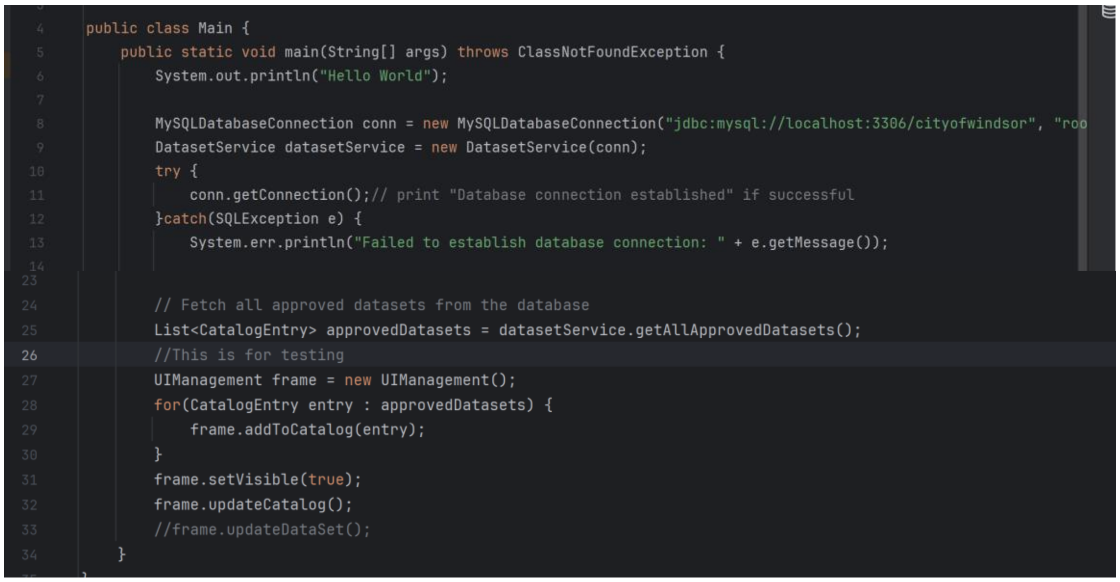
GRASP PATTERNS USED IN CODE

Pattern : Controller
Class: Main

- The purpose of this pattern is to handle the initial system operations and delegates responsibility to other classes (UIManagement, CatalogueEntry), ensuring that the system's workflow is managed efficiently. It is the main entry point to the use case or system[1].
  - It is to be the first object beyond the UI layer that is responsible for handling a system operation message (Chapter 17.13).
  - It should deal with user input, making decisions or "routing" the responsibilities, serve as an intermediary and not handle the current responsibilities directly with unrelated sections of operation.

- In our Code (File: Main.java), the Main Class is the first entry object of the application and coordinates the setup of various components such as the database connection, dataset service, and UI elements.
  - Line 8: Initializes the database connection, needed to access data.
  - Line 9: Initializes the Dataset Service.
  - Line 25: Fetches all approved datasets from the database and delegates task of updating UI to UIManagement class.



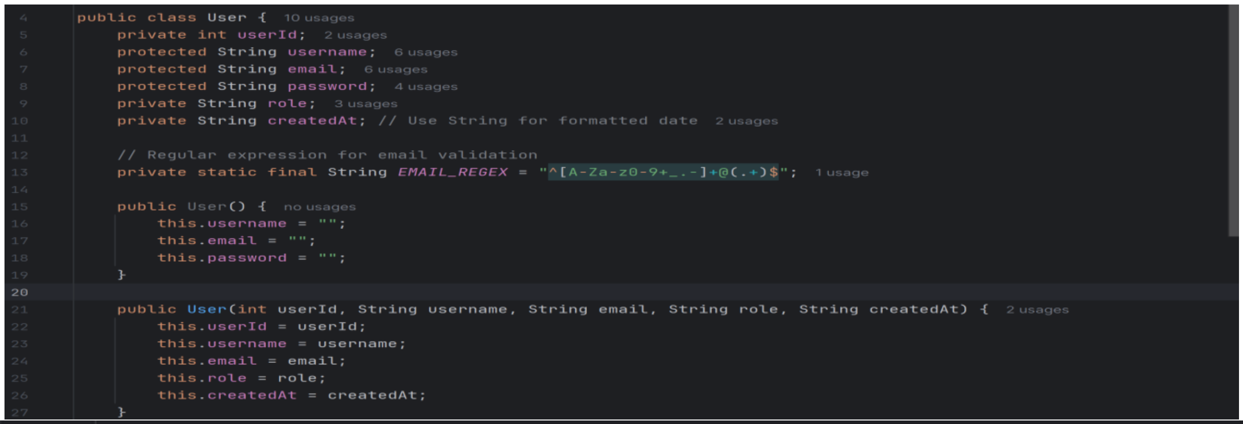
```
4   public class Main {
5       public static void main(String[] args) throws ClassNotFoundException {
6           System.out.println("Hello World");
7
8           MySQLDatabaseConnection conn = new MySQLDatabaseConnection("jdbc:mysql://localhost:3306/cityofwindsor", "roo
9           DatasetService datasetService = new DatasetService(conn);
10          try {
11              conn.getConnection();// print "Database connection established" if successful
12          }catch(SQLException e) {
13              System.err.println("Failed to establish database connection: " + e.getMessage());
14
23
24          // Fetch all approved datasets from the database
25          List<CatalogEntry> approvedDatasets = datasetService.getAllApprovedDatasets();
26          //This is for testing
27          UIManagement frame = new UIManagement();
28          for(CatalogEntry entry : approvedDatasets) {
29              frame.addToCatalog(entry);
30          }
31          frame.setVisible(true);
32          frame.updateCatalog();
33          //frame.updateDataSet();
34      }
```

(Fig 1.0 Main Class as Controller)

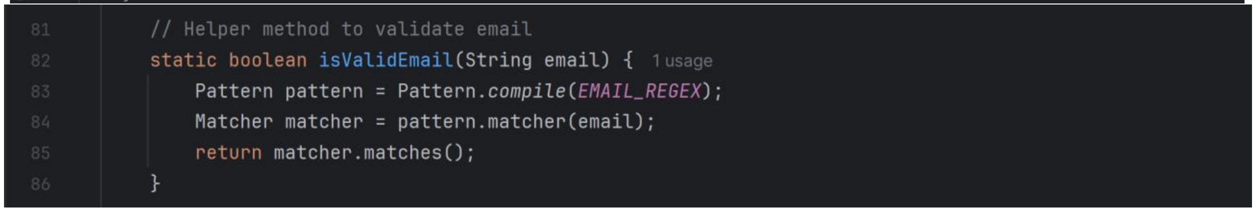
[1] https://dev.to/mgce/do-you-know-grasp-part-1-controller-and-creator-46bg

Pattern: Information Expert
Class: User

- The purpose of this pattern is to fulfill the responsibility that requires information spread across different classes. It starts assigning responsibilities to other objects by clearly stating the responsibility. It ensures high cohesion and low coupling, making it easier to manage and maintain.
- In our Code (File: User.java), the User Class encapsulates all the information related to a user, such as username, email, password and role. It also includes methods for validating email addresses and managing user-specific data. By centralizing user-related information and behaviors, it reduces dependencies between classes. It is the parent class to other subclasses such as the regular user and the Administrator class, which are variations of the user with different responsibilities.
    - Lines 4-10: encapsulates all information related to user.
    - Lines 21-27: sets all user attributes, ensuring the class manages user-specific data.
    - Lines 81-86: enforces data integrity by checking if email is valid.



```
4   public class User {  10 usages
5       private int userId;  2 usages
6       protected String username;  6 usages
7       protected String email;  6 usages
8       protected String password;  4 usages
9       private String role;  3 usages
10      private String createdAt; // Use String for formatted date  2 usages
11
12      // Regular expression for email validation
13      private static final String EMAIL_REGEX = "^[A-Za-z0-9+_.-]+@(.+)$";  1 usage
14
15      public User() {  no usages
16          this.username = "";
17          this.email = "";
18          this.password = "";
19      }
20
21      public User(int userId, String username, String email, String role, String createdAt) {  2 usages
22          this.userId = userId;
23          this.username = username;
24          this.email = email;
25          this.role = role;
26          this.createdAt = createdAt;
27      }
```

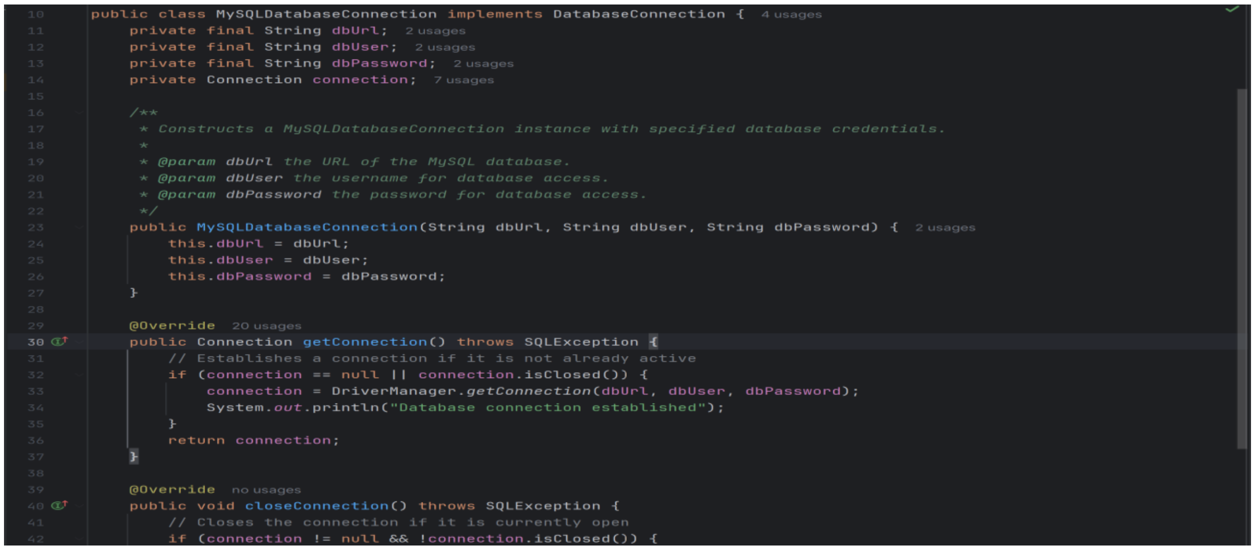


```
81      // Helper method to validate email
82      static boolean isValidEmail(String email) {  1 usage
83          Pattern pattern = Pattern.compile(EMAIL_REGEX);
84          Matcher matcher = pattern.matcher(email);
85          return matcher.matches();
86      }
```

(Fig 2.0 User Class as Information Expert)

Pattern: Low Coupling
Class: MySQLDatabaseConnection

- The purpose of this pattern is to minimize dependencies between the database connection logic and other parts of the application, making the system more modular and easier to maintain since this class doesn't depend on other classes. It cannot be considered in isolation from other patterns such as expert and high cohesion.
- In our Code (File: MySQLDatabaseConnection.java), the Class encapsulates database connection responsibilites and provides a clear interface and methods such as getConnection and closeConnection to manage connections without exposing underlying implementation details.
    - Lines 10-14: ensures database and connection details are encapsulated within the MySQLDatabaseConnection class.
    - Lines 23-27: ensures the class has information to manage database connections without depending on other classes.
    - Lines 30&40: the getConnection and closeConnection methods are called by other classes to get databse connections without going through the whole class logic.



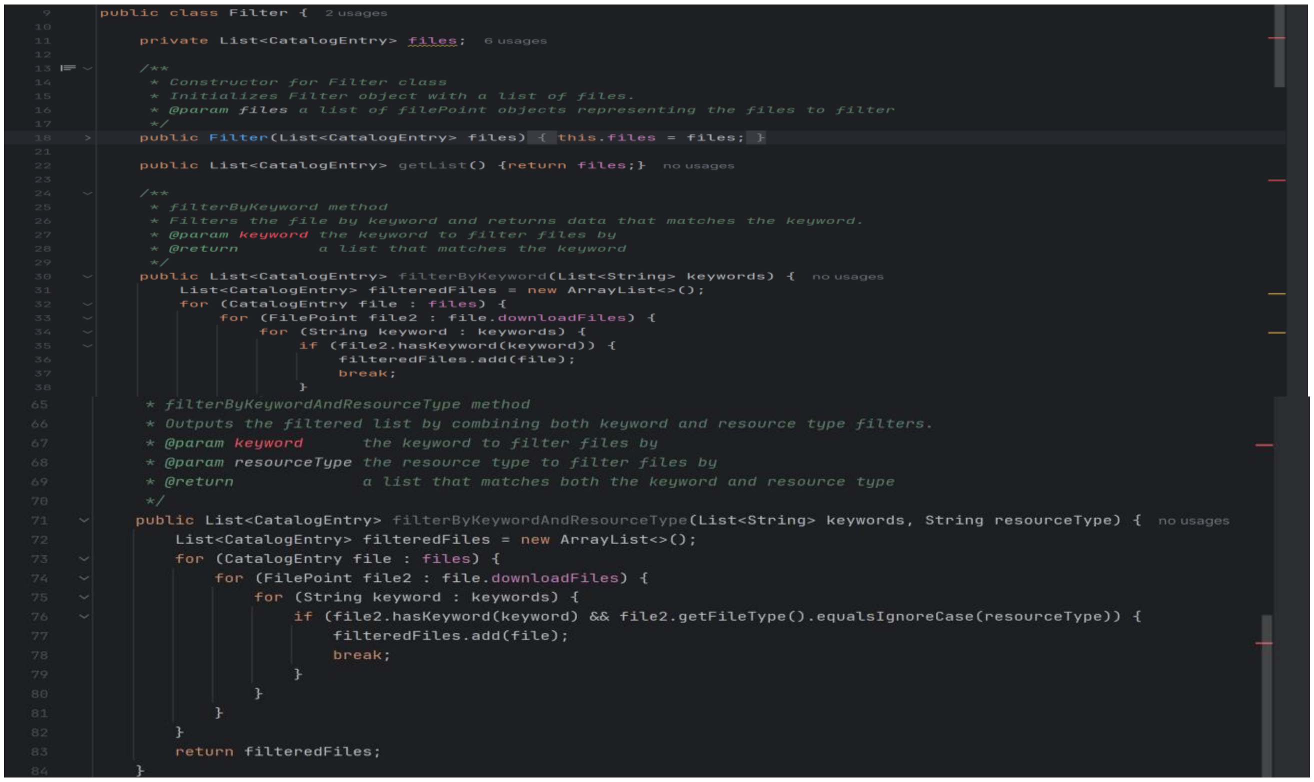
```
10  public class MySQLDatabaseConnection implements DatabaseConnection {  4 usages
11      private final String dbUrl;  2 usages
12      private final String dbUser;  2 usages
13      private final String dbPassword;  2 usages
14      private Connection connection;  7 usages
15
16      /**
17       * Constructs a MySQLDatabaseConnection instance with specified database credentials.
18       *
19       * @param dbUrl the URL of the MySQL database.
20       * @param dbUser the username for database access.
21       * @param dbPassword the password for database access.
22       */
23      public MySQLDatabaseConnection(String dbUrl, String dbUser, String dbPassword) {  2 usages
24          this.dbUrl = dbUrl;
25          this.dbUser = dbUser;
26          this.dbPassword = dbPassword;
27      }
28
29      @Override  20 usages
30      public Connection getConnection() throws SQLException {
31          // Establishes a connection if it is not already active
32          if (connection == null || connection.isClosed()) {
33              connection = DriverManager.getConnection(dbUrl, dbUser, dbPassword);
34              System.out.println("Database connection established");
35          }
36          return connection;
37      }
38
39      @Override  no usages
40      public void closeConnection() throws SQLException {
41          // Closes the connection if it is currently open
42          if (connection != null && !connection.isClosed()) {
```

(Fig 3.0 MySQLDatabaseConnection Class as Low Coupling)

Pattern: High Cohesion
Class: Filter

- The purpose of this pattern is to ensure a class focuses solely on single well-defined operation. It should demonstrate high cohesion, be easily understandable, manageable, testable, and avoid unrelated operations.

- In our code (File: Filter.java) the Filter class provides methods to filter datasets based on keywords, resource types, and combinations of both. According to the High Cohesion pattern, by encapsulating all filtering logic within a single class, the Filter class ensures that it is focused and all methods are highly related to a single responsibility, making the code easier to understand and maintain.
    - Line 18: public Filter(List<CatalogEntry> files) {this.files = files;} initializes the filter class with a list of CatalogEntry objects and ensures that the class only deals with filtering methods on datasets.



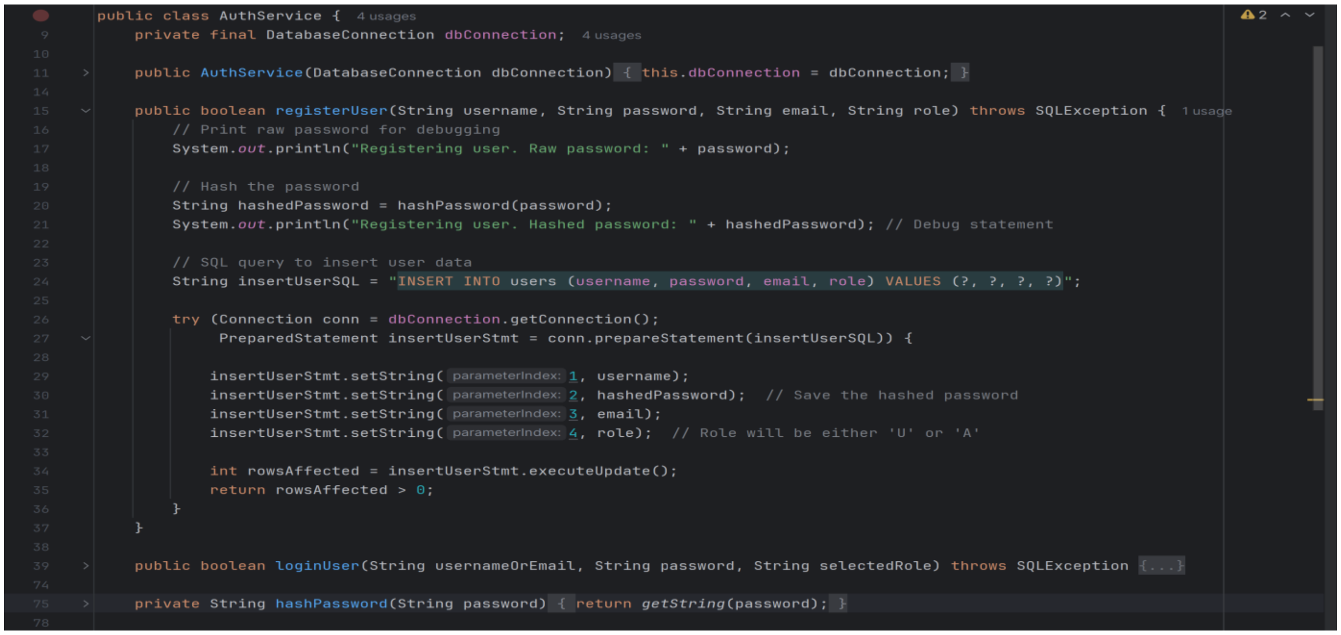
```
9   public class Filter {  2 usages
10
11      private List<CatalogEntry> files;  6 usages
12
13      /**
14       * Constructor for Filter class
15       * Initializes Filter object with a list of files.
16       * @param files a list of filePoint objects representing the files to filter
17       */
18      public Filter(List<CatalogEntry> files) { this.files = files; }
21
22      public List<CatalogEntry> getList() {return files;}  no usages
23
24      /**
25       * filterByKeyword method
26       * Filters the file by keyword and returns data that matches the keyword.
27       * @param keyword the keyword to filter files by
28       * @return         a list that matches the keyword
29       */
30      public List<CatalogEntry> filterByKeyword(List<String> keywords) {  no usages
31          List<CatalogEntry> filteredFiles = new ArrayList<>();
32          for (CatalogEntry file : files) {
33              for (FilePoint file2 : file.downloadFiles) {
34                  for (String keyword : keywords) {
35                      if (file2.hasKeyword(keyword)) {
36                          filteredFiles.add(file);
37                          break;
38                      }
65       * filterByKeywordAndResourceType method
66       * Outputs the filtered list by combining both keyword and resource type filters.
67       * @param keyword      the keyword to filter files by
68       * @param resourceType the resource type to filter files by
69       * @return             a list that matches both the keyword and resource type
70       */
71      public List<CatalogEntry> filterByKeywordAndResourceType(List<String> keywords, String resourceType) {  no usages
72          List<CatalogEntry> filteredFiles = new ArrayList<>();
73          for (CatalogEntry file : files) {
74              for (FilePoint file2 : file.downloadFiles) {
75                  for (String keyword : keywords) {
76                      if (file2.hasKeyword(keyword) && file2.getFileType().equalsIgnoreCase(resourceType)) {
77                          filteredFiles.add(file);
78                          break;
79                      }
80                  }
81              }
82          }
83          return filteredFiles;
84      }
```

(Fig 4.0 Filter Class as High Cohesion)

Pattern: Indirection
Class: AuthService

- The purpose of this pattern is to assign the responsibility of an intermediate object to mediate between other components or services so that they are not directly coupled, promoting flexibility and maintainability.
- In our code (File: AuthService.java), the AuthService class uses the Indirection pattern to manage user authentication. It acts as an intermediary between the user interface and the database, handling the logic for registering and logging in users. This separation allows for changes in authentication logic without affecting other parts of the system.

- Line 11: ensures related functionalities are encapsulated.
- Line 26: separates database connection logic from the AuthService class, allowing it to act as an intermediary.



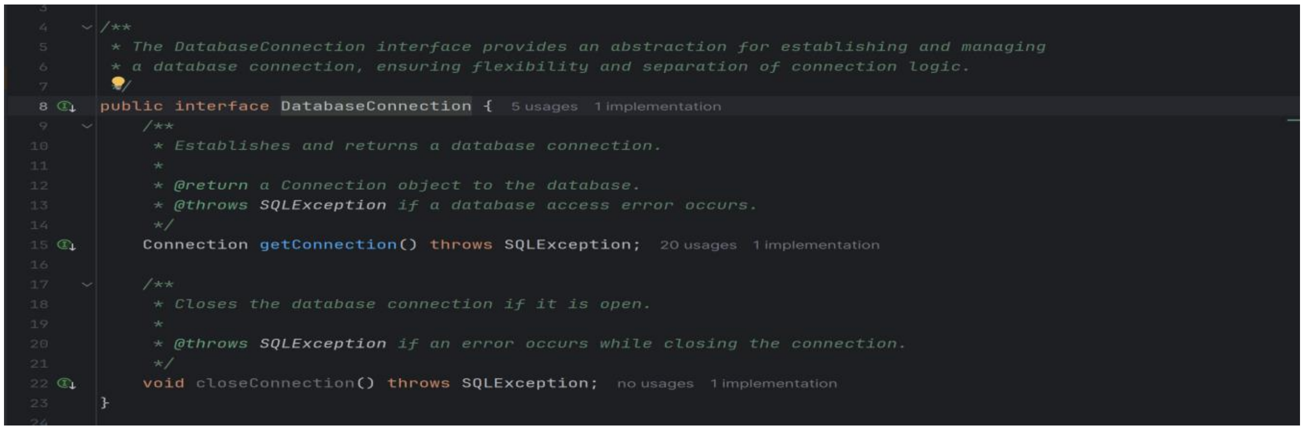
```
   public class AuthService {  4 usages
9      private final DatabaseConnection dbConnection;  4 usages
10
11 >   public AuthService(DatabaseConnection dbConnection) { this.dbConnection = dbConnection; }
14
15 v   public boolean registerUser(String username, String password, String email, String role) throws SQLException {  1 usage
16         // Print raw password for debugging
17         System.out.println("Registering user. Raw password: " + password);
18
19         // Hash the password
20         String hashedPassword = hashPassword(password);
21         System.out.println("Registering user. Hashed password: " + hashedPassword); // Debug statement
22
23         // SQL query to insert user data
24         String insertUserSQL = "INSERT INTO users (username, password, email, role) VALUES (?, ?, ?, ?)";
25
26         try (Connection conn = dbConnection.getConnection();
27 v            PreparedStatement insertUserStmt = conn.prepareStatement(insertUserSQL)) {
28
29             insertUserStmt.setString( parameterIndex: 1, username);
30             insertUserStmt.setString( parameterIndex: 2, hashedPassword);  // Save the hashed password
31             insertUserStmt.setString( parameterIndex: 3, email);
32             insertUserStmt.setString( parameterIndex: 4, role);  // Role will be either 'U' or 'A'
33
34             int rowsAffected = insertUserStmt.executeUpdate();
35             return rowsAffected > 0;
36         }
37     }
38
39 >   public boolean loginUser(String usernameOrEmail, String password, String selectedRole) throws SQLException {...}
74
75 >   private String hashPassword(String password) { return getString(password); }
78
```

(Fig 5.0 AuthService Class as Indirection)

Pattern: Polymorphism

Interface: DatabaseConnection

- The purpose of this pattern is to assign responsibility for behaviour using polymorphic operations, defining a common interface for different types of implementations. Allowing the system to use different database connections interchangeably provides flexibility and extensibility by enabling the addition of new database connection types without modifying existing code.
- In our code (File: DatabaseConnection.java), the DatabaseConnection interface defines a contract for database connection management. For example, if the project requires use of PostgreSQL Database, the class implementing PostgreSQL database simply need to implement DatabaseConnection.
    - Line 15: establishes database connection. Any class implementing the database must provide this method.



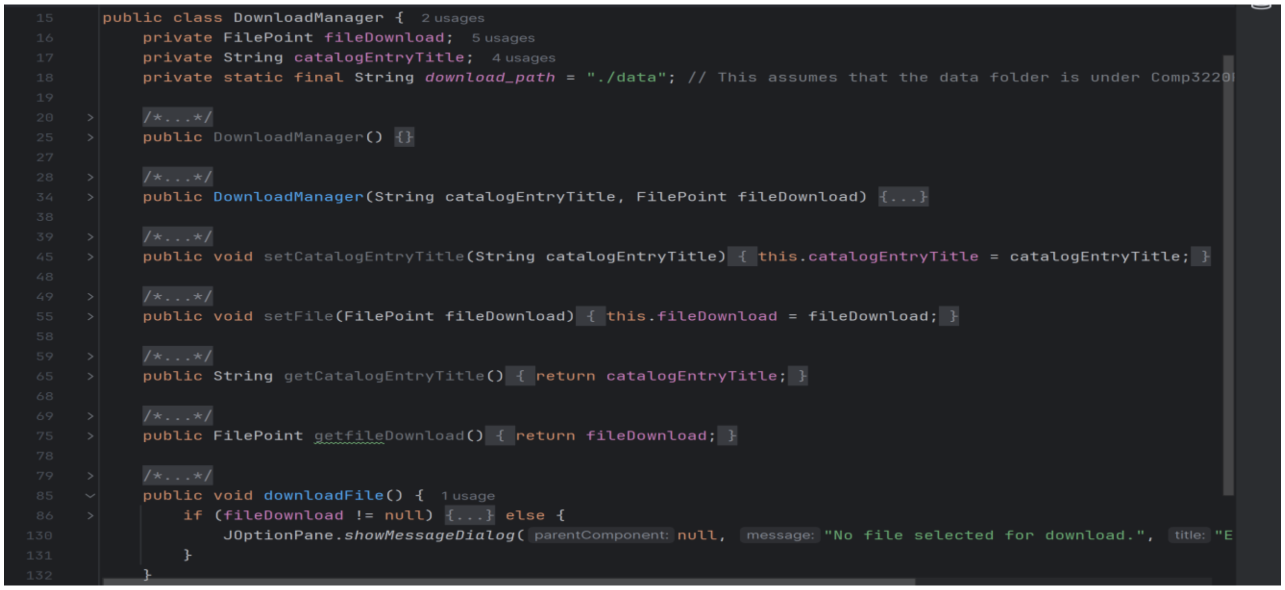
```java
/**
 * The DatabaseConnection interface provides an abstraction for establishing and managing
 * a database connection, ensuring flexibility and separation of connection logic.
 */
public interface DatabaseConnection {  5 usages  1 implementation
    /**
     * Establishes and returns a database connection.
     *
     * @return a Connection object to the database.
     * @throws SQLException if a database access error occurs.
     */
    Connection getConnection() throws SQLException;  20 usages  1 implementation

    /**
     * Closes the database connection if it is open.
     *
     * @throws SQLException if an error occurs while closing the connection.
     */
    void closeConnection() throws SQLException;  no usages  1 implementation
}
```

(Fig 6.0 DatabaseConnection Interface as Polymorphic)

Pattern: Pure Fabrication
Class: DownloadManager

- The purpose of this pattern is to assign a cohesive set of responsibilities to an artificial/convenience class that does not represent a concept of the problem domain. It is to achieve a specific functionality that doesn't belong to any domain-related class. This improves reusability and modularity.
- In our code (File: DownloadManager.java), The DownloadManager class manages file downloads. By encapsulating the download logic within this class, the system achieves better separation of concerns and reusability.
    - Line 85: checks if file has been selected and constructs file path, separating file location logic from other classes.



```java
15   public class DownloadManager {  2 usages
16       private FilePoint fileDownload;  5 usages
17       private String catalogEntryTitle;  4 usages
18       private static final String download_path = "./data"; // This assumes that the data folder is under Comp3220
19
20       /*...*/
25       public DownloadManager() {}
27
28       /*...*/
34       public DownloadManager(String catalogEntryTitle, FilePoint fileDownload) {...}
38
39       /*...*/
45       public void setCatalogEntryTitle(String catalogEntryTitle) { this.catalogEntryTitle = catalogEntryTitle; }
48
49       /*...*/
55       public void setFile(FilePoint fileDownload) { this.fileDownload = fileDownload; }
58
59       /*...*/
65       public String getCatalogEntryTitle() { return catalogEntryTitle; }
68
69       /*...*/
75       public FilePoint getfileDownload() { return fileDownload; }
78
79       /*...*/
85       public void downloadFile() {  1 usage
86           if (fileDownload != null) {...} else {
130              JOptionPane.showMessageDialog( parentComponent: null,  message: "No file selected for download.",  title: "E
131          }
132      }
```

(Fig 6.0 DownloadManager Class as Pure Fabrication)

Note: For complete class code implementation see attached class files as screenshot highlight only some parts of the code

Contributors
Written by Ingrid Diaz & Paul Osuji Edited and reformatted by Joel Junkukutty , Jonathan Chiu , Kerry Su , Maria Kandikova