Smart Kanban Workflow Manager

Final Team Project Written Report

CMPE 202 - Software Systems Engineering

San Jose State University

Team: Model Builders

Alekya Gudise | Michael Kennedy

Project type	C++17 desktop application
GUI framework	Qt 6 Widgets
Build system	CMake
Main grading focus	Architecture, GUI, UML diagrams, design principles, design patterns, run instructions
Source package	smart-kanban-fixed source folder: src/domain, src/patterns, src/services, src/ui, CMakeLists.txt, BUILD.md

Table of Contents

- 1. Project Overview
- 2. Build, Run, and Screenshot Evidence
- 3. GUI and Feature Walkthrough
- 4. Code-Aligned Architecture
- 5. UML Diagrams
- 6. Design Patterns Mapped to the Code
- 7. Good Design Principles Used in the Code
- 8. Important Workflows
- Conclusion

Executive Summary

We built Smart Kanban Workflow Manager as a Qt 6/C++17 desktop Kanban application to demonstrate software systems engineering concepts through a working GUI application. Our app lets a software team create, edit, delete, move, filter, approve, and track tasks across a five-stage workflow. We did not make it just a GUI mockup; we intentionally organized the code around a layered architecture, a service facade, explicit workflow states, undoable commands, observer-driven refresh, strategy-based filtering, factory-based task creation, and a chain of responsibility for automation notifications.

In this report, we explain our application, give concrete build and run instructions, describe the GUI, include UML-style architecture and workflow diagrams, and map design principles and design patterns directly to our submitted code.

1. Project Overview

Our application is a desktop Kanban workflow manager for tracking software work items. A task begins in To Do, can move through In Progress and Review, can be marked Blocked when progress stops, and ends in Done after review approval. We support three task categories - Feature, Bug, and Chore - and four priorities - Low, Medium, High, and Critical. Each task stores a title, description, assignee, priority, due date, workflow stage, task type, and manager approval status.

Workflow stage	Meaning in the application
To Do	New or planned work waiting to start.
In Progress	Work currently being implemented.
Review	Work waiting for manager review and approval.
Blocked	Work that cannot proceed because of an issue or dependency.
Done	Completed work. In this project, Review-to-Done requires manager approval.

We split the user-facing behavior into two roles. Developer mode gives the normal project execution tools: Add Task, Edit Task, Delete Selected, drag-and-drop movement, filtering, Undo, and Redo. Manager mode changes the board into a review dashboard. In that mode, we disable ordinary editing and dragging, show Approve and Reject controls on Review-stage cards, and provide a Team Workload tab that summarizes each assignee's work distribution.

Role	Main responsibilities
Developer	Add, edit, delete, move, filter, undo, redo, and inspect activity/automation panels.
Manager	Approve or reject Review tasks, inspect manager banner, and view Team Workload reporting.

Our application demonstrates that object-oriented design can directly support visible product features. For example, our Command pattern is visible through Undo/Redo, our Observer pattern is visible through automatic board and log refreshes, our State pattern is visible through rejected invalid transitions, and our Strategy pattern is visible through runtime filters.

2. Build, Run, and Screenshot Evidence

We configured the project as a CMake-based Qt Widgets application. The target executable is named SmartKanban. Our submitted BUILD.md file also includes platform-specific packaging notes for macOS, Windows, and Linux.

Requirement	Details
Qt	Qt 6.4 or later with the Widgets module.
CMake	3.21 or later.
C++ compiler	C++17-capable compiler such as Apple Clang, GCC/Clang, or MSVC 2019+.
Optional IDE	VS Code with CMake Tools/C++ extensions, or Qt Creator.

```
Last login: Mon May 18 18:51:51 on ttys005
Alekya@MacBook smart-kanban-fixed 5 % rm -rf build
Alekya@MacBook smart-kanban-fixed 5 % cmake -S . -B build -DCMAKE_BUILD_TYPE=Release
cmake --build build --config Release
open build/SmartKanban.app
-- The CXX compiler identification is AppleClang 17.0.0.17000603
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD - Success
-- Found Threads: TRUE
-- Performing Test HAVE_STDATOMIC
-- Performing Test HAVE_STDATOMIC - Success
-- Found WrapAtomic: TRUE
-- Found OpenGL: /Library/Developer/CommandLineTools/SDKs/MacOSX.sdk/System/Library/Frameworks/OpenGL.framework
-- Found WrapOpenGL: TRUE
-- Could NOT find WrapVulkanHeaders (missing: Vulkan_INCLUDE_DIR)
-- Configuring done (1.2s)
-- Generating done (0.0s)
-- Build files have been written to: /Users/Alekya/Downloads/smart-kanban-fixed 5/build
[  0%] Built target SmartKanban_autogen_timestamp_deps
[  5%] Automatic MOC and UIC for target SmartKanban
[  5%] Built target SmartKanban_autogen
[ 11%] Building CXX object CMakeFiles/SmartKanban.dir/SmartKanban_autogen/mocs_compilation.cpp.o
[ 16%] Building CXX object CMakeFiles/SmartKanban.dir/src/main.cpp.o
[ 22%] Building CXX object CMakeFiles/SmartKanban.dir/src/domain/workflowstate.cpp.o
[ 27%] Building CXX object CMakeFiles/SmartKanban.dir/src/domain/task.cpp.o
[ 33%] Building CXX object CMakeFiles/SmartKanban.dir/src/domain/taskfactory.cpp.o
[ 38%] Building CXX object CMakeFiles/SmartKanban.dir/src/domain/board.cpp.o
[ 44%] Building CXX object CMakeFiles/SmartKanban.dir/src/patterns/automation.cpp.o
[ 50%] Building CXX object CMakeFiles/SmartKanban.dir/src/patterns/command.cpp.o
[ 55%] Building CXX object CMakeFiles/SmartKanban.dir/src/services/boardservice.cpp.o
[ 61%] Building CXX object CMakeFiles/SmartKanban.dir/src/ui/taskdialog.cpp.o
[ 66%] Building CXX object CMakeFiles/SmartKanban.dir/src/ui/tasklistwidget.cpp.o
[ 72%] Building CXX object CMakeFiles/SmartKanban.dir/src/ui/boardwidget.cpp.o
[ 77%] Building CXX object CMakeFiles/SmartKanban.dir/src/ui/activitylogwidget.cpp.o
[ 83%] Building CXX object CMakeFiles/SmartKanban.dir/src/ui/notificationwidget.cpp.o
[ 88%] Building CXX object CMakeFiles/SmartKanban.dir/src/ui/mainwindow.cpp.o
[ 94%] Building CXX object CMakeFiles/SmartKanban.dir/src/ui/teamworkloadwidget.cpp.o
[100%] Linking CXX executable SmartKanban.app/Contents/MacOS/SmartKanban
[100%] Built target SmartKanban
Alekya@MacBook smart-kanban-fixed 5 %
```

Figure 1. Build and run evidence: CMake configures, compiles, links SmartKanban, and opens the Qt application.

2.1 Recommended macOS commands

```
# From the extracted project root
brew install cmake qt
rm -rf build
cmake -S . -B build -DCMAKE_PREFIX_PATH="$(brew --prefix qt)"
cmake --build build
open build/SmartKanban.app
```

2.2 Generic CMake commands

```
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release
cmake --build build --config Release

# Run after build:
open build/SmartKanban.app       # macOS bundle
./build/SmartKanban              # Linux or non-bundle macOS build
build\Release\SmartKanban.exe  # Windows Release build
```

2.3 Common Qt6 CMake issue

If CMake reports that Qt6Config.cmake or qt6-config.cmake cannot be found, the Qt development files are missing or CMake does not know where Qt is installed. We fix that by installing Qt6 and passing the Qt CMake path through CMAKE_PREFIX_PATH, for example:

```
cmake -S . -B build -DCMAKE_PREFIX_PATH="$HOME/Qt/6.7.0/gcc_64/lib/cmake"
```

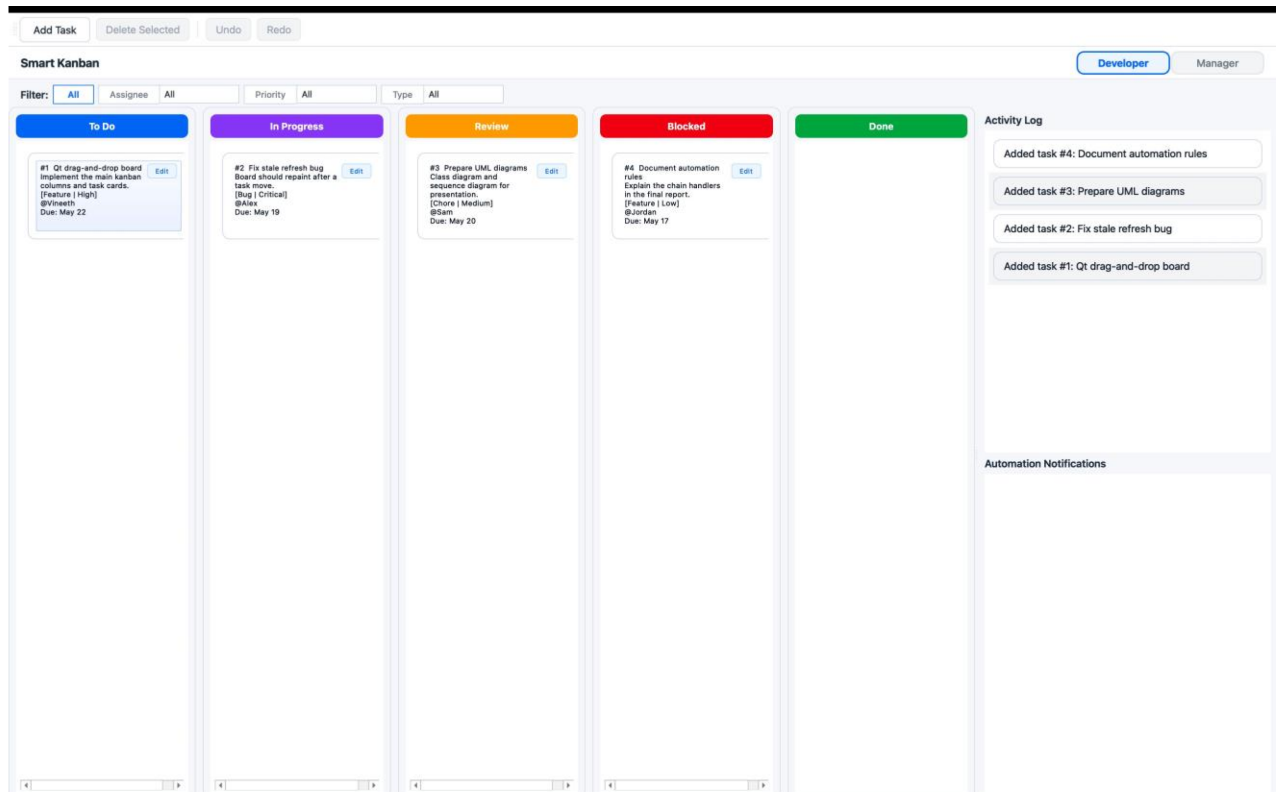


Figure 2. Running SmartKanban application after launch in Developer mode, showing the five workflow columns, seeded task cards, role switcher, filters, and Activity Log.

3. GUI and Feature Walkthrough

We built the GUI with Qt Widgets. MainWindow assembles the application window, role switcher, developer toolbar, filter bar, central board area, activity log, automation notification panel, manager banner, and manager tabs. BoardWidget renders five workflow columns, while each column is a TaskListWidget that owns card selection and drag/drop behavior.

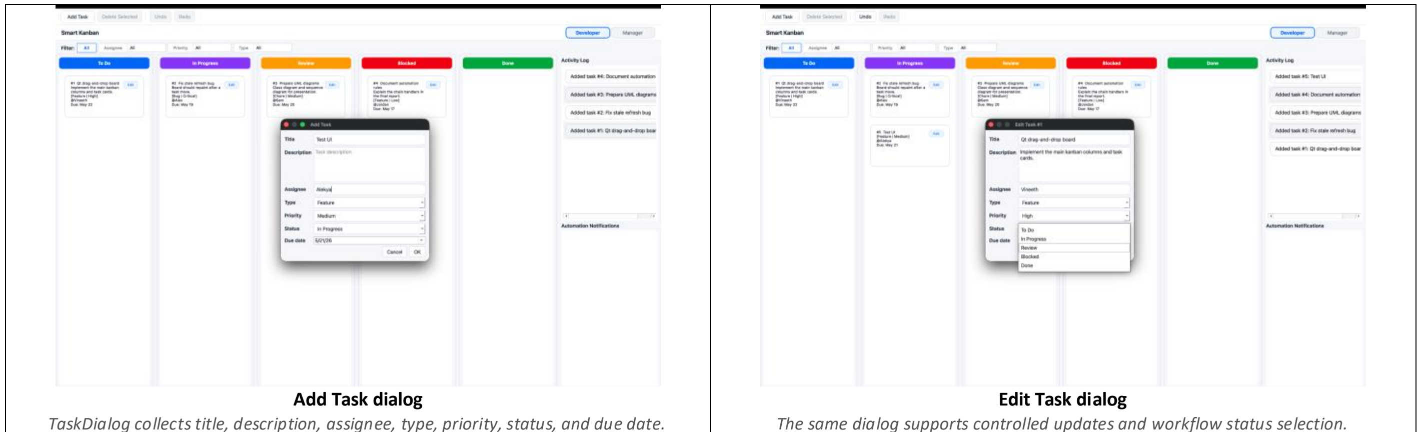
GUI area	Behavior	Main code
Top role switcher	Switches between Developer and Manager modes. Developer mode shows edit tools; Manager mode shows approval/workload views.	MainWindow::switchRole()
Developer toolbar	Add Task, Delete Selected, Undo, and Redo controls.	MainWindow constructor, updateButtons()
Filter bar	All/Assignee/Priority/Type filter controls. Multiple combo values are combined using CompositeFilterStrategy.	MainWindow::buildFilterBar(), applyFilter()
Kanban board	Five colored workflow columns containing task cards.	BoardWidget, TaskListWidget
Task dialog	Collects title, description, assignee, type, priority, status, and due date.	TaskDialog
Side panel	Displays activity log and automation notifications.	ActivityLogWidget, NotificationWidget
Manager dashboard	Shows banner, Board View, Team Workload, and approval buttons on Review cards.	MainWindow, TeamWorkloadWidget

At startup, our main.cpp creates QApplication, applies the Fusion style and palette, creates BoardService, constructs MainWindow, seeds demo data, and shows the window. We include seeded demo tasks so the presentation can immediately demonstrate all columns and automation behavior without requiring us to create data from scratch.

Figure 2 provides the embedded run screenshot for the written-report rubric. It shows the running application in Developer mode with workflow columns, seeded tasks, filters, the role switcher, and the Activity Log.

3.1 Task Creation, Editing, and Filtering

The next user story is task management. A developer opens the Add Task dialog, enters the task information, and submits the form. The application then creates the correct concrete task type through the factory, records the action through the command system, and notifies the UI through observers.



Add Task dialog

TaskDialog collects title, description, assignee, type, priority, status, and due date.

Edit Task dialog

The same dialog supports controlled updates and workflow status selection.

Filtering is intentionally implemented as a view concern. Selecting an assignee, priority, or type does not modify stored tasks; it only changes which cards BoardWidget renders. This is why the filtering behavior maps cleanly to the Strategy pattern.

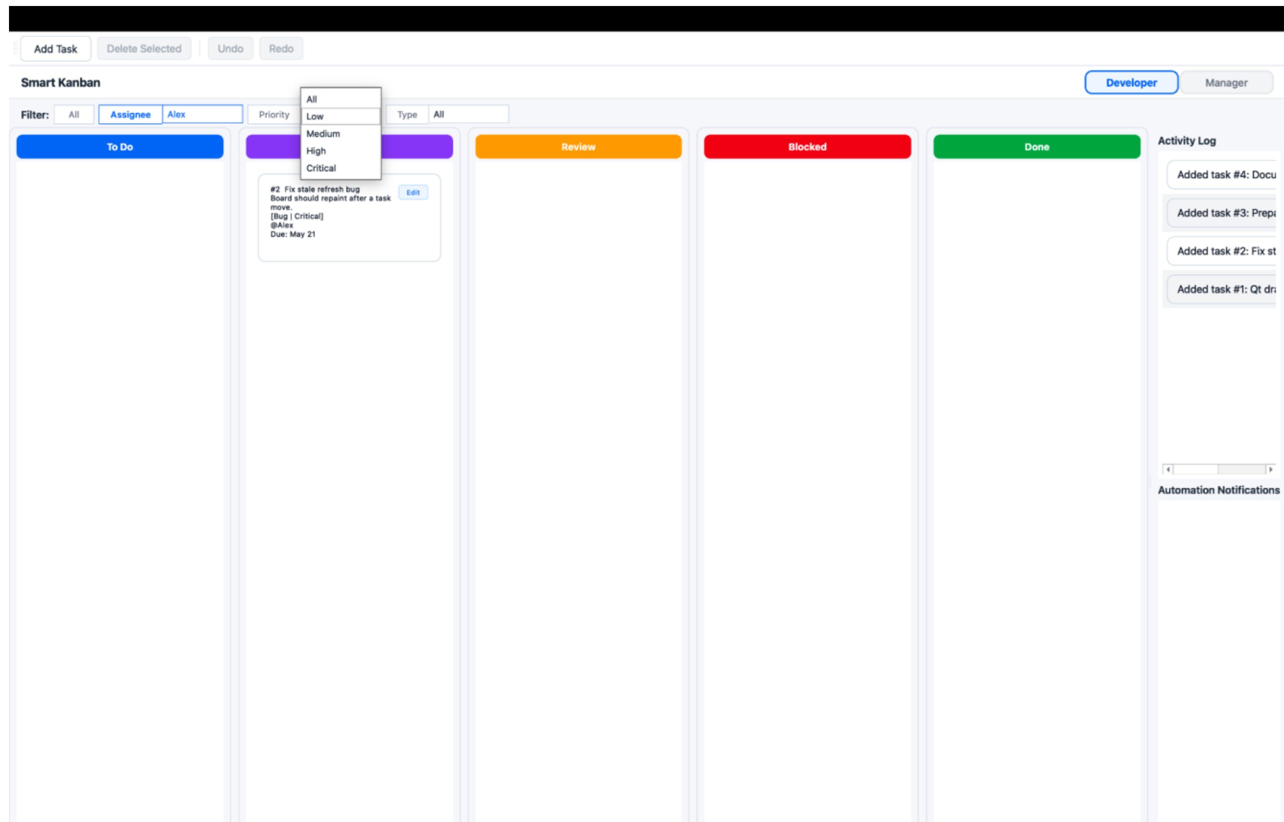
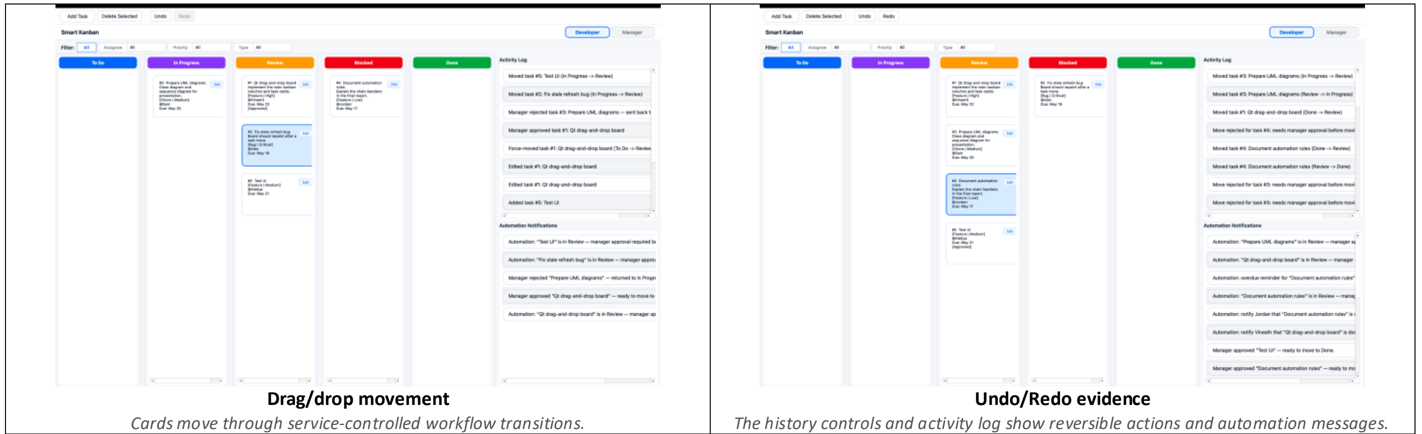


Figure 5. Filter evidence: assignee and priority filters narrow the board without changing task data.

3.2 Movement, Undo/Redo, Automation, and Manager Approval

The main board interaction is drag/drop movement. TaskListWidget emits a taskDropped(taskId, targetStage) signal instead of directly moving the card. This is important because BoardService must validate the transition, update command history, run automation rules, and notify observers before the screen changes.



Drag/drop movement

Cards move through service-controlled workflow transitions.

Undo/Redo evidence

The history controls and activity log show reversible actions and automation messages.

Manager mode adds the approval story. Tasks in Review show Approve and Reject controls. A task cannot simply jump from Review to Done unless the manager approval status is Approved. Rejection returns the work to In Progress for rework.

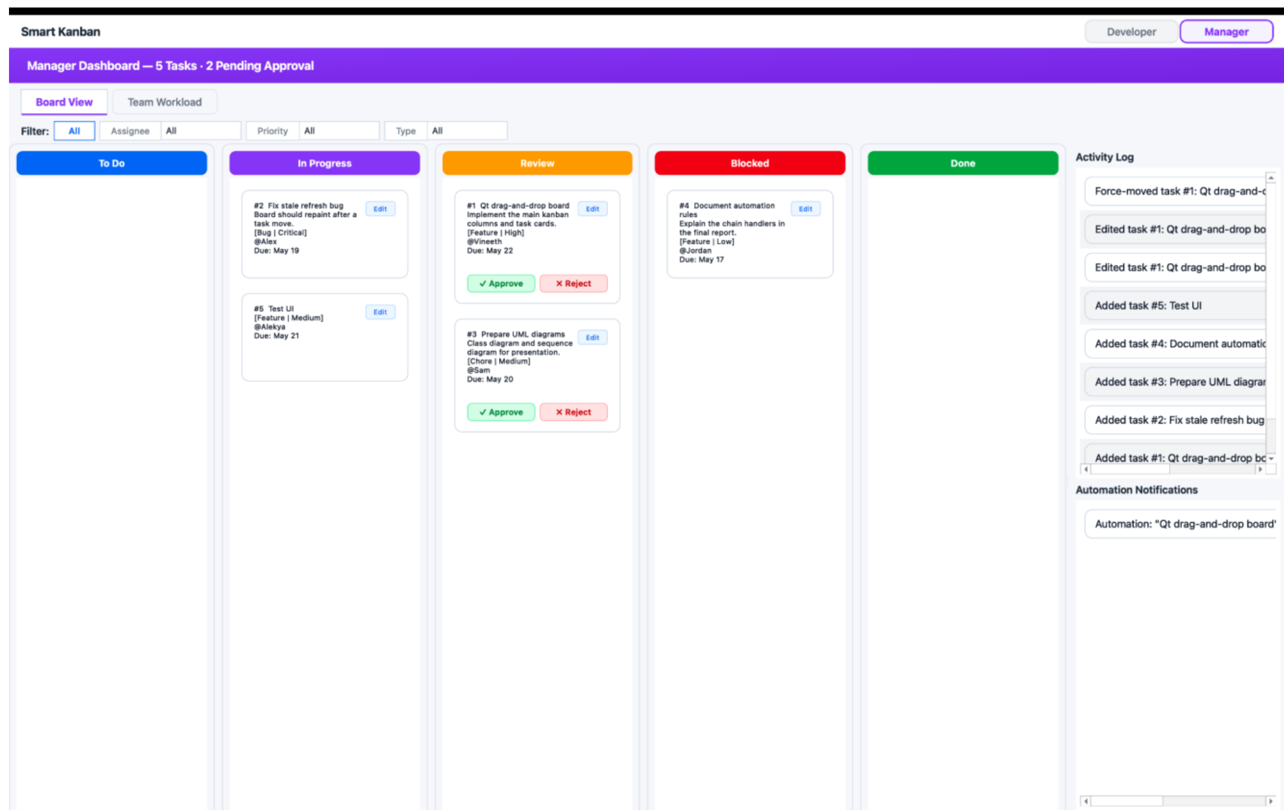


Figure 8. Manager dashboard: pending approvals with Approve and Reject controls on Review cards.

4. Code-Aligned Architecture

We organized the code into four main areas: domain, pattern infrastructure, service orchestration, and UI. This organization is visible in the src folder and in the CMakeLists.txt source list. Our architecture keeps data modeling, workflow rules, reusable pattern infrastructure, application use cases, and visual presentation separated.

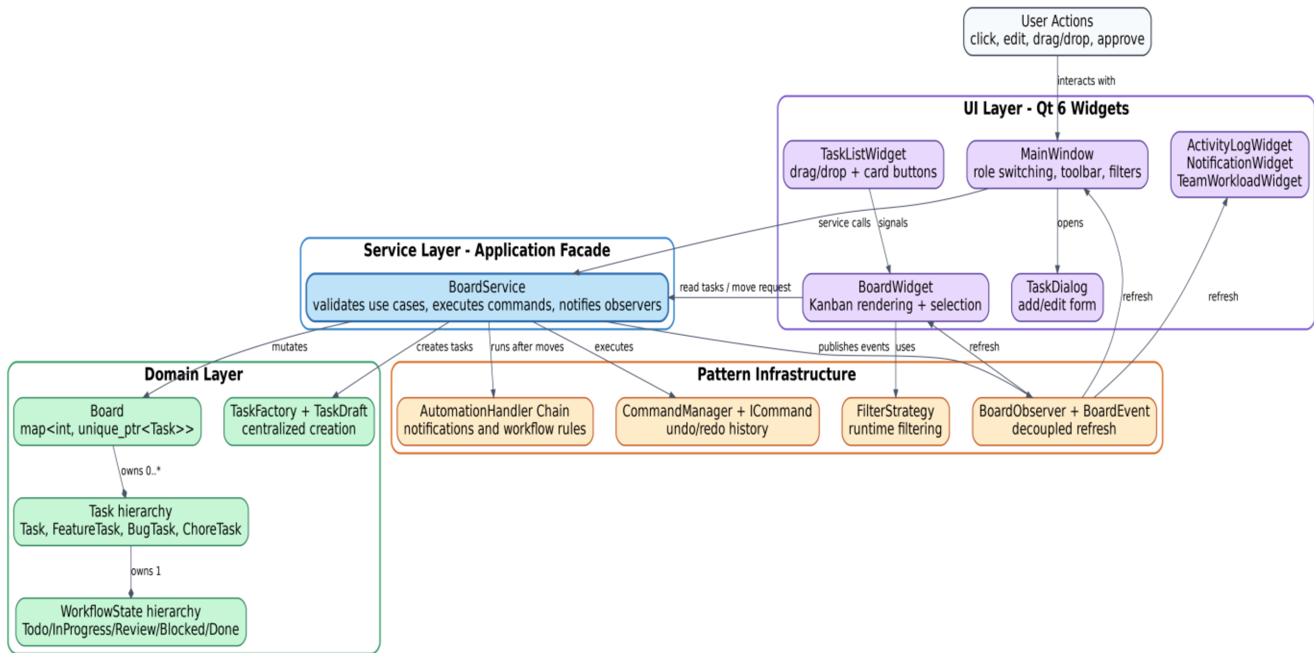


Figure 9. Architecture diagram showing how user actions enter through Qt UI widgets, flow through BoardService, and then reach the domain model, command system, workflow rules, observers, filters, and automation handlers.

Layer	Folder	Important classes	Responsibility
Domain	src/domain	Task, FeatureTask, BugTask, ChoreTask, TaskDraft, Board, WorkflowState, TaskFactory, enums	Models tasks, priorities, workflow stages, approval state, legal transitions, and task creation.
Pattern infrastructure	src/patterns	ICommand, CommandManager, BoardObserver, BoardEvent, FilterStrategy, AutomationHandler	Provides reusable mechanisms for undo/redo, observer notifications, filters, and automation messages.
Service	src/services	BoardService	Acts as the application facade. Coordinates Board, TaskFactory, commands, observers, automation, and validation.
UI	src/ui	MainWindow, BoardWidget, TaskListWidget, TaskDialog, ActivityLogWidget, NotificationWidget, TeamWorkloadWidget	Builds and refreshes the Qt interface. Sends user requests to BoardService and reacts to BoardEvent updates.

A core design decision we made is that normal board mutation goes through BoardService. Our UI classes do not directly insert, remove, or transition tasks inside Board. Instead, user gestures become service calls such as createTask(), editTask(), deleteTask(), moveTask(), approveTask(), rejectTask(), undo(), and redo(). This keeps validation, command history, automation messages, and observer notification in one reliable orchestration point.

5. UML Diagrams

5.1 Class and Pattern Relationship Diagram

The diagram below summarizes our main class relationships without listing every private UI widget. BoardService is the center of our application use cases. It owns the board, factory, command manager, automation chain, and observers. Our UI talks to the service; the service talks to the domain and pattern infrastructure.

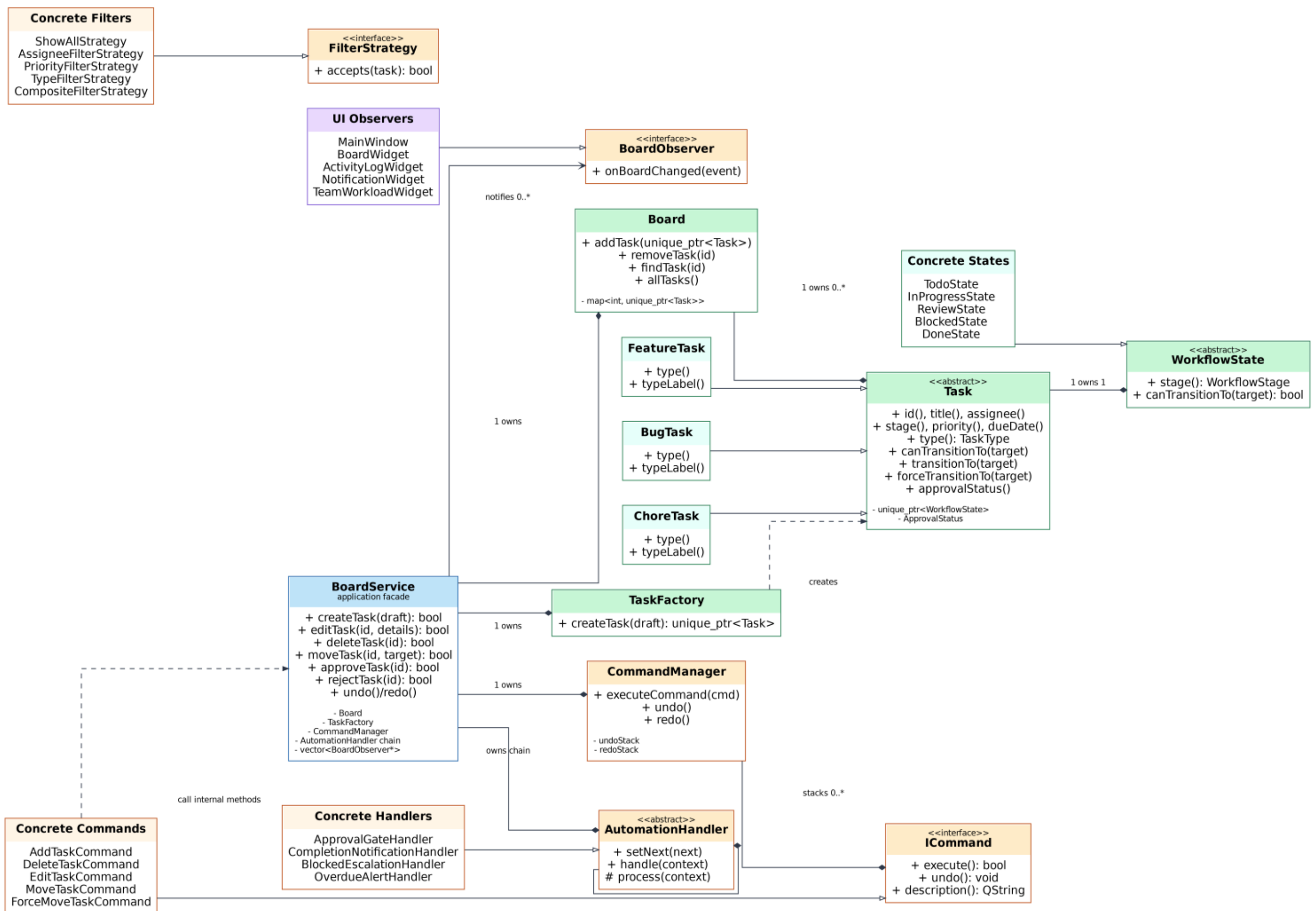


Figure 10. Simplified UML class and pattern relationship diagram.

The important UML relationships we show include inheritance from abstract pattern interfaces, composition through std::unique_ptr ownership, and observer registration through BoardObserver pointers. We deliberately group concrete classes by role so the diagram stays readable for the written report and oral presentation.

5.2 UML State and Sequence Diagrams

The state diagram describes the lifecycle rules for a task. The important rule is the approval gate: a Review task can move to Done only when it is approved. Otherwise, the service rejects the transition and the activity log records why it was not accepted.

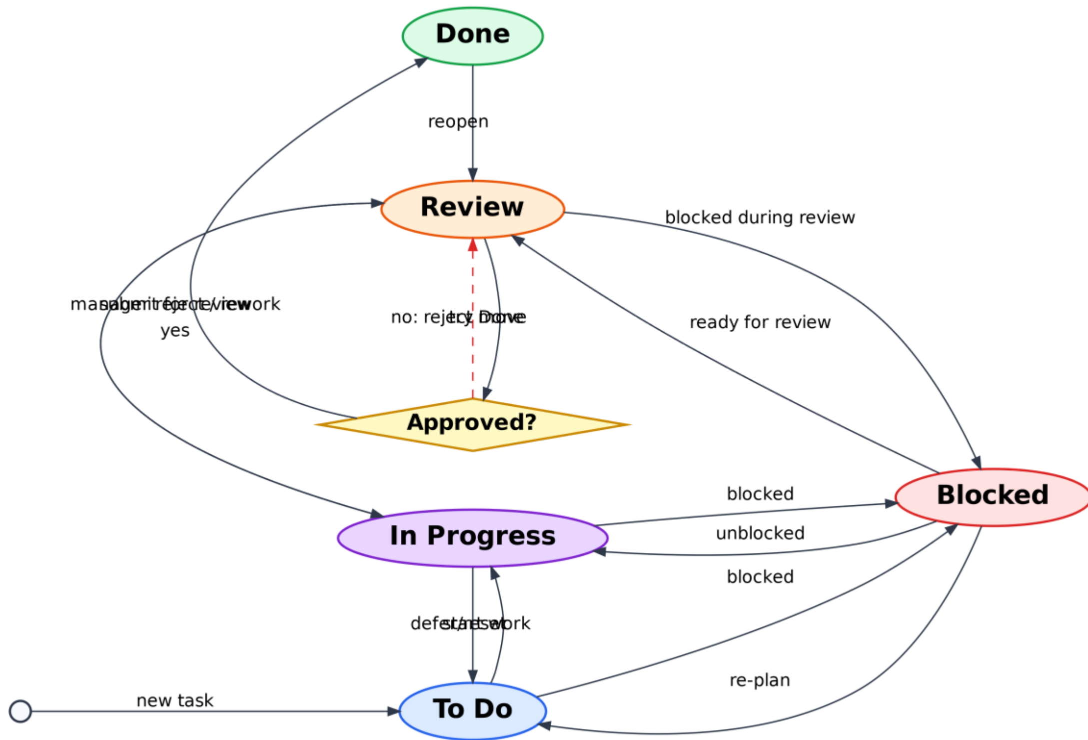
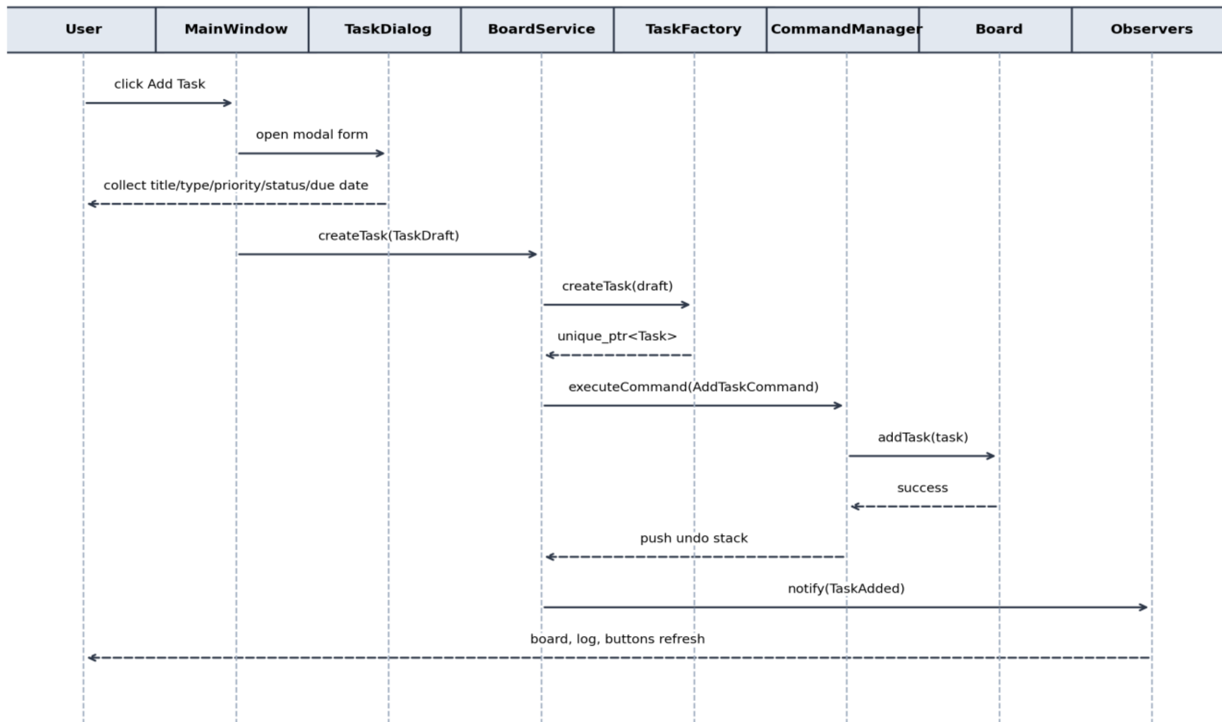


Figure 11. UML state diagram for workflow transitions and the Review-to-Done approval gate.

UML Sequence Diagram - Add Task with Command + Observer



Key point: the UI never directly mutates Board; the service creates a command and publishes a BoardEvent.

Figure 12. Add task sequence using TaskDialog, TaskFactory, Command, BoardService, Board, and Observers.

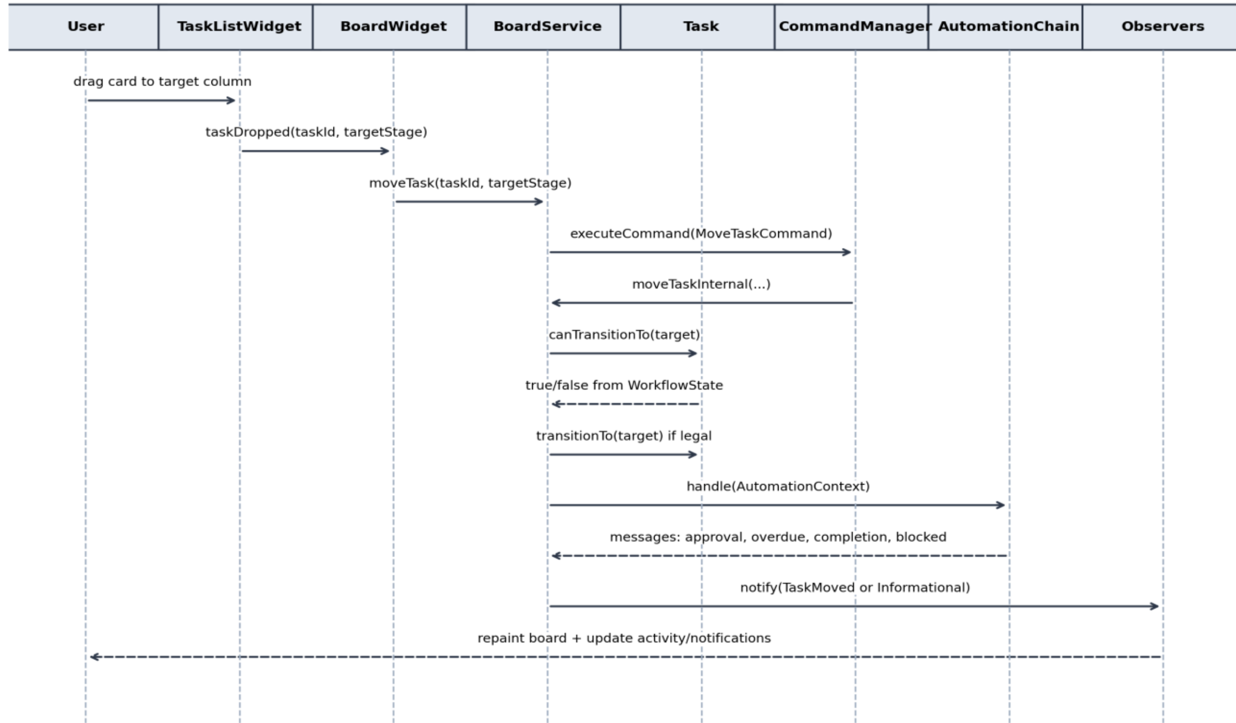


Figure 13. Drag/drop move sequence using State validation, Command execution, automation, and Observer refresh.

6. Design Patterns Mapped to the Code

We implemented six major design-pattern families. Each pattern solves a visible application problem and maps to concrete files/classes in our submitted code. We avoid overstating the implementation: for example, manager approve/reject is implemented as direct service behavior, while create/delete/edit/move/force-move are command-backed actions.

Pattern	Code location	Main classes	Feature supported
Command	src/patterns/command.*; BoardService	ICommand, CommandManager, AddTaskCommand, DeleteTaskCommand, EditTaskCommand, MoveTaskCommand, ForceMoveTaskCommand	Undo/redo for create, delete, edit, move, and explicit override moves.
Observer	boardobserver.h, boardevent.h, BoardService::notify(), UI widgets	BoardObserver, BoardEvent, MainWindow, BoardWidget, ActivityLogWidget, NotificationWidget, TeamWorkloadWidget	Board refresh, logs, notifications, and manager banner update after events.
State	workflowstate.*, task.*	WorkflowState, TodoState, InProgressState, ReviewState, BlockedState, DoneState	Legal transition rules for workflow movement.
Strategy	filterstrategy.h, MainWindow, BoardWidget	FilterStrategy, ShowAllStrategy, AssigneeFilterStrategy, PriorityFilterStrategy, TypeFilterStrategy, CompositeFilterStrategy	Runtime filtering without changing board data.
Factory	taskfactory.*, task.*	TaskFactory, TaskDraft, FeatureTask, BugTask, ChoreTask	Centralized concrete task creation and ID assignment.
Chain of Responsibility	automation.*; BoardService	AutomationHandler, ApprovalGateHandler, CompletionNotificationHandler, BlockedEscalationHandler, OverdueAlertHandler	Independent automation notifications after task movement.

6.1 Pattern explanations

Command

We use CommandManager to store std::unique_ptr<ICommand> objects on undo and redo stacks. Our commands capture enough state to reverse themselves: DeleteTaskCommand keeps the removed task, EditTaskCommand stores before/after details, and MoveTaskCommand stores source and target stages.

Observer

We have BoardService broadcast BoardEvent objects to registered observers. Each panel reacts independently, so BoardService does not need to know how the board, activity log, notification list, manager banner, or workload view is redrawn.

State

Each Task owns a WorkflowState. Each concrete state returns its current stage and decides which target stages are legal. This keeps transition logic out of our drag/drop UI code and prevents invalid moves from silently changing state.

Strategy

BoardWidget renders using the active FilterStrategy. We have MainWindow build a CompositeFilterStrategy from selected filter controls, so filters can be combined and replaced without rewriting the board rendering loop.

Factory

We use TaskFactory to convert a TaskDraft into the correct concrete Task subclass and assign IDs in one place. Callers depend on the Task abstraction instead of directly constructing FeatureTask, BugTask, or ChoreTask.

Chain of Responsibility

Our AutomationHandler calls its own process() method and forwards the same AutomationContext to the next handler. The default chain handles manager approval reminders, completion notifications, blocked escalations, and overdue warnings.

7. Good Design Principles Used in the Code

One of our strongest scoring areas is the direct connection between design principles and working code. We emphasize that these principles are not theoretical labels; they are present in how we structured responsibilities, ownership, dependencies, and extension points.

Principle	How the code demonstrates it
Single Responsibility Principle	Task stores task data and delegates workflow behavior. Board stores tasks. BoardService coordinates use cases. TaskDialog collects form input. TeamWorkloadWidget only builds workload reporting.
Open/Closed Principle	New filters can be added through FilterStrategy subclasses. New automation rules can be added by subclassing AutomationHandler. New commands can be added by implementing ICommand.
Liskov Substitution Principle	FeatureTask, BugTask, and ChoreTask are used through Task pointers. Workflow states are used through WorkflowState pointers. Commands are stored through ICommand pointers.
Interface Segregation Principle	BoardObserver exposes only onBoardChanged(). FilterStrategy exposes only accepts(). ICommand exposes only execute(), undo(), and description().
Dependency Inversion / abstraction use	CommandManager depends on ICommand. BoardService notifies BoardObserver. BoardWidget depends on FilterStrategy. Task depends on WorkflowState. These are the main abstraction points.
Encapsulation	Board owns a private task map. Task fields are accessed through methods. BoardService keeps internal mutation methods private and exposes public use-case methods.
Separation of Concerns	Drag/drop detection, board rendering, service orchestration, transition validation, undo/redo, automation, and logging are placed in separate classes.
Composition over inheritance	Task composes a WorkflowState. BoardWidget composes a FilterStrategy. BoardService composes an automation chain. Inheritance is reserved for stable polymorphic roles.

One important accuracy point is that our project does not define a separate IBoardService interface. That is acceptable for this project size because BoardService functions as a concrete application facade, while the major extension points still depend on abstractions. This explanation is more accurate than claiming the entire service layer is fully abstracted.

8. Important Workflows

Add/edit task. MainWindow opens TaskDialog, collects user input, and sends a draft or details object to BoardService. BoardService then uses TaskFactory and command objects so creation and editing remain consistent and reversible.

Drag/drop movement. TaskListWidget sends only a task ID and target stage. BoardService and WorkflowState validate the move before the board is changed, and observers refresh the GUI after the event.

Manager approval gate. Review-to-Done is guarded by approval status. If the task is not approved, the move is rejected and the Activity Log explains the reason. Approved tasks can then move to Done.

Filtering and Undo/Redo. Filters are applied through interchangeable strategies, while undo/redo is centralized in CommandManager instead of being duplicated inside GUI event handlers.

Conclusion

Smart Kanban Workflow Manager meets the CMPE 202 team project goals because it is a working C++17/Qt 6 GUI application with clear architecture, professional UML documentation, code-aligned design patterns, and code-aligned design principles. The project demonstrates a complete workflow manager rather than a static mockup: task creation, editing, movement, filtering, manager approval, rejection, workload reporting, activity logging, automation notifications, and undo/redo are all visible in the running application.

The current scope is honest: the app is an in-memory desktop application without database persistence or networking. Those would be natural future extensions, but the current version already demonstrates the required software architecture, GUI behavior, UML diagrams, design principles, design patterns, and run evidence needed for the written report rubric.