**Spring 6 Practice Assignment - XML Configuration with Autowiring**

**Objective**: The objective of this assignment is to create a Spring 6 application using XML configuration that adheres to SOLID principles, promotes loose coupling, and demonstrates the usage of autowiring.

Instructions:

1. Create a new Spring 6 project using XML configuration.
2. Use the core concepts of Spring, including autowiring, to complete the tasks mentioned below.
3. Write your code in separate XML and Java files as required.
4. Make sure to provide comments explaining your code logic.
5. Test your application thoroughly to ensure it works as expected.

Tasks:

1. Create an interface called **MessageService** with a method **sendMessage(String message)**. This interface will define the contract for sending messages.
2. Create two classes that implement the **MessageService** interface: **EmailService** and **SMSService**. Implement the **sendMessage** method in both classes to print the message and the respective service name. These classes will be responsible for sending messages via email and SMS, respectively.
3. Create a class called **NotificationManager** that has dependencies on the **MessageService** interface. This class will manage notifications and use the appropriate **MessageService** implementation(s) to send notifications.
   * Implement setter methods for the **MessageService** dependencies in the **NotificationManager** class.
   * Add a method called **sendNotification(String message)** in the **NotificationManager** class that calls the **sendMessage** method of each **MessageService** dependency. This method will send notifications using all available message services.
4. Configure the Spring bean definitions in an XML file.
   * Define beans for the **EmailService**, **SMSService**, and **NotificationManager** classes in the XML file.
   * Use autowiring to inject the **MessageService** dependencies into the **NotificationManager** bean.
   * Test different types of autowiring:
     + Use constructor autowiring for one of the **MessageService** dependencies.
     + Use setter autowiring for the other **MessageService** dependency.
5. Create a main class to load the Spring context and retrieve the **NotificationManager** bean.
   * Load the Spring context using the XML configuration file.
   * Retrieve the **NotificationManager** bean from the context.
6. Test the application by calling the **sendNotification** method of the **NotificationManager** bean with a sample message. Ensure that the messages are sent using the appropriate message services (email and SMS).

SOLID Principles: Ensure that your code adheres to the following SOLID principles:

1. Single Responsibility Principle (SRP):
   * Each class should have a single responsibility.
   * The **EmailService** class should be responsible only for sending emails.
   * The **SMSService** class should be responsible only for sending SMS messages.
   * The **NotificationManager** class should be responsible for managing notifications.
2. Open-Closed Principle (OCP):
   * Design the classes and interfaces in a way that allows for extension without modifying existing code.
   * Ensure that the existing classes can be extended to support new message services without modification.
3. Liskov Substitution Principle (LSP):
   * Ensure that the **EmailService** and **SMSService** classes can be used interchangeably wherever **MessageService** is expected.
   * The **NotificationManager** should be able to work with any implementation of the **MessageService** interface.
4. Interface Segregation Principle (ISP):
   * Keep interfaces small and focused on a specific set of methods.
   * The **MessageService** interface should only contain methods related to sending messages, avoiding unnecessary methods that are not relevant to all implementations.
5. Dependency Inversion Principle (DIP):
   * Depend on abstractions rather than concrete implementations.
   * The **NotificationManager** should depend on the **MessageService** interface rather than specific implementations (**EmailService** or **SMSService**).
   * Use dependency injection to provide the implementations.

Loose Coupling: Ensure that your application demonstrates loose coupling by following these guidelines:

1. Dependency Injection (DI):
   * Use dependency injection to inject dependencies into classes.
   * Inject the **MessageService** dependencies into the **NotificationManager** class using constructor or setter injection.
2. Avoid Hard-Coded Dependencies:
   * Remove any hard-coded dependencies between classes.
   * Use dependency injection to provide the **MessageService** implementations to the **NotificationManager** class.
3. Use Interfaces or Abstract Classes:
   * Depend on interfaces or abstract classes rather than concrete implementations.
   * The **NotificationManager** should depend on the **MessageService** interface rather than specific implementations (**EmailService** or **SMSService**).
4. Favor Composition over Inheritance:
   * Encapsulate dependencies using composition rather than relying on inheritance.
   * Inject the **EmailService** and **SMSService** dependencies into the **NotificationManager** class.
5. Use Inversion of Control (IoC) Containers:
   * Utilize an IoC container, such as the Spring container, to manage the creation and wiring of objects.
   * Define bean configurations in XML or Java-based configuration files and let the container handle the creation and injection of dependencies.

Submission:

1. Compress the entire Spring project into a zip file.
2. Include any necessary instructions or explanations in a separate README.txt file if needed.

Note: Make sure to test your application thoroughly to ensure that it works as expected and meets the specified requirements.