Assignment 3: Command

Software Design Patterns

Student ID: XXXXXXXXX
Name: Colby Frison
Course: CS 4213 - Fall 2025

January 20, 2026

Contents

1 Part A: MVC Design

1.1 Selected System: Note-taking Application

I have chosen to analyze a **Note-taking Application**, a system that allows users to create, edit, organize, and search through personal notes. The system provides a clear separation of concerns that naturally maps to the Model-View-Controller (MVC) architectural pattern.

The note-taking application includes the following core features:

- Creating and editing notes with title and content
- Organizing notes into notebooks
- Tagging notes with categories for better organization
- Searching through notes by content or tags
- Viewing lists of notes with filtering capabilities

1.2 MVC Component Definitions

1.2.1 Model Components

The Model layer represents the business data and domain logic of the application. It is independent of the user interface and contains the core entities and data access operations.
Note: The core data entity representing a single note. It contains:

- Unique identifier (id)
- Title and content (text fields)
- Creation and modification timestamps
- Associated tags for categorization

The Note class encapsulates its data and provides controlled access through getter and setter methods. It also manages its own state, such as updating the last modified timestamp when content changes.

Notebook: A container for organizing multiple notes. It provides:

- A name for the notebook
- A collection of notes
- Methods to add, remove, and query notes

Tag: Represents a category or label that can be applied to notes. Tags allow users to organize notes across different notebooks and enable filtering and searching by category.

NoteRepository: The data access layer that handles persistence operations. It abstracts the storage mechanism (file system, database, etc.) and provides methods for:

- Saving and loading notes
- Searching notes by content or tags
- Deleting notes
- Querying all notes

The repository pattern ensures that the rest of the application does not need to know about the specific storage implementation, making it easier to change storage mechanisms in the future.

1.2.2 View Components

The View layer is responsible for presenting information to the user and capturing user input. Views are passive components that display data and notify controllers of user actions, but they do not directly manipulate the model.

NoteListView: Displays a list of all notes with basic information such as title, creation date, and associated tags. It provides functionality to:

- Display notes in a scrollable list
- Show note details on selection
- Refresh the display when notes are updated
- Notify the controller when a note is selected

NoteEditorView: A rich text editor interface for creating and editing notes. It provides:

- Input fields for title and content
- Save and cancel buttons
- Display of existing note data when editing
- User action notifications to the controller

SearchView: The search interface that allows users to find notes by content. It includes:

- A search input box
- Display of search results
- Submission of search queries to the controller

CategoryView: Interface for managing tags and categories. It allows users to:

- View all available tags
- Create new tags
- Select tags to filter notes
- Manage tag properties (name, color)

1.2.3 Controller Components

Controllers act as intermediaries between views and the model. They receive user input from views, coordinate operations on the model, and update views with results. Controllers contain no business logic themselves but orchestrate the interaction between components.

NoteController: Handles all note-related operations including:

- Creating new notes
- Loading and displaying notes for editing
- Saving note changes
- Deleting notes
- Loading all notes for list display
- Coordinating between NoteListView and NoteEditorView

SearchController: Manages search functionality by:

- Receiving search queries from SearchView
- Coordinating with NoteRepository to perform searches
- Filtering and formatting search results
- Updating SearchView with results

CategoryController: Handles tag and category operations including:

- Creating new tags
- Adding tags to notes
- Removing tags from notes
- Loading all tags for display

1.3 UML Class Diagram

The UML class diagram (Figure 1) illustrates the complete MVC architecture of the note-taking application. The diagram is organized into three packages representing the Model, View, and Controller layers.

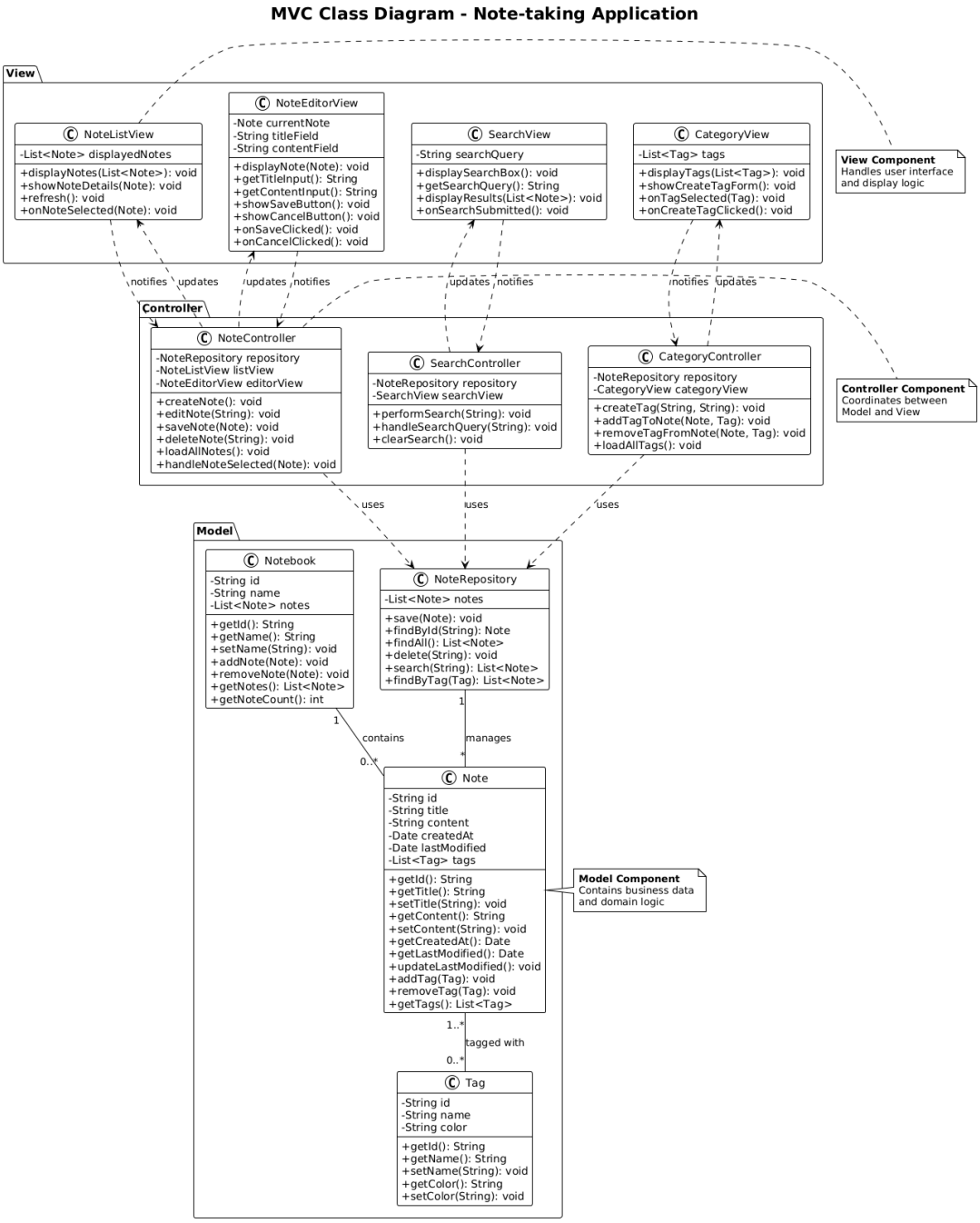

Figure 1: UML Class Diagram showing MVC architecture with Model, View, and Controller components

1.4 Sequence Diagram: Create Note Interaction

The sequence diagram (Figure 2) illustrates a complete user interaction for creating a new note. This interaction demonstrates how all three MVC components work together to fulfill a user request.

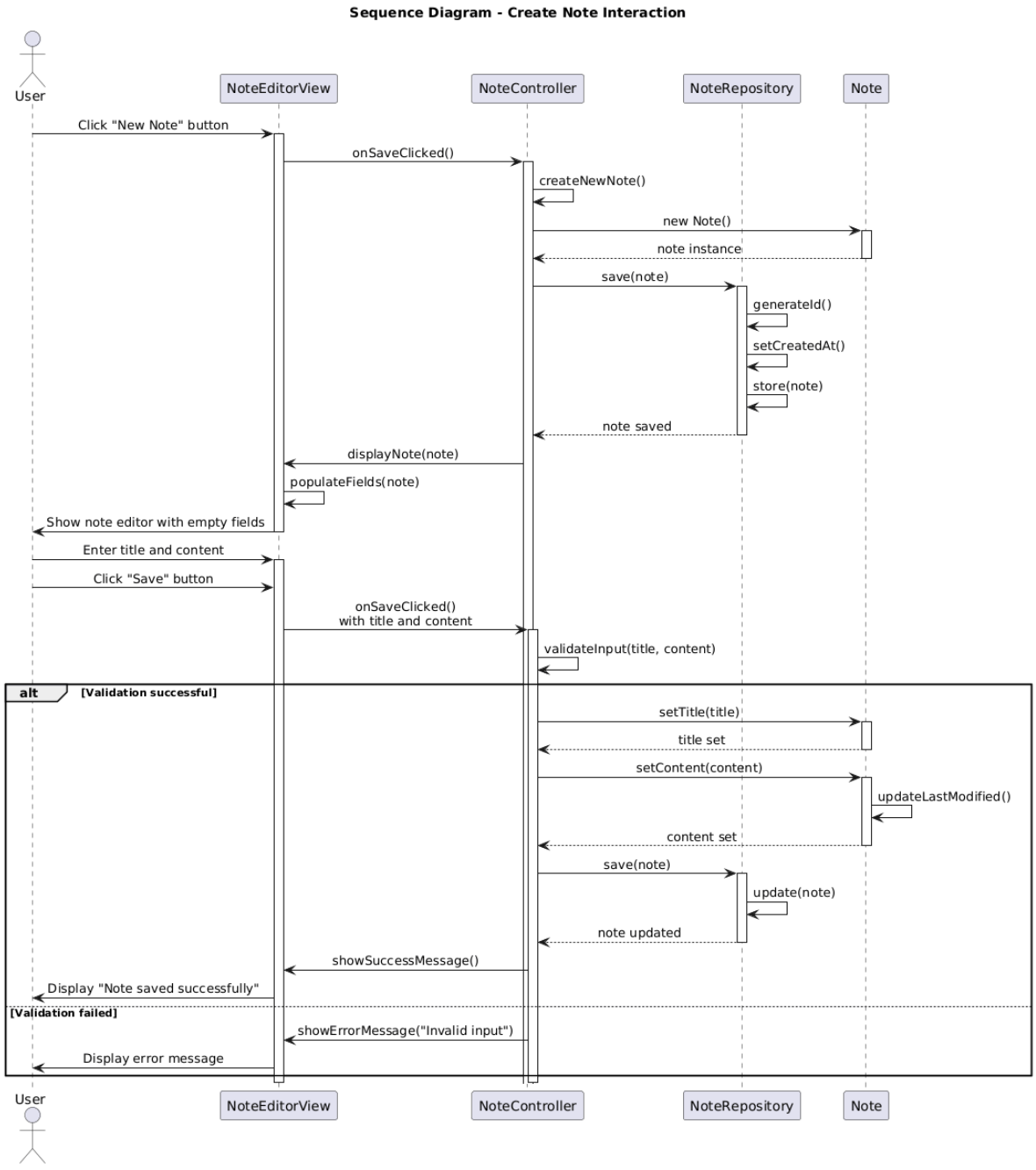

Figure 2: Sequence Diagram showing the complete "Create Note" user interaction flow

2 Part B: Refactoring Plan

This section presents a refactoring plan for improving a tightly coupled version of the note-taking application. The current system has several design issues that reduce maintainability, flexibility, and testability. Three specific refactoring areas are identified, each targeting a different quality attribute: **flexibility**, **reuse**, and **testability**.

2.1 Refactoring 1: Extract Interface for Flexibility

2.1.1 Problem Description

The current system has a tight coupling between the `NoteController` and the concrete `FileStorage` implementation. The controller directly instantiates and depends on `FileStorage`, making it impossible to switch to alternative storage mechanisms (such as `DatabaseStorage` or `CloudStorage`) without modifying the controller code.
This creates several problems:

- **Lack of Flexibility:** Changing storage mechanisms requires modifying controller code, violating the Open/Closed Principle.
- **Difficulty Testing:** Unit testing the controller requires actual file system operations, making tests slow and dependent on the file system state.
- **Vendor Lock-in:** The system is tightly bound to file-based storage, making it difficult to support multiple storage backends or allow users to choose their preferred storage method.
- **Code Duplication:** If multiple controllers need storage access, each would need to know about the concrete storage implementation.

2.1.2 Refactoring Technique: Extract Interface

The solution is to apply the **Extract Interface** refactoring technique. This involves:

1. Creating an `IStorageRepository` interface that defines the common operations needed by controllers
2. Making `FileStorage`, `DatabaseStorage`, and any future storage implementations implement this interface
3. Modifying `NoteController` to depend on the interface rather than the concrete class
4. Using dependency injection to provide the specific storage implementation at runtime

This refactoring follows the Dependency Inversion Principle, which states that high-level modules should not depend on low-level modules; both should depend on abstractions.

2.1.3 UML Diagram: Before and After

Figure 3 shows the system before and after the refactoring.

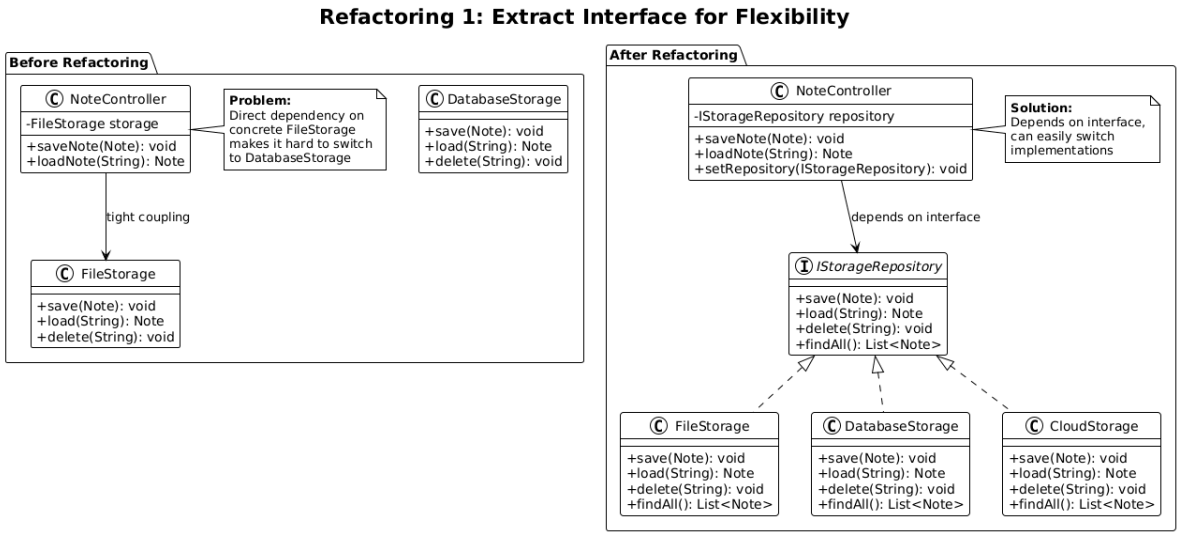

Figure 3: Refactoring 1: Extract Interface - Before (tight coupling) and After (interface-based design)

Before Refactoring: The controller directly depends on `FileStorage`, creating a tight coupling. Adding support for `DatabaseStorage` would require modifying the controller.
After Refactoring: The controller depends on the `IStorageRepository` interface. Multiple implementations can exist (FileStorage, DatabaseStorage, CloudStorage), and the controller can work with any of them. The specific implementation is provided through dependency injection, allowing the storage mechanism to be configured at runtime without changing controller code.
Benefits:

- The system can easily switch between storage implementations
- New storage types can be added without modifying existing code
- Controllers can be tested with mock storage implementations
- The system supports multiple storage backends simultaneously

2.2 Refactoring 2: Extract Class for Reuse

2.2.1 Problem Description

Validation and input sanitization logic is duplicated across multiple controller classes. The `NoteController`, `SearchController`, and `CategoryController` each contain similar methods for validating input, checking length constraints, sanitizing user input, and handling validation errors.

This duplication creates several problems:

- **Code Duplication:** The same validation logic is repeated in multiple places, violating the DRY (Don't Repeat Yourself) principle.
- **Inconsistent Behavior:** Different controllers may implement validation slightly differently, leading to inconsistent user experience and potential security issues.
- **Maintenance Burden:** When validation rules change (e.g., new requirements for password strength or content length), developers must update code in multiple locations, increasing the risk of errors and inconsistencies.
- **Difficulty Testing:** Validation logic scattered across controllers makes it harder to write comprehensive test suites for validation rules.
- **Lack of Reusability:** New controllers or features that need validation must reimplement the same logic rather than reusing existing code.

2.2.2 Refactoring Technique: Extract Class

The solution is to apply the **Extract Class** refactoring technique. This involves:

1. Creating a new `NoteValidator` class that centralizes all validation logic
2. Moving validation methods from controllers to the validator class
3. Creating a `ValidationResult` class to provide structured feedback about validation outcomes
4. Updating all controllers to use the shared validator instance
5. Ensuring the validator provides methods for all validation scenarios (title, content, tag names, search queries)

This refactoring follows the Single Responsibility Principle by giving the validator class one clear responsibility: validating input according to business rules.

2.2.3 UML Diagram: Before and After

Figure 4 shows the system before and after the refactoring.

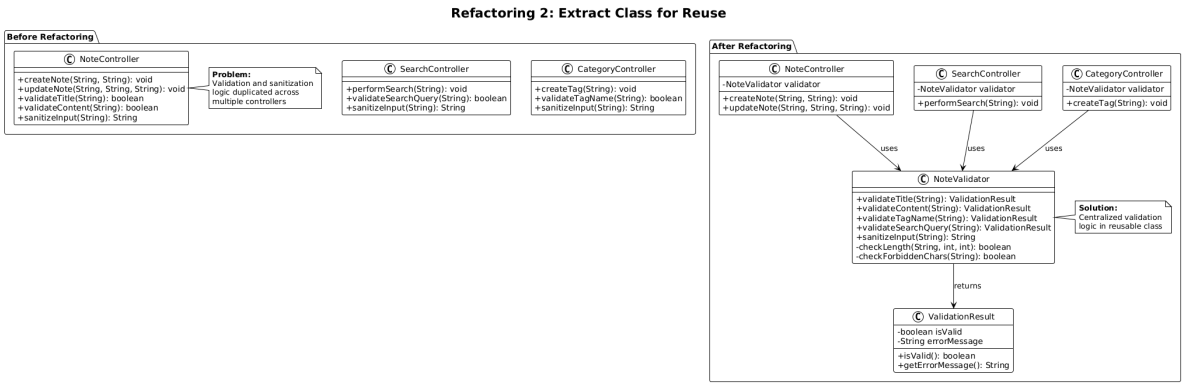

Figure 4: Refactoring 2: Extract Class - Before (duplicated validation) and After (centralized validator)

Before Refactoring: Each controller contains its own validation methods (`validateTitle`, `validateContent`, `sanitizeInput`, etc.), leading to code duplication and potential inconsistencies.

After Refactoring: All validation logic is centralized in the `NoteValidator` class. Controllers depend on the validator and delegate validation to it. The `ValidationResult` class provides a structured way to return validation outcomes, including success status and error messages.

Benefits:

- Validation logic is defined in one place, making it easier to maintain and update
- Consistent validation behavior across all controllers
- Easier to test validation rules in isolation
- New controllers can reuse existing validation logic
- Validation rules can be extended without modifying controller code

2.3 Refactoring 3: Extract Service Layer for Testability

2.3.1 Problem Description

Business logic is mixed with controller coordination logic, making the system difficult to test. The `NoteController` contains both coordination code (handling actions, updating views) and business logic (word count, formatting, checking for duplicate titles).
This mixing of concerns creates several problems:

- **Poor Testability:** To test business logic, test code must instantiate controllers and views, making unit tests complex and slow. Business logic cannot be tested independently of UI components.
- **Tight Coupling:** Business logic is coupled to controller and view code, making it difficult to reuse business operations in different contexts (e.g., batch processing, API endpoints, background jobs).
- **Violation of Single Responsibility:** Controllers have multiple responsibilities: coordinating user interactions, managing business logic, and updating views. This makes controllers large and difficult to understand.
- **Difficulty Reusing Logic:** Business operations like word count calculation or content formatting cannot be easily reused by other parts of the system without going through controllers.
- **Slower Development:** Changes to business logic require understanding and potentially modifying controller code, slowing down development and increasing the risk of introducing bugs.

2.3.2 Refactoring Technique: Extract Service Layer

The solution is to apply the **Extract Service Layer** refactoring technique (also known as introducing a Service Layer pattern). This involves:

1. Creating a `NoteService` class that contains all business logic operations
2. Moving business logic methods from the controller to the service (word count, formatting, duplicate checking, etc.)
3. Making the service depend on the repository for data access
4. Updating the controller to delegate business operations to the service
5. Ensuring the controller becomes a thin coordination layer that only handles user interaction flow

This refactoring follows the Separation of Concerns principle by clearly separating business logic from presentation and coordination logic.

2.3.3 UML Diagram: Before and After

Figure 5 shows the system before and after the refactoring.

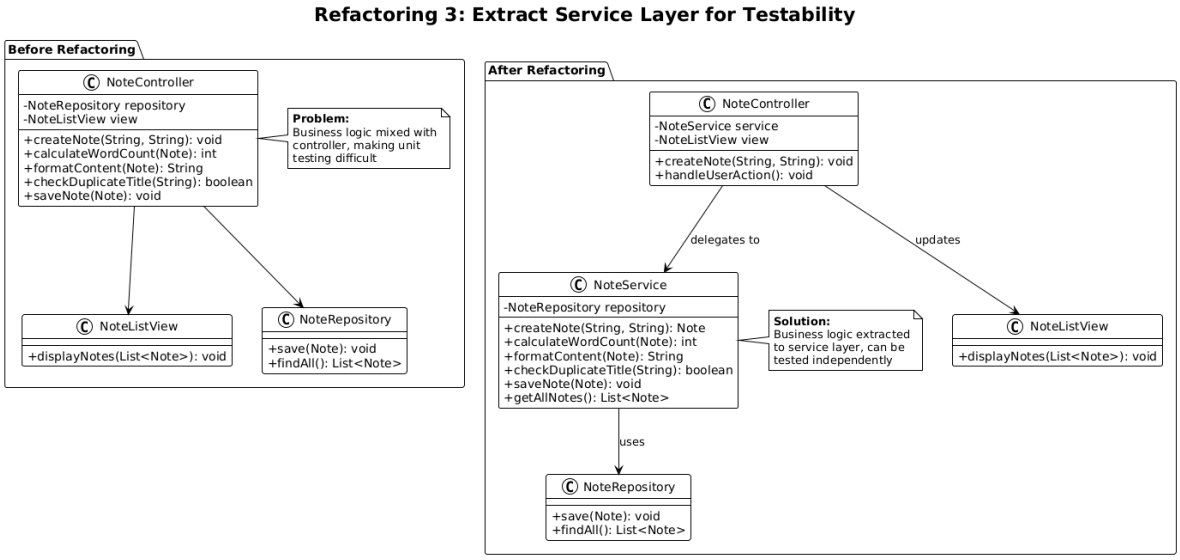

Figure 5: Refactoring 3: Extract Service Layer - Before (mixed concerns) and After (separated service layer)

Before Refactoring: The controller contains both coordination logic and business logic. Methods like `calculateWordCount`, `formatContent`, and `checkDuplicateTitle` are mixed with view update code, making it difficult to test business logic independently.

After Refactoring: Business logic is extracted to the `NoteService` layer. The service depends on the repository for data access and provides business operations that can be used by controllers or other parts of the system. The controller becomes a thin layer that coordinates user actions and delegates business operations to the service.

Benefits:

- Business logic can be unit tested independently of controllers and views
- Business operations can be reused by other components (APIs, background jobs, etc.)
- Controllers are simpler and easier to understand
- Business logic changes do not require modifying controller or view code
- The system can support multiple interfaces (web, mobile, API) that all use the same service layer
- Easier to apply business rules consistently across the application

3 Part C: Tool Use and Reflection

3.1 Refactoring Tool: Visual Studio Code

Visual Studio Code provides built-in refactoring capabilities that help with the refactorings described in this assignment.

3.1.1 Key Refactoring Features

VS Code provides several essential refactoring operations:
Rename Symbol: Automatically finds and updates all references when renaming classes, methods, or variables. Essential for all three refactorings when extracting classes or interfaces.
Extract Method: Extracts selected code blocks into new methods, automatically identifying parameters and return types. Useful for Refactoring 2 (Extract Class) when moving methods to the validator class.
Move Symbol: Moves classes, methods, or functions to different files, automatically updating import statements. Directly supports Refactoring 3 (Extract Service Layer) when moving business logic to service classes.
Extract Variable: Introduces variables for complex expressions, improving code readability during refactoring preparation.
Quick Fix and Code Actions: Provides context-aware suggestions for refactoring opportunities, such as extracting interfaces or creating new classes.

3.1.2 How VS Code Helps with Refactoring

VS Code's refactoring tools automatically update references across files, provide change previews, work across the entire workspace, and integrate with version control. These features make refactoring safer and more efficient than manual code changes.

3.2 Reflection

3.2.1 Which Refactor Would Have the Biggest Impact?

After analyzing the three refactoring scenarios, I believe that **Refactoring 3: Extract Service Layer for Testability** would have the biggest impact on the system's long-term maintainability and development velocity.

This refactoring addresses the fundamental problem of mixed concerns by separating business logic from controller coordination logic. By extracting business logic into a service layer, the system gains several critical advantages: business logic can be unit tested independently without requiring controllers or views, the same business operations can be reused across different interfaces (web, mobile, API), and the codebase becomes easier to understand and navigate. This separation also creates a foundation that makes other refactorings easier to apply, as the code is better organized with clear responsibilities.

While Refactoring 1 (Extract Interface) improves flexibility and Refactoring 2 (Extract Class) improves code reuse, Refactoring 3 provides the architectural foundation that enables both of these improvements and many others.

3.2.2 How Would These Changes Affect Future Development?

When applied together, these three refactorings would significantly improve the system's ability to evolve and adapt to changing requirements:

Refactoring 1 (Extract Interface) enables the system to adopt new storage technologies without major rewrites, supports multiple storage backends simultaneously, and allows for easier testing with mock implementations.

Refactoring 2 (Extract Class) ensures consistent validation behavior across the application, speeds up feature development by reusing validation logic, and makes it easier to update validation rules in one place when requirements change.

Refactoring 3 (Extract Service Layer) enables support for multiple interfaces (web, mobile, API) sharing the same business logic, allows frontend and backend teams to work in parallel, and makes business logic changes independent of UI code.

Together, these refactorings create a system that is more maintainable, testable, extensible, and flexible. The clear separation of concerns makes it easier to locate and modify code, each layer can be tested independently, and new features can be added without modifying existing code. This architectural foundation supports long-term development efforts and makes the system resilient to changes in requirements or technology.

References

1. Kung, David C. (2013). *Software Engineering: An Object-Oriented Perspective* (2nd ed.). Course textbook chapters 16.6, 16.7, 17.1, 17.2, and 17.3.