**Applying SOLID Principles to Spring Boot Applications**

In software development, **Object-Oriented Design** is really important for creating code that can be easily changed, expanded, and used again.

The **SOLID** principles are a set of five design principles in **object-oriented programming** and **software development** that aim to create more maintainable, flexible, and scalable software. They were introduced by Robert C. Martin and are widely used as guidelines for designing clean and efficient code. Each letter in the word “**SOLID**” represents one of these principles:

1. **S**ingle Responsibility Principle (SRP)
2. **O**pen/Closed Principle (OCP)
3. **L**iskov’s Substitution Principle (LSP)
4. **I**nterface Segregation Principle (ISP)
5. **D**ependency Inversion Principle (DIP)

In this article, we will examine how each principle is used in the Spring Boot application

1. **Single Responsibility Principle (SRP)**

Robert C. Martin describes it:

*A class should have one, and only one, reason to change.*

The Single Responsibility principle has**two key principles**, as the name suggests.



Let’s examine the incorrect usage in the example below.

// Incorrect implementation of SRP  
@RestController  
@RequestMapping("/report")  
public class ReportController {  
  
 private final ReportService reportService;  
  
  
 public ReportController(ReportService reportService) {  
 this.reportService = reportService;  
 }  
  
 @PostMapping("/send")  
 public ResponseEntity<Report> generateAndSendReport(@RequestParam String reportContent,  
 @RequestParam String to,  
 @RequestParam String subject) {  
 String report = reportService.generateReport(reportContent);  
 reportService.sendReportByEmail(report, to, subject);  
 return new ResponseEntity<>(HttpStatus.OK);  
 }  
}

// Incorrect implementation of SRP  
// The class is responsible for generating a report and sending email  
@Service  
@Slf4j  
public class ReportServiceImpl implements ReportService {  
  
 private final ReportRepository reportRepository;  
  
 public ReportServiceImpl(ReportRepository reportRepository) {  
 this.reportRepository = reportRepository;  
 }  
  
 @Override  
 public String generateReport(String reportContent) {  
 Report report = new Report();  
 report.setReportContent(reportContent);  
 return reportRepository.save(report).toString();  
 }  
  
 @Override  
 public void sendReportByEmail(Long reportId, String to, String subject) {  
 Report report = findReportById(reportId);  
 sendEmail(report.getReportContent(), to, subject);  
 }  
  
 private Report findReportById(Long reportId) {  
 return reportRepository.findById(reportId)  
 .orElseThrow(() -> new RuntimeException("Report not found"));  
 }  
  
 private void sendEmail(String content, String to, String subject) {  
 log.info(content, to, subject);  
 }

As you can see **ReportService** has multiple responsibilities which violates Single Responsibility:

* **Generate Report:** The class is responsible for generating a report and saving it to the repository in the generateReport method.
* **Send Report by Email**: The class is also responsible for sending a report by email in the sendReportByEmail method.

When creating the code, it requires to avoid putting too many tasks in one place — whether it’s a class or method.

This makes the code complex and hard to handle. It also makes it tricky to make small changes because they might affect other parts of the code, requiring to test everything even for minor updates.

Let’s correct this implementation;

To adhere to SRP, these responsibilities were separated into different classes.

@RestController  
@RequestMapping("/report")  
public class ReportController {  
  
 private final ReportService reportService;  
 private final EmailService emailService;  
  
 public ReportController(ReportService reportService, EmailService emailService) {  
 this.reportService = reportService;  
 this.emailService = emailService;  
 }  
  
 @PostMapping("/send")  
 public ResponseEntity<Report> generateAndSendReport(@RequestParam String reportContent,  
 @RequestParam String to,  
 @RequestParam String subject) {  
 // correct impl reportService is responsible for generation  
 Long reportId = Long.valueOf(reportService.generateReport(reportContent));  
 // correct impl emailService is responsible for sending  
 emailService.sendReportByEmail(reportId, to, subject);  
 return new ResponseEntity<>(HttpStatus.OK);  
 }  
}

@Service  
public class ReportServiceImpl implements ReportService {  
  
 private final ReportRepository reportRepository;  
  
  
 public ReportServiceImpl(ReportRepository reportRepository, EmailService emailService) {  
 this.reportRepository = reportRepository;  
 }  
  
 @Override  
 public String generateReport(String reportContent) {  
 Report report = new Report();  
 report.setReportContent(reportContent);  
 return reportRepository.save(report).toString();  
 }  
  
  
@Service  
public class EmailServiceImpl implements EmailService {  
  
 private final ReportRepository reportRepository;  
  
 public EmailServiceImpl(ReportRepository reportRepository) {  
 this.reportRepository = reportRepository;  
 }  
  
 @Override  
 public void sendReportByEmail(Long reportId, String to, String subject) {  
 Report report = findReportById(reportId);  
 if (ObjectUtils.isEmpty(report) || !StringUtils.hasLength(report.getReportContent())) {  
 throw new RuntimeException("Report or report content is empty");  
 }  
 }  
  
 private Report findReportById(Long reportId) {  
 return reportRepository.findById(reportId)  
 .orElseThrow(() -> new RuntimeException("Report not found"));  
 }  
  
}

The refactored code includes changes below;

* ReportServiceImpl is responsible for generating reports.
* EmailServiceImpl is responsible for sending reports -that were generated by ReportServiceImpl-by email.
* The ReportController manages the process of generating and sending reports by using the appropriate services.

**2. Open/Closed Principle (OCP)**

The Open-Closed Principle says that a **class should be open for extension and closed to modification.**This helps avoid introducing bugs to a working application. In simpler terms, this means that **you should be able to add new functionality to a class without changing its existing code**.

Let’s examine the incorrect usage in the example below.

// Incorrect implementation violating OCP  
public class ReportGeneratorService {  
 public String generateReport(Report report) {  
 if ("PDF".equals(report.getReportType())) {  
 // Incorrect: Direct implementation for generating PDF report  
 return "PDF report generated";  
 } else if ("Excel".equals(report.getReportType())) {  
 // Incorrect: Direct implementation for generating Excel report  
 return "Excel report generated";  
 } else {  
 return "Unsupported report type";  
 }  
 }  
}

In this incorrect implementation, the generateReport method of ReportService has conditional statements to check the report type and directly generates the report accordingly. This violates the Open-Closed Principle because **if you want to add support for a new report type, you would need to modify this class**.

Let’s correct this implementation;

public interface ReportGenerator {  
 String generateReport(Report report);  
}  
  
// Concrete implementation for generating PDF reports  
@Component  
public class PdfReportGenerator implements ReportGenerator {  
 @Override  
 public String generateReport(Report report) {  
 // Impl of pdf report  
 return String.format("PDF report generated for %s", report.getReportType());  
 }  
}  
  
// Concrete implementation for generating Excel reports  
@Component  
public class ExcelReportGenerator implements ReportGenerator {  
 @Override  
 public String generateReport(Report report) {  
 // Impl of excel report  
 return String.format("Excel report generated for %s", report.getReportType());  
 }  
}  
  
// Service that follows OCP  
@Service  
public class ReportGeneratorService {  
  
 private final Map<String, ReportGenerator> reportGenerators;  
  
 @Autowired  
 public ReportGeneratorService(List<ReportGenerator> generators) {  
 // Initialize the map of report generators  
 this.reportGenerators = generators.stream()  
 .collect(Collectors.toMap(generator -> generator.getClass().getSimpleName(), Function.identity()));  
 }  
  
 public String generateReport(Report report, String reportType) {  
 return reportGenerators.getOrDefault(reportType, unsupportedReportGenerator())  
 .generateReport(report);  
 }  
  
 private ReportGenerator unsupportedReportGenerator() {  
 return report -> "Unsupported report type";  
 }  
}

**Interface ->**ReportGenerator

* Added **an interface (**ReportGeneratorto define a common method for report generation.

**Concrete Implementations** ->PdfReportGenerator and ExcelReportGenerator

* Created classes implementing the interface **for PDF and Excel report generation**.
* Followed the Open-Closed principle of allowing extension without modifying existing code.

**Report Generator Service ->** ReportGeneratorService

* Introduced a service managing different report generator implementations.
* **Allows adding new report generators without changing existing code.**

In a summary, the service handles these implementations dynamically, making it easy to add new features without changing existing code, following the Open-Closed Principle.

**3. Liskov’s Substitution Principle (LSP)**

The Liskov Substitution Principle states that if you have a class, you **should be able to replace it with a subclass without causing any problems in your program**.

In other words, you can use the specialized version wherever you use the more general version, and everything should still work correctly.

Let’s examine the incorrect usage in the example below.

// Incorrect implementation violating LSP  
public class Bird {  
 public void fly() {  
 // I can fly  
 }  
  
 public void swim() {  
 // I can swim  
 }  
}  
  
public class Penguin extends Bird {  
  
 // Penguins cannot fly, but we override the fly method and throws Exception  
 @Override  
 public void fly() {  
 throw new UnsupportedOperationException("Penguins cannot fly");  
 }  
}

Let’s correct this implementation;

// Correct implementation for LSP  
public class Bird {  
  
 // methods  
}  
  
public interface Flyable {  
 void fly();  
}  
  
public interface Swimmable {  
 void swim();  
}  
  
  
public class Penguin extends Bird implements Swimmable {  
 // Penguins cannot fly, therefore we only implement swim interface  
 @Override  
 public void swim() {  
 System.out.println("I can swim");  
 }  
}  
  
public class Eagle extends Bird implements Flyable {  
 @Override  
 public void fly() {  
 System.out.println("I can fly");  
 }  
}

* Birdclass serves **as a base class for birds** and includes common properties or methods shared among all birds.
* Introduced Flyable and Swimmable interfaces **to represent specific behaviors.**
* In the Penguin class, implemented the Swimmable interface to reflect penguins' swimming ability.
* In the Eagle class, implemented the Flyable interface to reflect eagles' flying ability.

**By separating specific behaviors into interfaces and implementing them in subclasses**, we follow Liskov Substitution Principle that lets us switch subclasses without causing any surprising issues.

**4. Interface Segregation Principle (ISP)**

Interface Segregation Principle states that **larger interfaces should be split into smaller ones.**

**By doing so, we can ensure that implementing classes only need to be concerned about the methods that are of interest to them.**

Let’s examine the incorrect usage in the example below.

public interface Athlete {  
  
 void compete();  
  
 void swim();  
  
 void highJump();  
  
 void longJump();  
}  
  
// Incorrect implementation violating Interface Segregation  
public class JohnDoe implements Athlete {  
 @Override  
 public void compete() {  
 System.out.println("John Doe started competing");  
 }  
  
 @Override  
 public void swim() {  
 System.out.println("John Doe started swimming");  
 }  
  
 @Override  
 public void highJump() {  
 // Not neccessary for John Doe  
 }  
  
 @Override  
 public void longJump() {  
 // Not neccessary for John Doe  
 }  
}

Imagine **John Doe as a swimmer**. He is forced to provide empty implementations for highJump and longJump, which are irrelevant to his role as a swimmer.

Let’s correct this implementation;

public interface Athlete {  
  
 void compete();  
}  
  
public interface JumpingAthlete {  
  
 void highJump();  
  
 void longJump();  
}  
  
public interface SwimmingAthlete {  
  
 void swim();  
}  
  
// Correct implementation for Interface Segregation  
public class JohnDoe implements Athlete, SwimmingAthlete {  
 @Override  
 public void compete() {  
 System.out.println("John Doe started competing");  
 }  
  
 @Override  
 public void swim() {  
 System.out.println("John Doe started swimming");  
 }  
}

The original Athlete interface has been split into three separate interfaces: Athlete for general activities, JumpingAthlete for jumping-related activities, and SwimmingAthlete for swimming.

This adheres to the Interface Segregation Principle, ensuring that **a class is not forced to implement methods it does not need.**

[Obtained the example from the post authored by Emmanouil Gkatziouras](https://dzone.com/articles/solid-principles-interface-segregation-principle)

**5. Dependency Inversion Principle (DIP)**

The Dependency Inversion Principle (DIP) states that**high level modules should not depend on low level modules**; both should depend on abstractions. Abstractions should not depend on details.

Let’s examine the incorrect usage in the example below.

// Incorrect impl of Dependency Inversion Principle  
@Service  
public class PayPalPaymentService {  
  
 public void processPayment(Order order) {  
 // payment processing logic  
 }  
}  
  
@RestController  
public class PaymentController {  
  
 // Direct dependency on a specific implementation  
 private final PayPalPaymentService paymentService;  
  
 // Constructor directly initializes a specific implementation  
 public PaymentController() {  
 this.paymentService = new PayPalPaymentService();  
 }  
  
 @PostMapping("/pay")  
 public void pay(@RequestBody Order order) {  
 paymentService.processPayment(order);  
 }  
}

Let’s correct this implementation;

// Introduced interface  
public interface PaymentService {  
 void processPayment(Order order);  
}  
  
// Implemented interface in a service class  
@Service  
public class PayPalPaymentService implements PaymentService {  
 @Override  
 public void processPayment(Order order) {  
 // payment processing logic  
 }  
}  
  
@RestController  
public class PaymentController {  
  
 private final PaymentService paymentService;  
  
 // Constructor injection  
 public PaymentController(PaymentService paymentService) {  
 this.paymentService = paymentService;  
 }  
  
 @PostMapping("/pay")  
 public void pay(@RequestBody Order order) {  
 paymentService.processPayment(order);  
 }  
}

* Introduced the PaymentService interface.
* Injected the PaymentService interface into the controller's constructor in order to provide abstraction in controller.
* **Controller depends on the abstraction** (PaymentService), allowing for dependency injection of any class implementing the interface.

The example is adapted [here](https://amrtechuniverse.com/solid-principles-in-spring-boot)

**Dependency Inversion Principle (DIP)** and **Dependency Injection (DI)** are connected concepts in the Spring Framework. DIP, introduced by [Uncle Bob Martin](https://cleancoders.com/episode/clean-code-episode-13), is about keeping code loosely connected. It separates code for Dependency Injection in Spring, where the framework manages the application during runtime.

**Conclusion**

SOLID principles are essential in Object-Oriented Programming (OOP) because they provide a set of guidelines and best practices to design software that is more maintainable, flexible, and scalable.

In this article, we started by discussing mistakes in applying SOLID principles in Java applications. After that, we examined the related example to see how these issues were fixed.

All examples are presented at a basic level, you can refer to the provided references for further reading.

**References**

<https://www.freecodecamp.org/news/solid-principles-explained-in-plain-english/>

<https://www.baeldung.com/solid-principles>

<https://www.linkedin.com/pulse/solid-principles-software-development-java-ramiro-victor/>

<https://amrtechuniverse.com/solid-principles-in-spring-boot>

<https://dzone.com/articles/solid-principles-interface-segregation-principle>

<https://trekhleb.medium.com/solid-principles-around-you-6db2f0e12139>

<https://springframework.guru/principles-of-object-oriented-design/dependency-inversion-principle/>