## CSS-499 Milestone One Narrative

## Introduction (2-3 minutes)

Hi, my name is Michelle Eason, and this is my Computer Science Capstone code review. In this video, I’ll walk through three artifacts from my previous coursework to analyze their current functionality, identify areas for improvement, and share my enhancement plans. We’ll cover three categories: Software Engineering and Design, Algorithms and Data Structures, and Databases. Each section will include an overview of what the code does now, a critical analysis based on the Code Review Checklist, and a clear plan for enhancements that align with course outcomes and my career goals. Let’s get started.

## Category One: Software Engineering and Design (8-10 minutes)

I. Existing Code Functionality

The first artifact I’m reviewing comes from my Reverse Software Engineering course. This project is a C++ application that simulates basic user interaction and access control through a simple console interface. Users can log in using a hardcoded username and password, navigate a text-based menu, and perform a few placeholder actions like viewing a welcome message or exiting the program. The code is organized into a main.cpp file and a few helper functions, but all authentication logic is implemented inline. There is no actual database or secure credential storage, everything is defined directly in the code. Once authenticated, the user sees a menu where they can select numbered options, but there is no persistence of data or meaningful operations beyond simple console output. Overall, the code meets the basic assignment requirements, but it is not ready for real-world use.

II. Analysis of Existing Code

Next, I’ll analyze this code using the Code Review Checklist categories:

1. Structure:  
 - The authentication function is too large and mixes input, validation, and output. This violates the single responsibility principle.  
 - There are no separate modules or classes for different functionalities. Everything is in main or in minimal helper functions.  
 - Repeated code exists in the menu loop, such as printing the menu options and reading input, which could be refactored into a utility function.

2. Documentation:  
 - Comments are minimal. There are none explaining the purpose of each function, the expected input, or the output.  
 - The code lacks a header comment that summarizes the file’s functionality and version history.

3. Variables:  
 - Variable names like ‘u’ for username and ‘p’ for password are not descriptive.  
 - Magic strings are used directly in the code, such as “admin” and “password123,” instead of symbolic constants.

4. Defensive Programming and Security:  
 - Credentials are hardcoded, making the program insecure and inflexible.  
 - No input validation is performed, allowing empty or malicious inputs to pass through.  
 - There is no hashing of passwords; the plain text password comparison is vulnerable.

5. Error Handling:  
 - If a user enters an invalid option at the menu, the program simply exits or loops without user-friendly feedback.  
 - There is no try-catch or error-checking around critical operations like reading input.

These issues reduce maintainability, security, and scalability.

III. Planned Enhancements

To address these weaknesses, I will implement the following enhancements:  
1. Secure Authentication:  
 - Replace hardcoded credentials with a secure login system that uses a configuration file or database to store user records.  
 - Hash passwords with a strong algorithm like SHA-256 and compare hashed values instead of plain text.

2. Input Validation and Error Handling:  
 - Validate username and password inputs to ensure they meet length and character requirements.  
 - Add exception handling around input operations to catch and recover from bad inputs.

3. Refactoring and Modularity:  
 - Extract authentication, menu handling, and business logic into separate classes and modules.  
 - Create a `UserAuthenticator` class, a `MenuHandler` class, and utility functions to print menus and read choices.

4. Logging and Documentation:  
 - Add logging statements for successful and failed login attempts.  
 - Document each class and function with header comments and inline notes.

These changes will demonstrate my skills in secure coding, modular design, and error management. They align with the course outcomes of developing a security mindset and designing computing solutions that balance performance and protection.

## Category Two: Algorithms and Data Structures (8-10 minutes)

I. Existing Code Functionality

The second artifact is my Software Testing, Automation, and Quality Assurance project—a contact management system written in C++. It provides basic CRUD operations: users can add, view, update, and delete contacts, appointments, and tasks. Under the hood, it uses arrays and simple linked lists to store items in memory. Unit tests written in JUnit verify methods in classes like `ContactService`, `AppointmentService`, and `TaskService`. These tests check normal operations and exception cases, such as adding a duplicate contact or deleting a non-existent appointment. While the tests offer good coverage, the underlying data structures are not optimized for large data sets.

II. Analysis of Existing Code

1. Efficiency and Data Structures:  
 - The code uses linear searches to find contacts by ID, which is O(n) time complexity.  
 - Arrays have a fixed size, requiring manual resizing or risk overflow when the dataset grows.  
 - Linked list traversal also takes O(n) time for searches.

2. Structure and Modularity:  
 - Service classes contain both business logic and data structure management, violating separation of concerns.  
 - Benchmarking or performance measurement code is absent, so efficiency cannot be quantified.

3. Documentation and Comments:  
 - Methods lack inline comments describing edge cases and algorithm choices.  
 - Test cases are documented, but implementation code lacks context.

4. Error Handling and Defensive Programming:  
 - Input validation for methods that accept IDs or descriptions is minimal.  
 - There is no handling for memory allocation failures in dynamic structures.

These issues limit the system’s scalability and make it difficult to trust its performance under heavy loads.

III. Planned Enhancements

To improve performance and demonstrate advanced algorithmic skills, I will implement the following:  
1. Advanced Data Structures:  
 - Replace arrays and linked lists with a `std::unordered\_map` (hash map) keyed by ID for O(1) lookups.  
 - Use `std::vector` for dynamic arrays that resize automatically and offer cache-friendly storage.

2. Algorithmic Optimization:  
 - Refactor search and update methods to leverage hash map operations.  
 - Remove manual loops for common operations by using standard library algorithms like `std::find\_if`.

3. Performance Benchmarking:  
 - Integrate timing code using `<chrono>` to measure execution time of operations for datasets of different sizes.  
 - Output performance metrics to a log or report file for analysis.

4. Code Refactoring and Comments:  
 - Separate business logic from data management in service classes.  
 - Document algorithm choices and complexity analysis inline with code.

These enhancements will showcase my ability to optimize algorithms and use modern C++ data structures. They align with course outcomes related to designing efficient computing solutions and employing innovative techniques.”

## Category Three: Databases (8-10 minutes)

I. Existing Code Functionality

The third artifact is the database component of the same contact management system. It connects to a MySQL database to store and retrieve contacts, appointments, and tasks. The code uses simple SQL statements embedded in C++ methods. For example, the `addContact` method constructs an `INSERT` query by concatenating strings and executes it via the database connector. Retrieved results are mapped into object instances, and deletion and update operations follow a similar pattern.

II. Analysis of Existing Code

1. Security:  
 - SQL statements are built via string concatenation, which is vulnerable to SQL injection.  
 - No encryption is used for sensitive fields like passwords or personal data.

2. Database Design and Structure:  
 - The schema is not normalized, leading to redundant storage of user details across multiple tables.  
 - There is no indexing strategy beyond primary keys, which can slow down queries as data grows.

3. Error Handling:  
 - The code does not check for SQL execution failures or handle exceptions thrown by the database API.  
 - Connection failures or network issues are not considered.

4. Access Control:  
 - All users have equal privileges; there is no role-based access control to restrict sensitive actions.

III. Planned Enhancements

To secure and optimize the database layer, I will implement:  
1. Parameterized Queries:  
 - Replace string concatenation with prepared statements (`?` placeholders) to prevent SQL injection.

2. Encryption:  
 - Encrypt sensitive columns like passwords or personal data using a strong algorithm before storage.  
 - Decrypt data when loading it for authorized operations.

3. Normalization and Indexing:  
 - Refactor the database schema to separate user information from contact details and tasks.  
 - Add indexes on frequently searched fields to speed up queries.

4. Role-Based Access Control:  
 - Introduce user roles in the database schema and enforce permissions in code.

5. Error Handling:  
 - Wrap database calls in try-catch blocks and log errors with descriptive messages.

These enhancements will demonstrate my skills in secure database programming and efficient database design. They align with the course outcomes of developing a security mindset and designing high-quality computing solutions.

## Closing Summary (2-3 minutes)

To wrap up, I’ve reviewed three artifacts covering software engineering and design, algorithms and data structures, and databases. For each, I described the existing functionality, analyzed weaknesses using the Code Review Checklist, and proposed targeted enhancements. These improvements—secure authentication, modular design, optimized data structures, performance benchmarking, parameterized queries, and more—will showcase my ability to deliver professional-quality, secure, and efficient solutions. Thank you for watching my code review, and I look forward to implementing these enhancements in my ePortfolio.