

Advanced Software Engineering

Part 06 — SOLID Principles

Dr. Amjad AbuHassan

Introduction to SOLID

- In simple words, design principles help us to create more maintainable, understandable, and flexible software.
- The goal of the SOLID principles is to reduce dependencies so that we can change one area of software without impacting others.
 - make designs easier to understand, maintain, and extend.
 - easier for software engineers to avoid issues and to build adaptive, and effective software.

Introduction to SOLID cont.

- Principles come with many benefits, but following the principles generally leads to writing longer and more complex code.
 - This means that it can extend the design process and make development a little more difficult.
- This extra time and effort are well worth it because it makes software so much easier to maintain, test, and extend.

SOLID Principles

- SOLID is an acronym for the first five object-oriented design (OOD) principles by Robert C. Martin.



Single Responsibility Principle (SRP)

- A class should have one and only one reason to change, meaning that a class should have only one job.
 - class should have only one responsibility and that responsibility should be encapsulated within the class.
- If you cannot come up with a meaningful name for your class, then it's probably doing too much.

Why we need the SRP

- The application that has multiple classes following this principle:
 - Becomes more maintainable, easier to understand.
 - Testing and writing test cases is much simpler.
 - The overall code quality will be better, thereby having fewer defects.
- Supports separation of concerns: which is a key aspect of modular software design.

What happens if we don't use the SRP?

- Classes with multiple responsibilities are
 - more difficult to modify without causing unintended consequences.
 - it becomes more complex and harder to understand.
 - Classes more likely to be tightly coupled to other classes, making it harder to reuse or test.

Code Example 1: Bank Account



```
1  class BankAccount {  
2      private int balance;  
3  
4      public void deposit(int amount) {  
5          this.balance += amount;  
6      }  
7  
8      public void withdraw(int amount) {  
9          this.balance -= amount;  
10     }  
11  
12     public void printStatement() {  
13         // code to print account statement  
14     }  
15 }
```


Code Example 2: Employee

```
1  class Employee {
2      private String name;
3      private int salary;
4
5      public void setName(String name) {
6          this.name = name;
7      }
8
9      public void setSalary(int salary) {
10         this.salary = salary;
11     }
12
13     public void calculateTax() {
14         // code to calculate employee tax
15     }
16
17     public void sendEmail() {
18         // code to send email to employee
19     }
20 }
```

Code Example 3: Logger



```
1  class Logger {
2      private String logFile;
3
4      public void setLogFile(String logFile) {
5          this.logFile = logFile;
6      }
7
8      public void logMessage(String message) {
9          // code to log message to log file
10     }
11
12     public void sendEmail(String message) {
13         // code to send email with message
14     }
15 }
```

Open/Closed Principle (OCP)

- Principle states that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.
- Open to an extension - you should design your classes so that new functionality can be added as new requirements are generated.
- Closed for modification - Once you have developed a class you should never modify it, except to correct bugs.

Open/Closed Principle (OCP) cont.

- Design and code should be done in a way that new functionality should be added with minimum or no changes in the existing code
- When needs to extend functionality - avoid tight coupling, don't use if-else/switch-case logic, do code refactoring as required.
- Techniques to achieve - Inheritance, Polymorphism, Generics
- Pattern to apply – Strategy Pattern, Template Method


Why we need the OCP

- Supports software evolution: By allowing for changes to be made through extension rather than modification
- Improves maintainability: When new functionality is added through extension, existing code is left unchanged
- Supports separation of concerns: which is a key aspect of modular software design.

What happens if we don't use the OCP?


- Increases coupling
- Decreases maintainability: it is harder to understand the code and debug issues
- Decreases testability: it is harder to test the code

Code Example 1: File Processor



```
1  class FileProcessor {
2      public void processFile(String fileName) {
3          // code to process file
4      }
5  }
6
7  class TextFileProcessor extends FileProcessor {
8      public void processFile(String fileName) {
9          // code to process text file
10     }
11 }
12
13 class BinaryFileProcessor extends FileProcessor {
14     public void processFile(String fileName) {
15         // code to process binary file
16     }
17 }
```

Code Example 2: Order Processing System



```
1  class OrderProcessor {
2      public void processOrder(Order order) {
3          // code to process order
4      }
5  }
6
7  class PhysicalOrderProcessor extends OrderProcessor {
8      public void processOrder(Order order) {
9          // code to process physical order
10     }
11 }
12
13 class DigitalOrderProcessor extends OrderProcessor {
14     public void processOrder(Order order) {
15         // code to process digital order
16     }
17 }
```


Liskov Substitution Principle (LSP)

Let $q(x)$ be a property provable about objects of x of type T . Then $q(y)$ should be provable for objects y of type S where S is a subtype of T .

- Every subclass (derived) should be substitutable for its base class.
- Introduced by Barbara Liskov (1987) "Data Abstraction & Hierarchy"
- This principle applies to inheritance hierarchies and is just an extension of the Open-Closed Principle.

LSP Definition

- We must make sure that new derived classes are extending the base classes without changing their original behavior.
 - Basically, derived classes should never do less than their base class.
- You should consider how the client programs are using the class hierarchy.
 - If a subtype does something that the client of the supertype does not expect, then this is in violation of LSP.

Why we need the LSP

- It helps to ensure that our software systems are maintainable and scalable.
- Easier to make changes to our software systems without having to modify existing code.
- Reduce the risk of bugs and other problems in our code

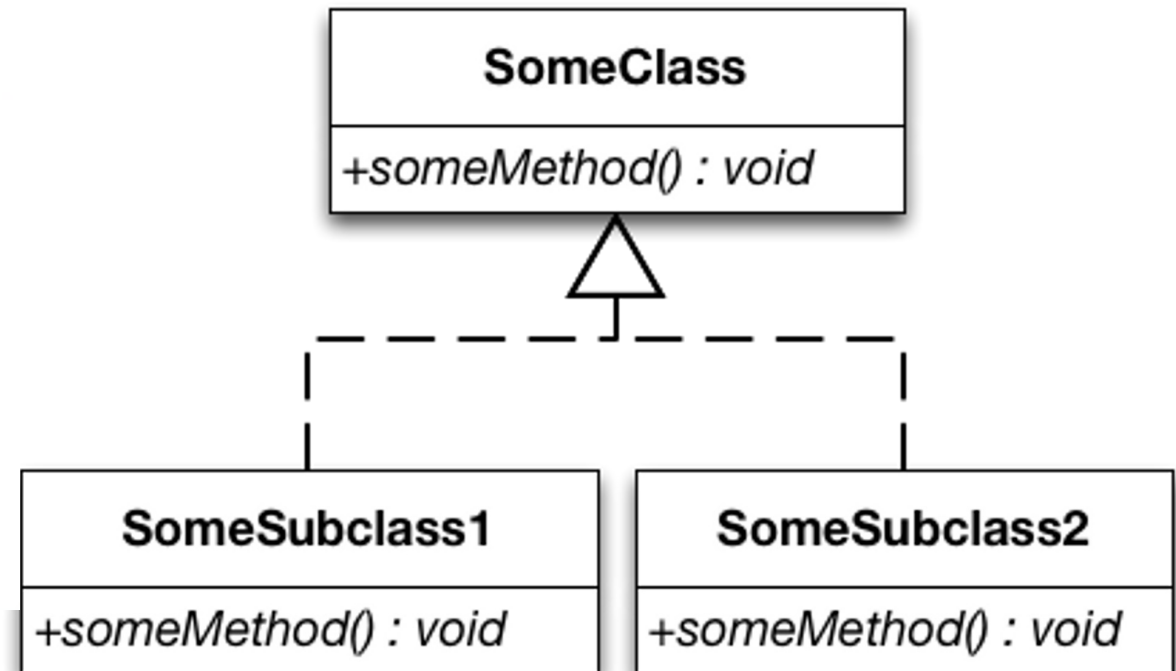
What happens if we don't use the LSP?

- We may create subclasses that behave differently than their superclasses, making it difficult to use them interchangeably.
- This can lead to unexpected behaviors and bugs in our software systems, making it harder to maintain and evolve our software systems over time.

Substitutability in object-oriented programs



```
1 void clientMethod(SomeClass sc) {  
2     // ...  
3     sc.someMethod();  
4     // ...  
5 }
```



Substitutability in object-oriented programs cont.

- In object-oriented programs, subclasses are substitutable for superclasses in client code: In `clientMethod`, `sc` may be an instance of `SomeClass` or any of its subclasses.
- Hence, if `clientMethod` works with instances of `SomeClass`, it does so with instances of any subclass of `SomeClass`. They provide all methods of `SomeClass` and eventually more.

LSP Violation

A method in a subclass break a superclass method's contract in several possible ways:

- Returning an object that's incompatible with the object returned by the superclass method.
- Throwing a new exception that's not thrown by the superclass method.
- Changing the semantics or introducing side effects that are not part of the superclass's contract.

Code Example 1: Bird Class



```
1  class Bird {  
2      public void fly() {  
3          // code to fly  
4      }  
5  }  
6  
7  class Ostrich extends Bird {  
8      public void fly() {  
9          throw new UnsupportedOperationException();  
10     }  
11 }
```


Code Example 2: Square



```
1  class Rectangle {
2      protected int width;
3      protected int height;
4
5      public void setWidth(int width) {
6          this.width = width;
7      }
8
9      public void setHeight(int height) {
10         this.height = height;
11     }
12
13     public int getArea() {
14         return width * height;
15     }
16 }
17
```

```
18 class Square extends Rectangle {
19     public void setWidth(int width) {
20         super.setWidth(width);
21         super.setHeight(width);
22     }
23
24     public void setHeight(int height) {
25         super.setWidth(height);
26         super.setHeight(height);
27     }
28 }
```

Interface Segregation Principle (ISP)

- Clients should not be forced to implement interfaces it does not use.
 - ISP splits interfaces that are very large into smaller and more specific ones so that clients will only have to know about the methods that are need.
- Many client-specific interfaces are better than one general-purpose interface.
 - When we have non-cohesive interfaces, the ISP guides us to create multiple, smaller, cohesive interfaces.

Why we need the ISP

- The ISP helps us write more maintainable code by avoiding the creation of overly complex classes that must implement a large number of methods they do not use.
- When we create smaller, more focused interfaces, classes can be more specific and focused in their responsibilities, making them easier to understand and modify.

What happens if we don't use the ISP?

- We may end up with overly complex classes that are difficult to understand and maintain.
- These classes may also be harder to test, as they may have many methods that are not relevant to their specific responsibilities.
- If we create overly general interfaces, classes may be forced to implement methods they do not use, leading to a large amount of unnecessary code.

Code Example

General-Purpose Interface

```
1  interface Vehicle {
2      void startEngine();
3      void stopEngine();
4      void drive();
5      void park();
6  }
7
8  class Car implements Vehicle {
9      public void startEngine() { /* code to start engine */ }
10     public void stopEngine() { /* code to stop engine */ }
11     public void drive() { /* code to drive */ }
12     public void park() { /* code to park */ }
13 }
14
15 class Bicycle implements Vehicle {
16     public void startEngine() { /* code to start engine */ }
17     public void stopEngine() { /* code to stop engine */ }
18     public void drive() { /* code to drive */ }
19     public void park() { /* code to park */ }
20 }
```

Code Example

A Better Design using ISP

```
1  interface EngineVehicle {
2      void startEngine();
3      void stopEngine();
4  }
5
6  interface DriveVehicle {
7      void drive();
8  }
9
10 interface ParkVehicle {
11     void park();
12 }
13
14 class Car implements EngineVehicle, DriveVehicle, ParkVehicle {
15     public void startEngine() { /* code to start engine */ }
16     public void stopEngine() { /* code to stop engine */ }
17     public void drive() { /* code to drive */ }
18     public void park() { /* code to park */ }
19 }
20
21 class Bicycle implements DriveVehicle, ParkVehicle {
22     public void drive() { /* code to drive */ }
23     public void park() { /* code to park */ }
24 }
```

Dependency Inversion Principle (DIP)

- It states that high-level modules should not depend on low-level modules. Both should depend on abstractions.
- Every dependency in the design should target an interface or an abstract class. No dependency should target a concrete class.

Why we need the DIP

- Decouples code, making it easier to change and maintain
- Increases code reuse
- Improves application scalability
- Promotes writing code that is easier to test

What happens if we don't use the DIP?

- Increases coupling between components
- Makes code harder to change and maintain
- Reduces code reuse
- Decreases scalability
- Makes code harder to test

Code Example 1: Without DIP



```
1  class EmailService {
2      public void sendEmail(String message) { /* code to send email */ }
3  }
4
5  class UserController {
6      private EmailService emailService = new EmailService();
7
8      public void registerUser(String name, String email) {
9          // code to register user
10         emailService.sendEmail("Welcome to our site, " + name);
11     }
12 }
```

Code Example 1: With DIP



```
1 interface EmailService {
2     void sendEmail(String message);
3 }
4
5 class SMTPEmailService implements EmailService {
6     public void sendEmail(String message) { /* code to send email */ }
7 }
8
9 class UserController {
10     private EmailService emailService;
11
12     public UserController(EmailService emailService) {
13         this.emailService = emailService;
14     }
15
16     public void registerUser(String name, String email) {
17         // code to register user
18         emailService.sendEmail("Welcome to our site, " + name);
19     }
20 }
```

Code Example 2:

```
1 interface DataAccess {
2     List<String> getData();
3 }
4
5 class MySQLDataAccess implements DataAccess {
6     public List<String> getData() { /* code to get data from MySQL */ }
7 }
8
9 class MongoDBDataAccess implements DataAccess {
10     public List<String> getData() { /* code to get data from MongoDB */ }
11 }
12
13 class BusinessLogic {
14     private DataAccess dataAccess;
15
16     public BusinessLogic(DataAccess dataAccess) {
17         this.dataAccess = dataAccess;
18     }
19
20     public List<String> getData() {
21         return dataAccess.getData();
22     }
23 }
```