Single Responsibility Principle -

Here's how the Single Responsibility Principle is applied in these classes:

```
public class AdminUser extends User {
   // ... other methods and fields

   public void adminFunction() {
      System.out.println("Performing admin-specific function.");
   }

   public void adminResetPassword(ResetPassword user, String newPassword) {
      System.out.println("Performing admin-specific function.");
      user.resetUserPassword(newPassword);
   }
}

public class User implements ResetPassword {
   private String username;
   private String password;

   public User(String username, String password) {
      this.username = username;
      this.password = password;
   }

   public String getUsername() {
      return username;
   }

   public void setUsername(String username) {
      this.username = username;
   }

   public String getPassword() {
      return password;
   }

   public void setPassword(String password) {
      this.password = password;
   }
```

```
    @Override
    public void resetUserPassword(String newPassword) {
        this.password = newPassword;
    }
}
```

- **AdminUser Class:**
    - The **AdminUser** class is responsible for representing an administrator user and performing admin-specific functions (**adminFunction**). It also has a method (**adminResetPassword**) that takes an instance implementing **ResetPassword** and resets the user password. The **AdminUser** class is focused on the responsibilities related to an admin user.
- **User Class:**
    - The **User** class represents a general user and implements the **ResetPassword** interface. It is responsible for managing user-related information (username, password) and providing a method (**resetUserPassword**) to reset the user password. The **User** class encapsulates responsibilities related to a regular user and password management.

By separating the responsibilities of admin-specific actions (**AdminUser**) and generic user-related actions (**User**), these classes adhere to the Single Responsibility Principle. Each class has a clear, specific responsibility, making the code more modular and easier to understand. The **ResetPassword** interface is a separate abstraction for the common behavior of resetting a user's password, allowing different user types to implement it according to their needs.

The **BiometricAuthenticationService** class is responsible for biometric authentication. It implements the **BiometricAuthentication** interface, defining a single responsibility related to authenticating users using biometric data.

```
public class BiometricAuthenticationService implements BiometricAuthentication {
    @Override
    public boolean authenticateWithBiometrics(User user, String biometricData) {
        // Implement biometric authentication logic
        // Return true if authentication succeeds, false otherwise
        return false;
    }
```

```
}
```

The **InMemoryUserRepository** class is responsible for managing user data in memory, including user registration and login with password. The methods are focused on specific responsibilities related to user data management, adhering to the SRP.

```
public class InMemoryUserRepository implements UserRepository {
   // ... other methods and fields

   public void registerUser(User user) {
      // User registration logic
      System.out.println("User registered successfully!");
   }

   // ... other methods

   public boolean loginUserWithPassword(User user, String password, PasswordAuthentication
authenticationService) {
      // Login with password logic
      if (authenticationService.authenticateWithPassword(user, password)) {
         System.out.println("Login successful!");
         if (user instanceof AdminUser) {
            ((AdminUser) user).adminFunction();
         }
         return true;
      } else {
         System.out.println("Login failed. Please check your credentials.");
         return false;
      }
   }

   // ... other methods
}
```

The **UsernamePasswordAuthentication** class is responsible for password-based authentication. It implements the **PasswordAuthentication** interface, encapsulating the responsibility of authenticating users based on their passwords.

```
public class UsernamePasswordAuthentication implements PasswordAuthentication {
   @Override
   public boolean authenticateWithPassword(User user, String password) {
      return user != null && user.getPassword().equals(password);
   }
}
```

These examples demonstrate how each class or service in the code has a specific responsibility, and any changes to that responsibility are encapsulated within the corresponding class. This adherence to SRP makes the code more modular, maintainable, and easier to comprehend.

Open Closed Principle -

The Open/Closed Principle (OCP) states that a class should be open for extension but closed for modification. This means that the behavior of a class can be extended without modifying its source code. Let's see how the Open/Closed Principle is applied in the provided code:

```
public class InMemoryUserRepository implements UserRepository {
   // ... other methods and fields

   public boolean loginUserWithPassword(User user, String password, PasswordAuthentication
authenticationService) {
      // Login with password logic
      if (authenticationService.authenticateWithPassword(user, password)) {
         System.out.println("Login successful!");
         if (user instanceof AdminUser) {
            ((AdminUser) user).adminFunction();
         }
         return true;
      } else {
         System.out.println("Login failed. Please check your credentials.");
         return false;
      }
   }
```

```
    // ... other methods
}
```

The **InMemoryUserRepository** class demonstrates the Open/Closed Principle by being open for extension. The method **loginUserWithPassword** accepts a **PasswordAuthentication** interface, allowing different authentication strategies to be passed in without modifying the existing code. This allows for the introduction of new authentication methods without changing the core logic of the **InMemoryUserRepository** class.

BiometricAuthenticationService Class:

```
public class BiometricAuthenticationService implements BiometricAuthentication {
    @Override
    public boolean authenticateWithBiometrics(User user, String biometricData) {
        // Implement biometric authentication logic
        // Return true if authentication succeeds, false otherwise
        return false;
    }
}
```

Liskov Substitution Principle (LSP)

The Liskov Substitution Principle (LSP) states that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. Let's examine the usage of Liskov Substitution Principle in the provided code.

```
public class AdminUser extends User {
    // ... other methods and fields

    public void adminFunction() {
        System.out.println("Performing admin-specific function.");
    }

    public void adminResetPassword(ResetPassword user, String newPassword) {
        System.out.println("Performing admin-specific function.");
        user.resetUserPassword(newPassword);
    }
}
```

The **AdminUser** class extends the **User** class. Instances of **AdminUser** can be used in places where instances of **User** are expected. For example, in the **InMemoryUserRepository** class:

```
public boolean loginUserWithPassword(User user, String password, PasswordAuthentication
authenticationService) {
   // Login with password logic
   if (authenticationService.authenticateWithPassword(user, password)) {
      System.out.println("Login successful!");
      if (user instanceof AdminUser) {
         ((AdminUser) user).adminFunction();
      }
      return true;
   } else {
      System.out.println("Login failed. Please check your credentials.");
      return false;
   }
}
```

The **loginUserWithPassword** method takes a **User** parameter. Since **AdminUser** is a subclass of **User**, an instance of **AdminUser** can be passed to this method, and the method can seamlessly call **adminFunction()** on it. This adheres to the Liskov Substitution Principle.

In summary, the Liskov Substitution Principle is demonstrated when a subclass (**AdminUser**) can be used in place of its superclass (**User**) without affecting the correctness of the program. Instances of **AdminUser** can be treated as instances of **User** where appropriate, and the code remains consistent and functional.

Interface Segregation Principle (ISP)

The Interface Segregation Principle (ISP) states that a class should not be forced to implement interfaces it does not use. In other words, a class should only be required to implement methods that are relevant to its behavior. Let's analyze the usage of Interface Segregation Principle in the provided code.

BiometricAuthentication Interface

```
public interface BiometricAuthentication {
   boolean authenticateWithBiometrics(User user, String biometricData);
}
```

The **BiometricAuthentication** interface defines a single method **authenticateWithBiometrics**. Classes that implement this interface are expected to provide an implementation for biometric authentication. This adheres to ISP by ensuring that classes implementing **BiometricAuthentication** only need to implement the methods relevant to biometric authentication.

PasswordAuthentication Interface:

```
public interface PasswordAuthentication {
   boolean authenticateWithPassword(User user, String password);
}
```

Similarly, the **PasswordAuthentication** interface defines a single method **authenticateWithPassword**. Classes implementing this interface are expected to provide an implementation for password-based authentication. This follows ISP by requiring classes to implement only the methods relevant to password authentication.

ResetPassword Interface:

```
public interface ResetPassword {
   void resetUserPassword(String newPassword);
}
```

The **ResetPassword** interface defines a single method **resetUserPassword**. Classes implementing this interface are expected to provide an implementation for resetting a user's password. ISP is followed here, as classes implementing **ResetPassword** only need to implement the method relevant to password reset.

UserRepository Interface:

```
public interface UserRepository {
   void registerUser(User user);

   boolean loginUserWithPassword(User user, String password, PasswordAuthentication
authenticationService);

   boolean loginUserWithBiometrics(User user, String biometricData, BiometricAuthentication
authenticationService);
}
```

The **UserRepository** interface defines methods for registering users and authenticating them using both password and biometrics. While this interface combines multiple methods, it is still following ISP, as implementing classes can choose to provide only the relevant methods based

on their capabilities. For example, a class might implement only password-based authentication and not biometric authentication.

Dependency Injection

```
public class OnlineStore {
   public static void main(String[] args) {
      UserRepository userRepository = new InMemoryUserRepository();

      User newUser = new User("john_doe", "password123");
      userRepository.registerUser(newUser);

      User adminUser = new AdminUser("admin_user", "admin_password");
      userRepository.registerUser(adminUser);

      PasswordAuthentication passwordAuth = new UsernamePasswordAuthentication();
      userRepository.loginUserWithPassword(newUser, "password123", passwordAuth);
      userRepository.loginUserWithPassword(adminUser, "admin_password", passwordAuth);
   }
}
```

The **OnlineStore** class demonstrates Dependency Injection by creating an instance of **InMemoryUserRepository** and passing it to the **UserRepository** variable. This allows the **OnlineStore** class to use the functionality provided by **InMemoryUserRepository** without being tightly coupled to its implementation.