### Detailed Notes: Refactoring to Comply with Single Responsibility Principle

#### Introduction

* The **Single Responsibility Principle (SRP)** states that a class should have only one reason to change.
* In this example, we analyze a class called UserController, identify SRP violations, and demonstrate how to refactor the code to adhere to SRP.

#### Ask this question to identify if your class/module is following SRP or not

1. What exactly this class/module is supposed to do and what exactly it is doing right now.

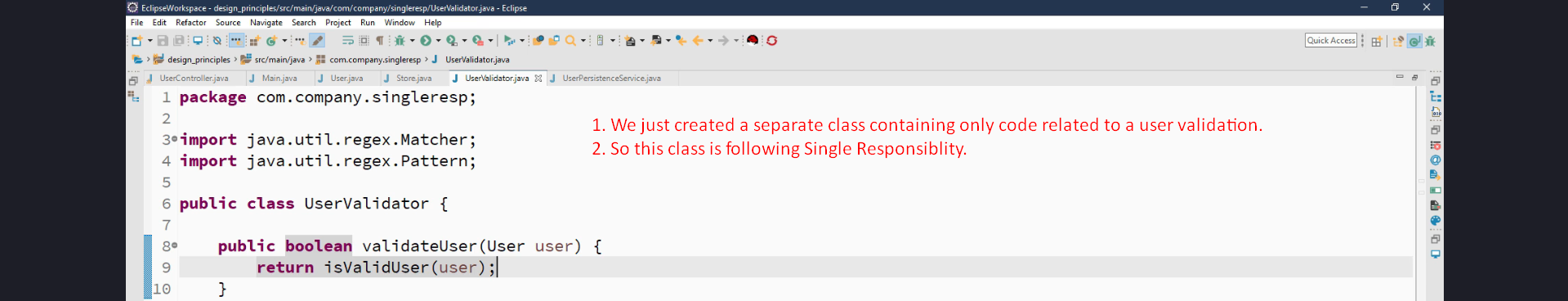
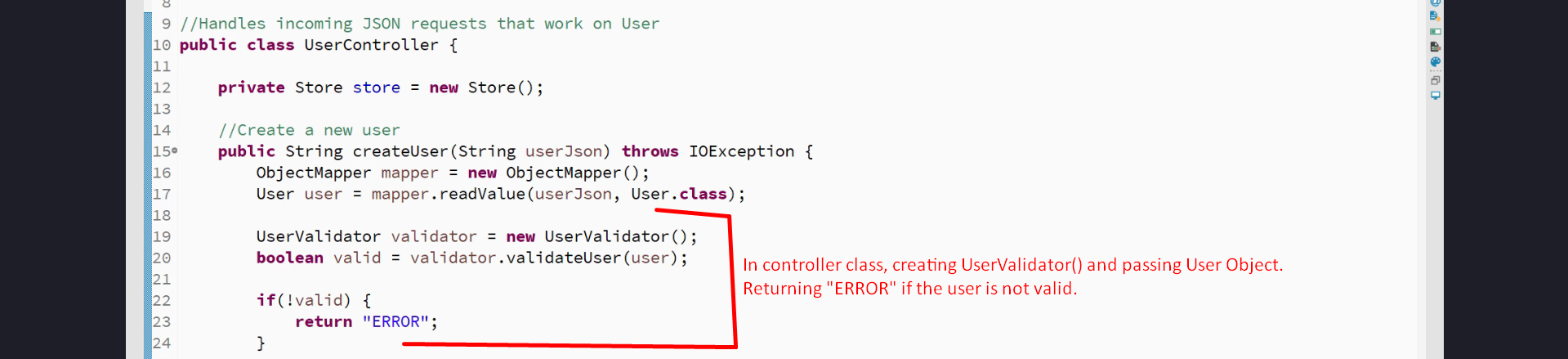
#### Initial Code Analysis

* **Class Name:** UserController : Controller is supposed to receive a req and forward it to service but all the following responsibility code is inside the controller class 😊
  + **Responsibilities:** But Here controlling is doing all these things.
    1. Receiving client requests.
    2. Parsing JSON strings into Java objects -> Creating User object from JSON.
    3. Validating the user object.
    4. Persisting the user object -> repo.store(User)



* **Problem:**
  + The class has multiple reasons to change:
    1. Changes to validation logic.
    2. Changes to persistence logic (e.g., database or storage changes).
    3. Changes to request handling.
* **Impact:**
  + Violates SRP, causing ripple effects during changes, increasing maintenance complexity.

#### Refactoring Process

1. **Preparation:**
   * Ensure test cases exist to validate functionality before and after refactoring.
   * Example: Use a main method or unit tests like JUnit to verify behavior.
2. **Step 1: Extract Validation Logic**
   * Create a new class UserValidator to handle user validation.
   * Move validation methods from UserController to UserValidator.
   * Update UserController to delegate validation tasks to UserValidator.  
       
       
     
3. **Step 2: Extract Persistence Logic**
   * Create a new class UserPersistenceService to handle storage tasks.
   * Move persistence logic (e.g., database interaction) from UserController to UserPersistenceService.

**Code Snippet:**

public class UserPersistenceService {

private Store store = new Store(); // Simulates a database

public void saveUser(User user) {

store.store(user);

}

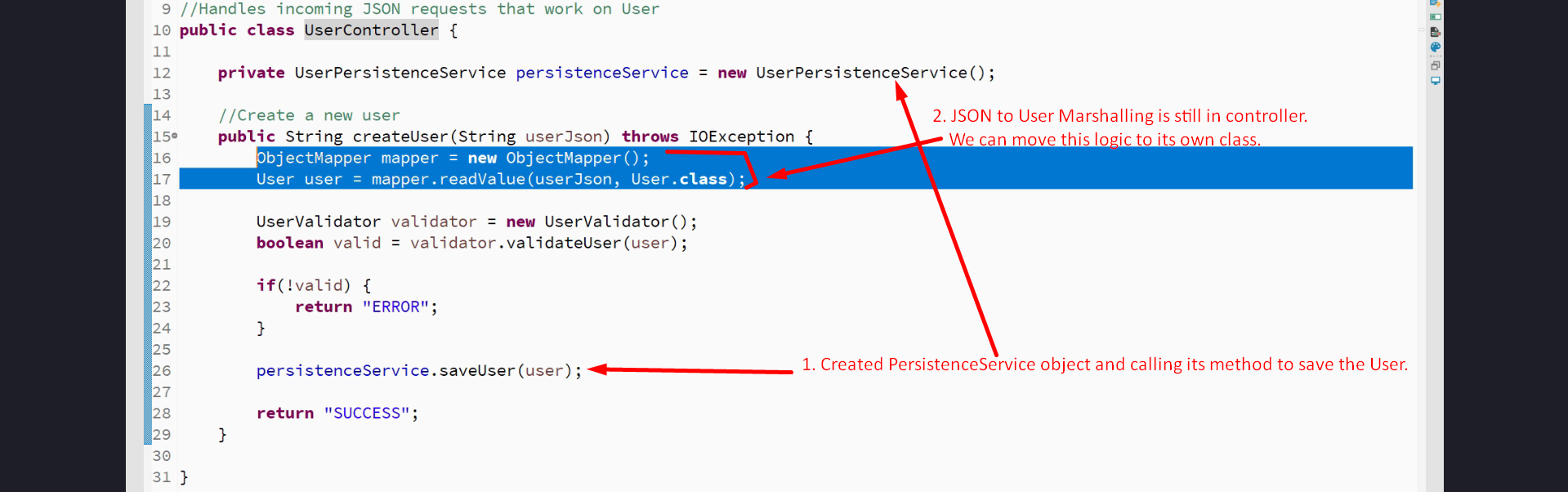
}

// Updated UserController

UserPersistenceService persistenceService = new UserPersistenceService();

persistenceService.saveUser(user);

1. **Step 3: Simplify UserController**
   * Ensure UserController only handles:
     + Receiving requests.
     + Coordinating actions (validation, persistence).
   * Delegate responsibilities to the appropriate classes.

**Code Snippet:**

#### Final Design

* **UserController:** Handles client requests, delegates tasks.
* **UserValidator:** Handles all validation logic.
* **UserPersistenceService:** Manages persistence operations.

**Diagram:**

+-------------------+ +-------------------+ +---------------------------+

| UserController | | UserValidator | | UserPersistenceService |

|-------------------| |-------------------| |---------------------------|

| - createUser() | | - validateUser() | | - saveUser() |

+-------------------+ +-------------------+ +---------------------------+

#### Benefits of Refactoring

* **Adheres to SRP:** Each class now has a single responsibility.
* **Improved Maintainability:** Changes to validation or persistence affect only the corresponding class.
* **Reduced Ripple Effect:** Modifications are localized, minimizing unintended side effects.

#### Key Takeaways

1. Identify and isolate responsibilities in a class.
2. Use **delegation (Method Call)** to ensure each class focuses on one responsibility.
3. Validate functionality after each refactoring step using test cases.

#### Real-Life Application

* In a real Spring application:
  + Use dependency injection (e.g., @Autowired) to manage validator and persistence service instances.
  + Utilize frameworks like JPA for database operations.
  + Write comprehensive unit tests to ensure refactored code meets requirements.

By following these steps, the UserController adheres to the Single Responsibility Principle, resulting in cleaner and more maintainable code.