Student Management System

Team Members :

1- *Eman Hassan*
2- *Eyad Zakaria*
3- *Arwa Thabet*
4- *Demiana Adel*
5- *Ezzeldin Mostafa*
6- *Amira Mohammed*

Overview

The Student Management System is a Java-based application designed to manage students and administrators in an educational context. It allows users to perform CRUD operations and includes features for user authentication, registration, and role management. The application is integrated with a MySQL database and demonstrates the application of several design patterns.

Features

1. **User Authentication: Login and register functionality for students and admins.**
2. **Add Entities: Add new students or admins to the system.**
3. **View Entities: Display the list of students or admins.**

4. **Operations on Entities: Perform the following operations:**
    - **Search**
    - **Update**
    - **Delete**
    - **Submit**

Technologies Used

- **Programming Language: Java**
- **Database: MySQL**
- **Design Patterns Applied:**
    - **Singleton**
    - **Proxy**
    - **Builder**
    - **Prototype**
    - **Factory**

1. **We Used Singleton Pattern on Database Connection Class to : Ensures Single Database Connection Instance**
    - **A database connection is a costly resource. Creating multiple connections unnecessarily can:**
        - **Overwhelm the database server.**
        - **Cause resource exhaustion.**
        - **Reduce application performance.**
    - **The Singleton Pattern ensures that only one connection is created and reused throughout the application.**

Benefit: Reduces resource consumption and ensures efficient use of the database connection.

Global Access Point

- **The Singleton provides a global access point to the database connection through the getInstance() method.**
- **Any class in the application that needs the database connection can call:**

Benefit: Simplifies access to the database connection without duplicating logic.

Thread-Safe Initialization

- **Your implementation uses double-checked locking with synchronized, which ensures:**
  - **Thread safety during the creation of the DatabaseConnection instance.**
  - **Improved performance by avoiding unnecessary synchronization after initialization.**

Benefit: Prevents multiple threads from creating duplicate instances in a multi-threaded environment.

Prevents Connection Leaks

- **With a single instance, the risk of connection leaks (e.g., forgetting to close unused connections) is reduced.**

- **Since the connection is managed centrally, developers don’t accidentally open multiple connections without closing them.**

Benefit: Ensures a reliable and manageable database connection lifecycle.

Centralized Error Handling

- **The Singleton constructor includes error handling for:**
  - **Loading the JDBC driver.**
  - **Connecting to the database.**
- **If any error occurs, a message is displayed to the user via JOptionPane.**
- **This centralizes error handling logic rather than duplicating it across multiple parts of the code.**

Benefit: Reduces code duplication and centralizes error management.

Improved Maintainability

- **By using a Singleton for database connections:**
  - **You can easily update connection parameters (e.g., URL, user, password) in one place.**
  - **Any future enhancements (e.g., logging, caching) to the database connection logic can be done in the DatabaseConnection class without changing other parts of the application.**

Benefit: Simplifies maintenance and reduces the risk of introducing errors.

---

2. We Used Proxy Pattern on login page to :

Separation of Concerns

- **The login page (Login class) should only focus on the UI—accepting user inputs and displaying messages.**
- **By introducing the Proxy, the authentication logic is abstracted and moved outside the UI class.**
- **The AuthenticationProxy handles:**
  - **Input validation.**
  - **Security checks.**
  - **Logging.**
  - **Delegation to the AuthenticationService (Real Subject) for actual authentication.**

  Benefit: Cleaner, more maintainable code that separates UI logic from business logic.
- **Input Validation**

  Before accessing the database, the proxy checks if the email or password fields are empty or invalid.

  This avoids unnecessary database calls.

3. We Used Builder Pattern on student page to :

Clear Separation of Object Construction from Representation:

The Builder pattern allows for the creation of Student objects in a step-by-step manner, rather than having one large constructor that takes all parameters. This leads to better separation of concerns

where the object construction logic is separated from the business logic.

Without the builder, if the Student class required multiple parameters (such as 7 fields in your case), the constructor would be cumbersome to manage and could lead to confusion. With the builder, you can easily handle each field separately and ensure that all required fields are set before the object is created.

Readability and Maintainability:

Fluent Interface: The builder provides a fluent interface. This allows you to chain method calls (e.g., .setId(id).setFirstName(firstName)...) in a clean and readable way. This makes the code more intuitive and easier to follow.
Naming: The methods in the builder (setId(), setFirstName(), etc.) clearly indicate what each field represents, making the code more self-explanatory.

Encapsulation:

The builder helps encapsulate the creation process of the Student object, protecting the Student class from being overloaded with multiple constructors or invalid states.

For instance, if the Student class only had a constructor that required all fields at once, it would be difficult to create a Student with default or null values for certain attributes. The builder pattern allows you to create Student objects step by step, ensuring that all fields are appropriately set without exposing the fields directly to the outside world.

Avoiding Constructor Overload:

If you had a constructor that accepts all parameters for the Student class, the number of constructors would increase if you needed to support variations (e.g., default or null values for some fields). This leads to code duplication and errors.

Using the builder pattern, you only need one constructor (the private one), and the builder can be used to handle different combinations of parameters.
Improved Flexibility and Scalability:

The builder pattern provides flexibility in object creation. For example, if you want to add new fields to the Student class later (e.g., address), you don't need to update the constructor and all calls to it. You can simply add a method to the builder (e.g., setAddress()) without disrupting the rest of the code.
It also makes your code more scalable when dealing with complex objects that may have a large number of fields or optional attributes.

4. We Used Prototype Pattern showStudent Page to :

Avoid Redundant Object Creation:

- Cloning an object rather than creating a new one can be more efficient in situations where creating a new instance requires more resources. Here, you could have a base template (`SShowStudent` instance) and use its clone to quickly populate your table without needing to recreate every single data field.
- The cloning helps in creating a new `SShowStudent` object based on the original, which avoids manually copying all data fields for each row.

Improved Performance:

- If the `SShowStudent` objects are complex or contain nested objects, cloning can be faster than re-creating new instances. For example, if a student record had multiple dependent objects (e.g., addresses, courses, etc.), cloning would allow you to reuse those dependent objects.

Simplifying Data Handling:

- By using the Prototype pattern, you can avoid manually constructing a new student object for every single row. The pattern abstracts away the creation of new objects by relying on the prototype and simply cloning it. This simplifies the code, as you're dealing with a single instance of the object structure rather than manually managing its construction each time.

Consistency:

- Since the cloned object is based on a prototype that has already been populated with valid data (from the database), the objects created in the table will have the same structure and integrity as the original prototype. This ensures consistency in how each `SShowStudent` is constructed.

5. **We Used factory pattern addStudent page : Separation of Object Creation and Business Logic**

- **Encapsulation of Object Creation: The Factory Pattern encapsulates the object creation logic, separating the instantiation of the AdddStudent object from the rest of the application logic (like the addStudent GUI class). This makes the code cleaner and easier to maintain. In this case, the creation of a Student object is handled by the ConcreteStudentFactory class, not directly in the addStudent GUI form. This ensures that the form doesn't**

need to worry about the construction details of the AdddStudent object, improving modularity.

- **Code Reusability: With the factory handling the creation of AdddStudent, if there are any changes to how AdddStudent objects should be created (e.g., new validation rules, different initialization steps), the changes are centralized in the factory class. This minimizes code duplication and improves maintainability.**

2. Flexibility and Extensibility

- **Easier to Modify and Extend: If you need to create different types of students (e.g., undergraduate vs graduate students, or students from different departments), you can extend the factory to support multiple types of student objects without modifying the addStudent class. You could create different factories for different student types and keep the addStudent GUI logic unchanged.**
- **Single Responsibility: The ConcreteStudentFactory class is responsible only for creating student objects, and does not involve itself with GUI-related logic or database interaction. This adheres to the Single Responsibility Principle (SRP) and makes the system easier to extend, e.g., if a new type of student needs to be added, only the factory needs to change.**