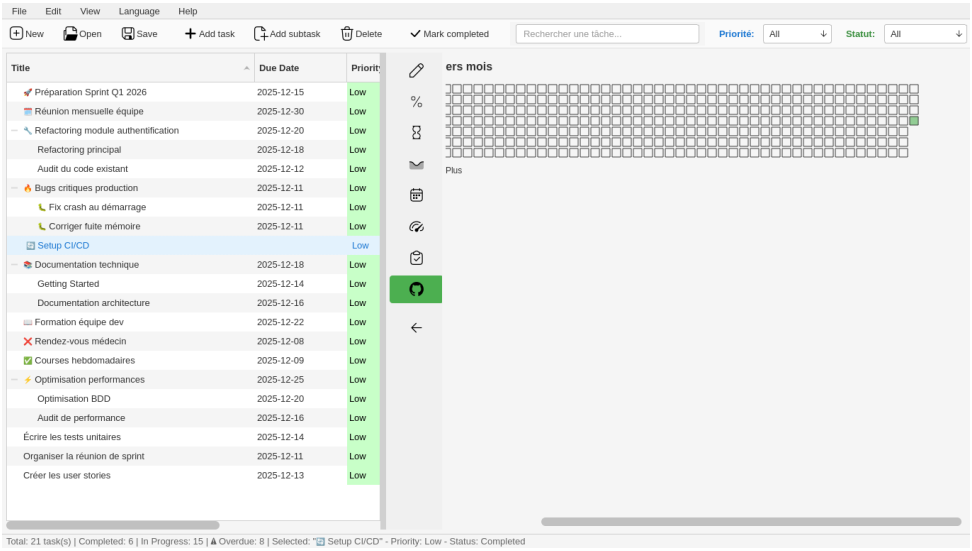

FIGURE 11 – Carte de chaleur d'activité (vue alternative)

7.3.3 Vue Calendrier

Le widget calendrier offre une vue chronologique des tâches avec leurs échéances.

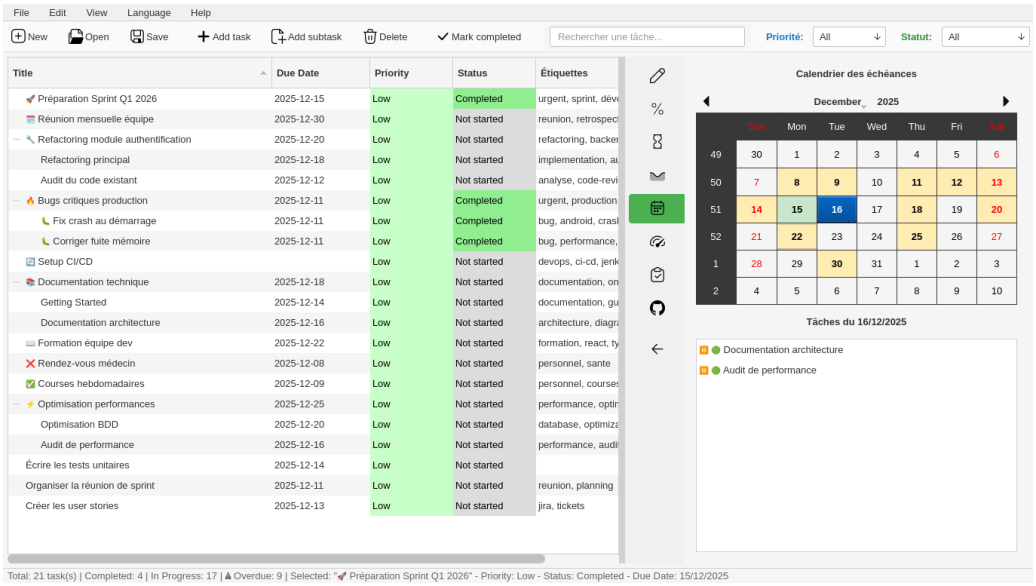

FIGURE 12 – Vue calendrier avec échéances des tâches

7.3.4 Vue d'avancement

La vue d'avancement affiche la progression globale du projet sous forme de graphiques et métriques.

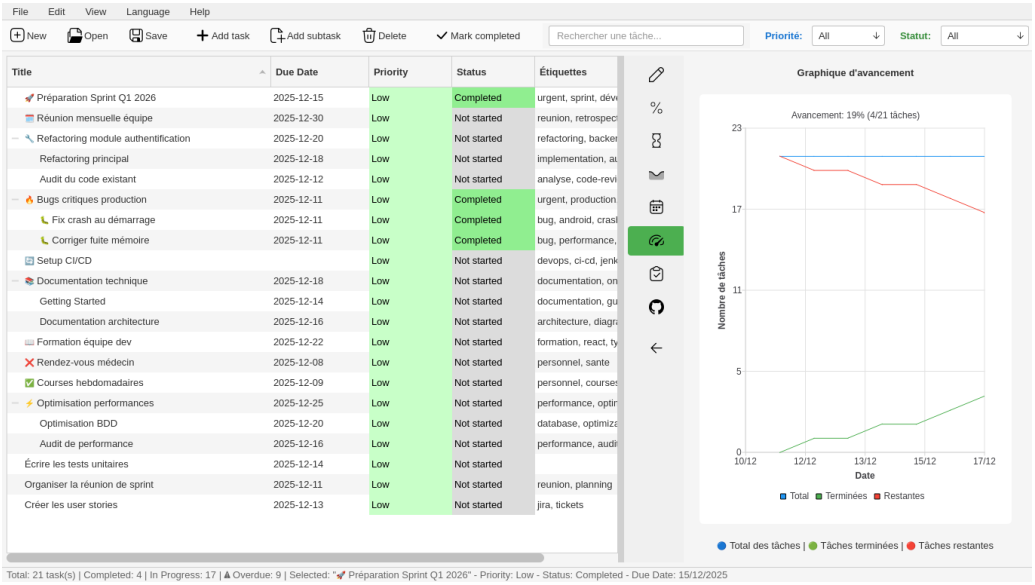

FIGURE 13 – Vue d'avancement du projet

7.3.5 Timer Pomodoro

L'application intègre un timer Pomodoro pour faciliter la gestion du temps de travail selon la technique Pomodoro.

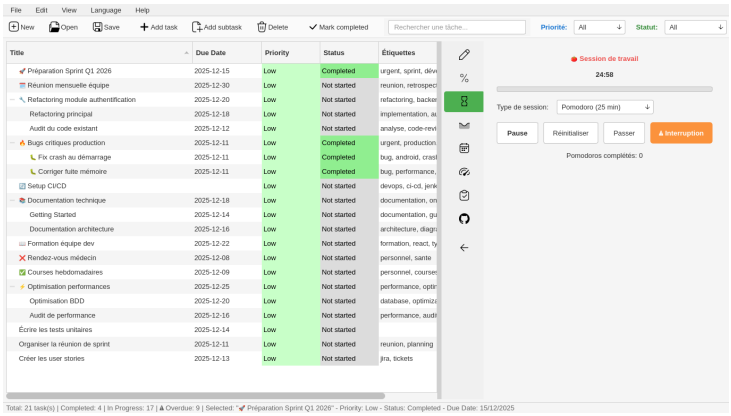

FIGURE 14 – Timer Pomodoro intégré

7.4 Persistance et export

`PersistenceManager` fournit des méthodes statiques pour sauvegarder et charger des tâches en JSON (format natif préservant la hiérarchie), CSV (format tabulaire), Markdown (format texte avec niveaux de titres), et PDF (via template HTML et `QPrinter`). Un système de sauvegarde automatique périodique est paramétrable.

7.5 Thèmes visuels

L'application intègre un système complet de gestion des thèmes visuels via la classe `ThemesManager` (définie dans `src/utils/themesmanager.h`). L'utilisateur peut choisir parmi trois thèmes pré-

définis, chacun optimisé pour un contexte d'utilisation spécifique. Les thèmes sont implémentés via des feuilles de style Qt (QSS) stockées dans le répertoire `resources/themes/`.

7.5.1 Thème clair (Light)

Le thème clair fournit une interface lumineuse adaptée aux environnements bien éclairés. Il utilise des couleurs pastel et des contrastes modérés.

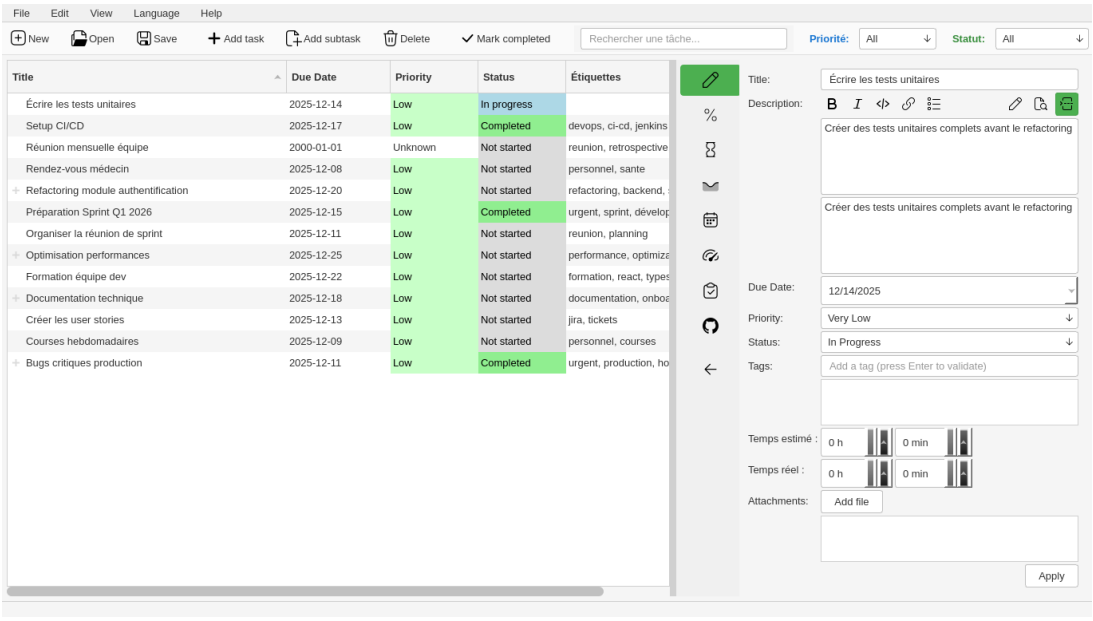

FIGURE 15 – Interface avec thème clair

7.5.2 Thème sombre (Dark)

Le thème sombre fournit une interface à dominante noire et grise, adaptée aux sessions de travail prolongées et aux environnements peu éclairés. Ce thème réduit la luminosité de l'écran tout en préservant la lisibilité.

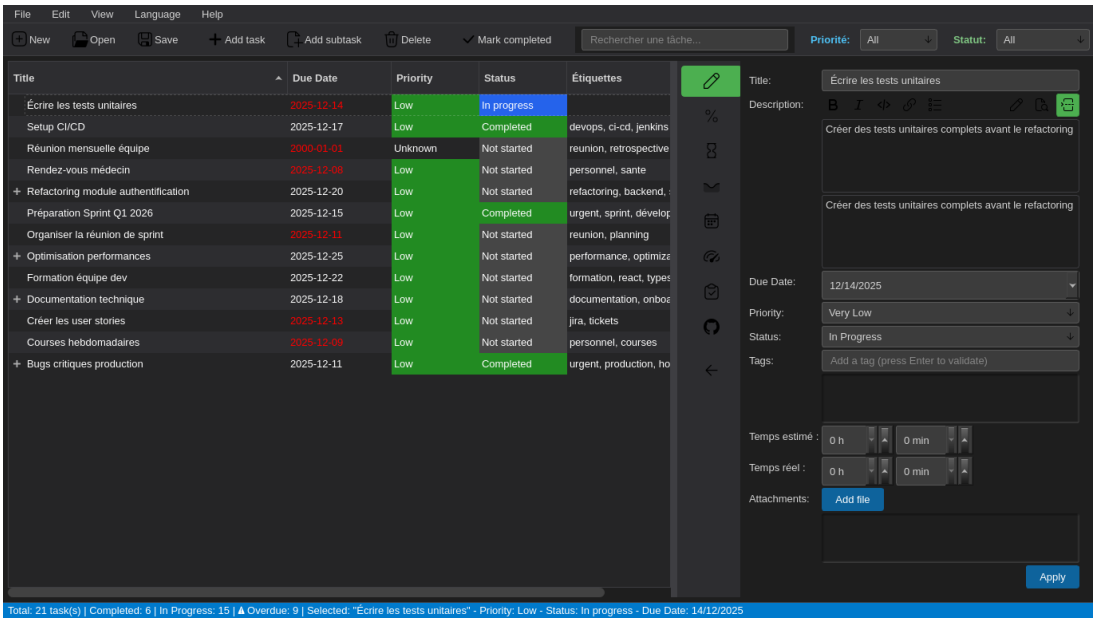

FIGURE 16 – Interface avec thème sombre

7.5.3 Thème chaleureux (Warm)

Le thème chaleureux combine des tons chauds et une luminosité intermédiaire, servant de compromis entre les thèmes clair et sombre. Il utilise des nuances orangées et ambrées.

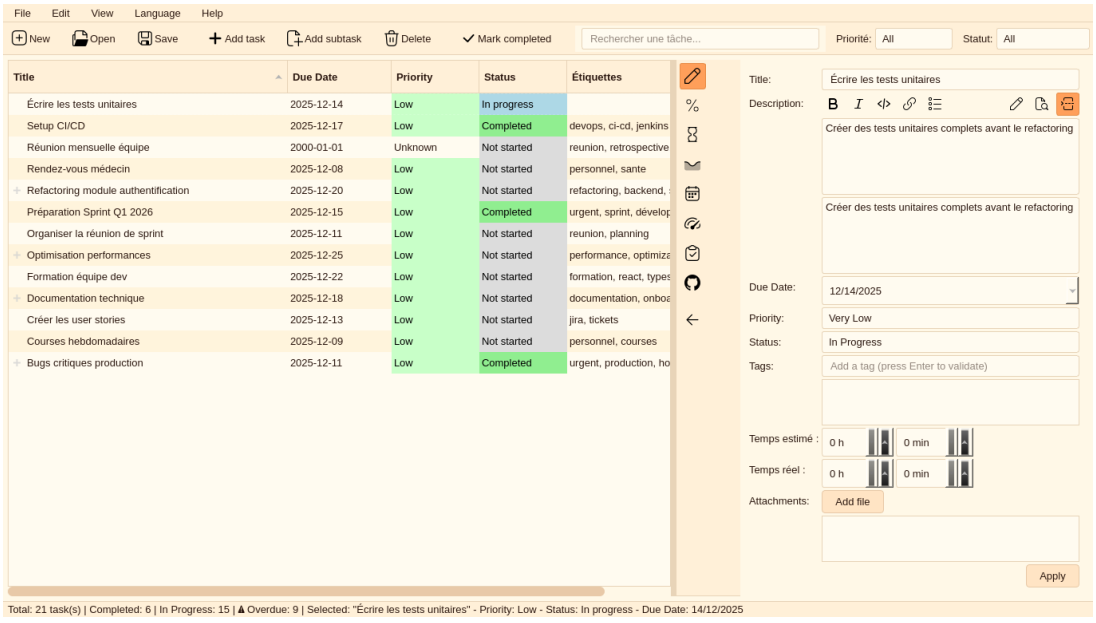

FIGURE 17 – Interface avec thème chaleureux

Implémentation technique. Le `ThemesManager` charge dynamiquement les fichiers QSS correspondants et applique les feuilles de style via `QApplication::setStyleSheet()`. Le thème actif est sauvegardé dans les préférences utilisateur (`QSettings`) et restauré automatiquement au démarrage de l'application. Le changement de thème est instantané et n'affecte pas l'état des données ni des fenêtres ouvertes. Tous les widgets de l'application respectent automatiquement le

thème actif grâce à l'utilisation systématique des classes de style Qt et à l'évitement des couleurs codées en dur dans le code C++.

7.6 Intégration Git

L'application propose un mode Git complémentaire (Ctrl+G) permettant de synchroniser les tâches avec les issues distantes (GitHub, GitLab, Gitea). `RepositoryManager` maintient une liste de dépôts configurés. `GitHubConnector` et `GitLabConnector` encapsulent les requêtes HTTP vers les API REST. `GitIssueTask` stocke les métadonnées Git supplémentaires (numéro d'issue, plateforme, assignés, milestone). La synchronisation bidirectionnelle permet de créer, mettre à jour, et importer des issues.

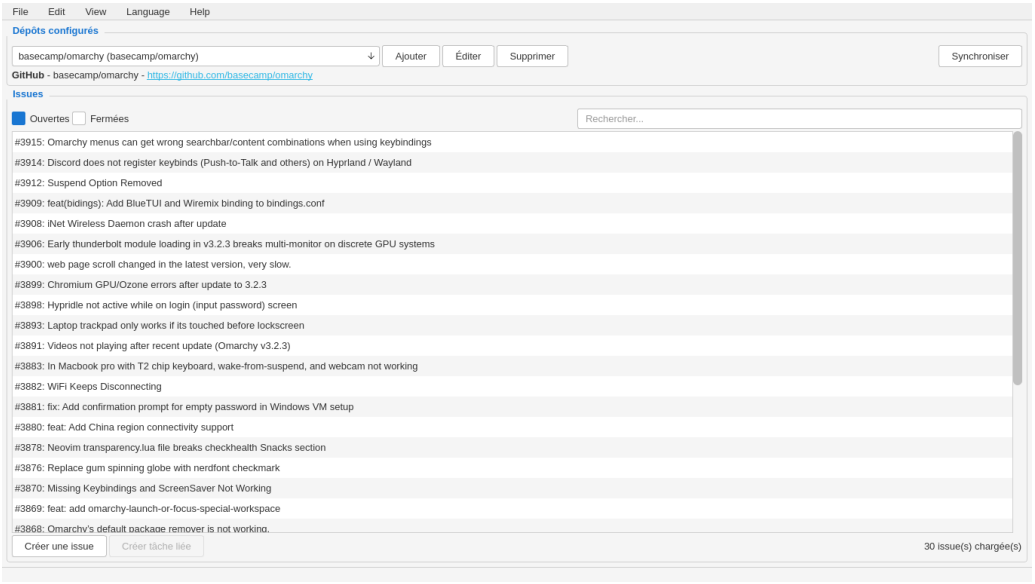

FIGURE 18 – Liste des issues Git synchronisées

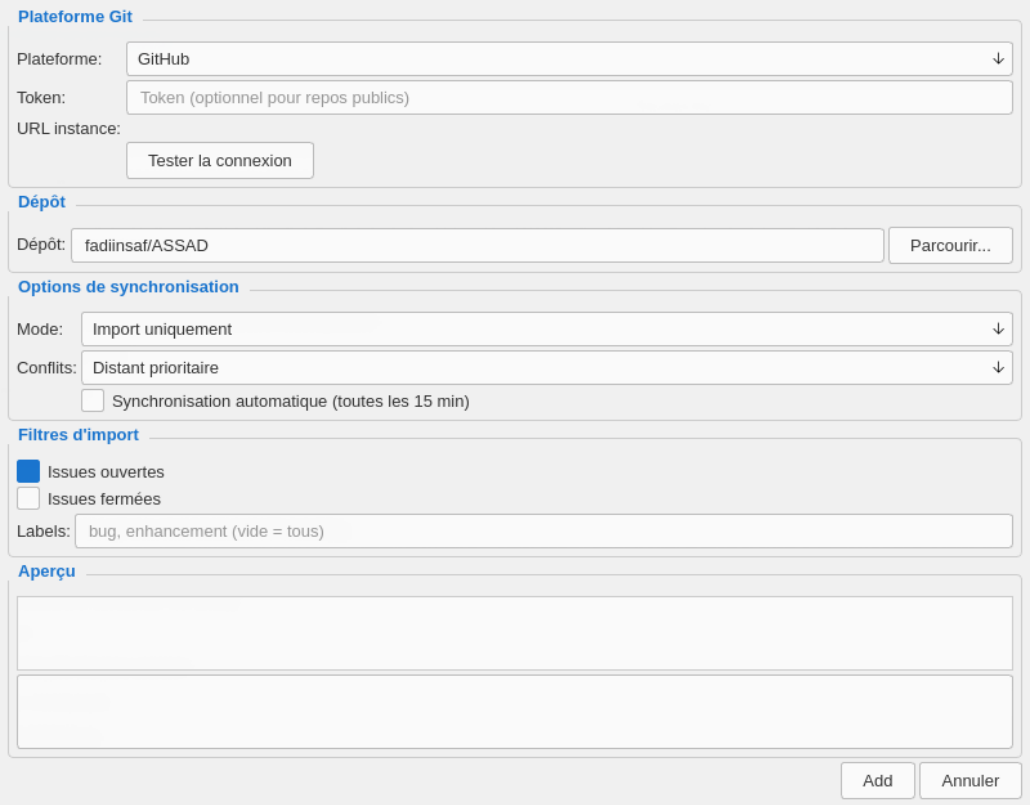

FIGURE 19 – Interface d'ajout de dépôt Git

Architecture Git. Le module Git s'articule autour de `GitProjectWidget` qui affiche la liste des dépôts configurés et permet d'ajouter/supprimer des connexions. Les connecteurs (`GitHubConnector`, `GitLabConnector`) héritent d'une interface commune et implémentent les appels API spécifiques à chaque plateforme. La synchronisation peut être manuelle (bouton "Sync") ou automatique (timer configurable). Les conflits de synchronisation sont détectés via les timestamps `lastSyncDate` et résolus selon une stratégie configurable (local prioritaire, distant prioritaire, ou demande utilisateur).

7.7 Internationalisation et traductions

L'application intègre un système complet d'internationalisation (i18n) via le framework de traduction Qt. Le projet prend en charge deux langues : le français (langue par défaut) et l'anglais américain. Les fichiers de traduction sont stockés dans le répertoire `translations/` :

- `ToDoApp_fr.ts` : Fichier source de traduction française
- `ToDoApp_en_US.ts` : Fichier source de traduction anglaise
- `ToDoApp_fr.qm` : Fichier compilé (binaire) pour le français
- `ToDoApp_en_US.qm` : Fichier compilé (binaire) pour l'anglais

Gestion des traductions. Les chaînes traduisibles dans le code C++ sont marquées avec la macro `tr()`, héritée de `QObject`. Pour les chaînes hors contexte de classe, la macro `QCoreApplication::translat` est utilisée. Le système génère automatiquement les fichiers `.ts` via `lupdate`, qui extrait toutes

les chaînes marquées comme traduisibles. Les traducteurs éditent ensuite ces fichiers XML via Qt Linguist, et `lrelease` les compile en fichiers `.qm` binaires chargés au runtime.

Détection de la langue. Au démarrage, l'application détecte la langue du système via `QLocale::system().na` et charge automatiquement le fichier de traduction correspondant. Si aucune traduction n'existe pour la langue système, l'application utilise le français par défaut. L'utilisateur peut changer la langue manuellement via le menu `Préférences > Langue`, et le changement prend effet immédiatement sans redémarrage grâce au mécanisme `QEvent::LanguageChange`.

Couverture des traductions. L'ensemble de l'interface utilisateur est traduit : menus, boutons, labels, messages d'erreur, tooltips, et dialogues. Les traductions incluent également les formats de date et d'heure localisés via `QLocale::toString()`, ainsi que les pluriels gérés automatiquement par le système Qt (`%n` dans les chaînes traduites). Le script `update_translations.sh` automatise la mise à jour des fichiers de traduction lors de l'ajout de nouvelles chaînes dans le code.

8 Justification des choix techniques

8.1 QAbstractItemModel pour la hiérarchie

`QAbstractItemModel` est la seule classe Qt permettant de modéliser des arbres de profondeur arbitraire tout en restant compatible avec `QTreeView`. Un modèle plat ou tabulaire ne peut représenter la structure parent-enfant récursive des tâches et sous-tâches.

8.2 Séparation MVC stricte

La séparation MVC stricte garantit la testabilité (classes de modèle testables sans widgets), facilite l'évolution (ajout de nouvelles vues sans modifier le modèle), et permet la réutilisation (le `TaskModel` pourrait être utilisé dans un autre contexte). La synchronisation automatique via signaux Qt élimine les bugs liés à des états incohérents.

8.3 Édition in-place

L'édition in-place permet à l'utilisateur de modifier rapidement une valeur sans ouvrir de dialogue modal, ce qui est plus pratique. Cette approche suit les recommandations ergonomiques modernes (Material Design, Human Interface Guidelines).

8.4 Proxy model pour le filtrage

Le proxy préserve l'intégrité des données : le filtrage est purement présentationnel et n'affecte jamais le modèle source. Cette approche respecte le principe de responsabilité unique : `TaskModel` gère les données, `TaskFilterProxyModel` gère le filtrage.

8.5 Intégration Git

L'intégration Git permet aux équipes de développement qui utilisent GitHub/GitLab de synchroniser leurs issues. ToDoApp offre des vues complémentaires (Kanban, Timeline, Pomodoro) et des fonctionnalités absentes des interfaces web. La synchronisation bidirectionnelle évite la duplication des données.

9 Conclusion

Ce projet démontre une implémentation de l'architecture MVC avec Qt. L'application ToDoApp utilise les mécanismes du framework : modèles personnalisés dérivés de `QAbstractItemModel`, modèles proxy de filtrage, délégués d'édition, signaux et slots pour la synchronisation automatique, et framework Undo/Redo.

L'architecture est modulaire, testable et extensible. L'ajout de nouvelles fonctionnalités est possible sans modifier la structure existante grâce à la séparation des responsabilités et à l'utilisation des patterns de conception de Qt. Les fonctionnalités implémentées incluent les opérations de base (création, modification, suppression, filtrage, recherche, persistance) ainsi que des fonctionnalités avancées (intégration Git, vue Kanban, timer Pomodoro, visualisations statistiques).