



SOLID

ПРИНЦИПИ

Single responsibility principle

Open/Closed principle

Liskov Substitution principle

Interface Segregation principle

Dependency Inversion principle

Для чого це потрібно?

SOLID-принципи потрібні для того, щоб писати чистий, зрозумілий і підтримуваний код. Вони допомагають проєктувати програмне забезпечення так, щоб воно легко масштабувалося, змінювалося без помилок і не ламало вже працюючий функціонал.

Трохи про розробника SOLID принципів



'Your code is bullshit!'

Роберт (*Uncle Bob*) Мартін - відомий програміст, автор і один із засновників Agile, провідник ідей чистого коду та чистої архітектури. Він сформулював ці принципи як підхід до створення гнучкого, й легко масштабованого коду в парадигмі ООП.

Single Responsibility Principle (SRP)
(Принцип єдиної відповідальності)

Кожен клас або модуль має відповідати лише за одну задачу чи функцію. Якщо клас робить забагато речей, його стає складно змінювати чи тестувати.

Приклад C#

Приклад C++

Open/Closed Principle (OCP)
(Принцип відкритості/закритості)

Класи мають бути відкритими для розширення, але закритими для модифікації. Тобто ви можете додавати нові можливості, не змінюючи старий код.

Приклад C#

Приклад C++

Liskov Substitution Principle (LSP)

(Принцип підстановки Лісков)

"Підкласи повинні бути замінними на своїх батьківських класах без зміни коректності програми."

Простими словами:

- Якщо у вас є клас **БАТЬКО**, то його підклас **СИН** має поводитися так, щоб його можна було використовувати замість Батька без несподіваних помилок.
 - Клієнтський код (функції, що використовують клас) не повинен знати, чи працює він із Батьком, чи із Сином.
-

Приклад із життя:

Є Інтерфейс "Птах" з методом Літати(). Якщо створити підклас "Пінгвін", який не вміє літати, це порушить LSP, бо пінгвіна не можна використовувати замість птаха.

Bad Practice:

C#

```
1 class Bird {  
    2 references  
2     public virtual void Fly() {  
3         Console.WriteLine("Flying...");  
4     }  
5 }  
6  
0 references  
7 class Penguin : Bird { // Пінгвін не вміє літати!  
    2 references  
8     public override void Fly() {  
9         throw new NotImplementedException("Penguins can't fly!");  
10    }  
11 }  
12  
13 // Клієнтський код  
14 void MakeBirdFly(Bird bird) {  
15     bird.Fly(); // Якщо передати Penguin – вилетить помилка!  
16 }
```

Kotlin

```
1 open class Bird {  
2     open fun fly() {  
3         println("Flying...")  
4     }  
5 }  
6  
7 class Penguin : Bird() {  
8     override fun fly() {  
9         throw UnsupportedOperationException("Penguins can't fly!")  
10    }  
11 }  
12  
13 // Клієнтський код  
14 fun makeBirdFly(bird: Bird) {  
15     bird.fly() // Помилка для пінгвіна!  
16 }
```

Чому це погано?

- Клас Penguin порушує LSP, бо не може замінити Bird без збоїв.
- Клієнтський код має знати, чи це Penguin, щоб уникнути винятків.

Good Practice:

1 interface IFlyable

2 {
3 | 3 references
4 | void Fly();
5 }

6 public class Bird
7 {

8 | 3 references
9 | public virtual void MakeSound()
10 | {
11 | | Console.WriteLine("Some bird sound");
12 | }
13 }

14 // Пінгвін - це птах, але не вмie літати

15 public class Penguin : Bird
16 {

17 | 0 references
18 | public void Swim()
19 | {
20 | | Console.WriteLine("Penguin swimming");
21 | }
22 }

23 | 2 references
24 | public override void MakeSound()
25 | {
26 | | Console.WriteLine("Penguin sound");
27 | }
28 }

28 // Панага - це птах, який вмie літати

29 public class Parrot : Bird, IFlyable
30 {

31 | 3 references
32 | public void Fly()
33 | {
34 | | Console.WriteLine("Parrot flying");
35 | }
36 }

37 | 2 references
38 | public override void MakeSound()
39 | {
40 | | Console.WriteLine("Parrot sound");
41 | }
42 }

0 references

42 class Program

43 {

44 | // Клієнтський код, який працює з птахами

45 | 2 references
46 | static void InteractWithBird(Bird bird)

47 | {
48 | | bird.MakeSound();
49 |
50 | | // Можна додатково перевірити, чи птах вмie літати
51 | | if (bird is IFlyable flyableBird)
52 | | {
53 | | | flyableBird.Fly();
54 | | }
55 | }
56 }

56 | // Клієнтський код, який працює з літаючими об'єктами

57 | 1 reference
58 | static void MakeFly(IFlyable flyable)
59 | {
60 | | flyable.Fly();
61 | }
62 }

0 references

62 static void Main(string[] args)

63 {
64 | var penguin = new Penguin();
65 | var parrot = new Parrot();
66 |
67 | // Працюємо з птахами
68 | InteractWithBird(penguin); // Виведе: Penguin sound
69 | InteractWithBird(parrot); // Виведе: Parrot sound та Parrot
70 | flying
71 |
72 | // Працюємо тільки з літаючими об'єктами
73 | MakeFly(parrot); // Виведе: Parrot flying
74 | // MakeFly(penguin); // Не скомпілюється, бо пінгвін не
75 | реалізує IFlyable
76 | }
77 }

C#

```
1 interface IFlyable {
2     fun fly()
3 }
4
5 open class Bird {
6     open fun makeSound() {
7         println("Some bird sound")
8     }
9 }
10
11 // Пінгвін - це птах, але не вмiє літати
12 class Penguin : Bird() {
13     fun swim() {
14         println("Penguin swimming")
15     }
16
17     override fun makeSound() {
18         println("Penguin sound")
19     }
20 }
21
22 // Папуга - це птах, який вмiє літати
23 class Parrot : Bird(), IFlyable {
24     override fun fly() {
25         println("Parrot flying")
26     }
27
28     override fun makeSound() {
29         println("Parrot sound")
30     }
31 }
```

```
1 interface IFlyable {
2     fun fly()
3 }
4
5 open class Bird {
6     open fun makeSound() {
7         println("Some bird sound")
8     }
9 }
10
11 // Пінгвін - це птах, але не вмiє літати
12 class Penguin : Bird() {
13     fun swim() {
14         println("Penguin swimming")
15     }
16
17     override fun makeSound() {
18         println("Penguin sound")
19     }
20 }
21
22 // Папуга - це птах, який вмiє літати
23 class Parrot : Bird(), IFlyable {
24     override fun fly() {
25         println("Parrot flying")
26     }
27
28     override fun makeSound() {
29         println("Parrot sound")
30     }
31 }
```

Kotlin

Interface Segregation Principle (ISP) (Принцип розділення інтерфейсів)

"Краще багато спеціалізованих інтерфейсів, ніж один універсальний."

Простими словами:

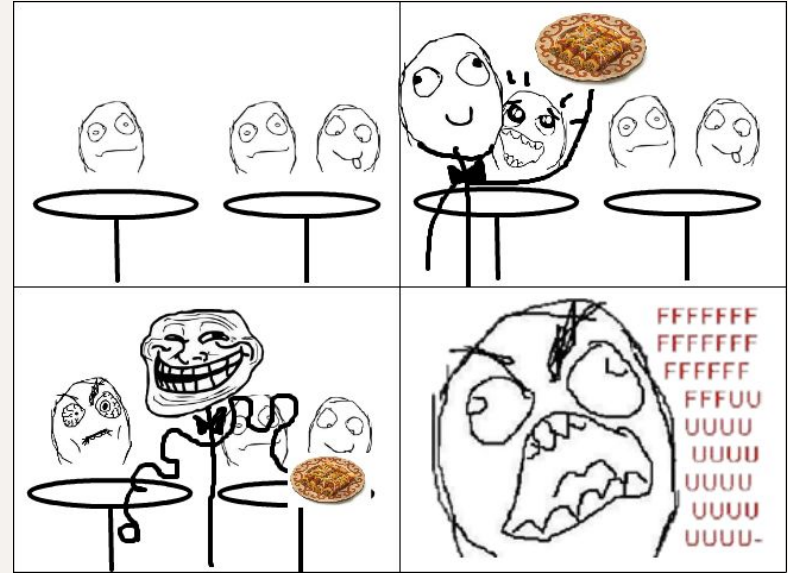
- Клієнти (класи) не повинні залежати від методів, які вони не використовують.
 - Великий "жирний" інтерфейс треба розбивати на дрібніші, щоб класи реалізовували лише те, що їм потрібно.
-

Приклад із життя:

У кафе є інтерфейс `IWorker`, який змушує всіх працівників вміти:

- Готувати їжу (`Cook`)
- Обслуговувати клієнтів (`Serve`)
- Мити посуд (`WashDishes`)

Проблема: Офіціант не повинен готувати, а кухар – мити посуд!



Bad Practice

```
1 public interface IWorker {  
    1 reference  
2     void Cook();  
    1 reference  
3     void Serve();  
    1 reference  
4     void WashDishes();  
5 }  
6  
0 references  
7 public class Waiter : IWorker {  
    1 reference  
8     public void Cook() { /* Не потрібно! */ }  
    1 reference  
9     public void Serve() { /* Так */ }  
    1 reference  
10    public void WashDishes() { /* Інколи ;) */ }
```

C#

```
1 interface IWorker {  
2     fun cook()  
3     fun serve()  
4     fun washDishes()  
5 }  
6  
7 class Waiter : IWorker {  
8     override fun cook() { /* не треба */ }  
9     override fun serve() { /* Так */ }  
10    override fun washDishes() { /* Інколи */ }  
11 }
```

Kotlin

Чому це погано?

Класи реалізують непотрібні методи (порушення ISP).

Зміна одного методу в інтерфейсі змусить змінювати всі класи.

Рішення: розділити інтерфейси!

Good Practice:

```
1 public interface IServe {  
    1 reference  
2     void Serve();  
3 }  
4  
0 references  
5 public interface ICook {  
    0 references  
6     void Cook();  
7 }  
8  
0 references  
9 public class Waiter : IServe {  
    1 reference  
10     public void Serve() { /* Тільки свої функції */ }  
11 }
```

C#	<pre>1 interface IServe { 2 fun serve() 3 } 4 5 interface ICook { 6 fun cook() 7 } 8 9 class Waiter : IServe { 10 override fun serve() { /* Тільки свої функції */ } 11 }</pre>	Kotlin
-----------	---	---------------

Dependency Inversion Principle (DIP)

(Принцип інверсії залежностей)

"Високорівневі модулі не повинні залежати від низькорівневих. Обидва повинні залежати від абстракцій (інтерфейсів або абстрактних класів)."

Простими словами:

1. Не пиши код, який безпосередньо залежить від конкретних класів (наприклад, `MySQLDatabase`, `FileLogger`).
2. Замість цього, створи інтерфейс (наприклад, `IDatabase`, `ILogger`) і змусь свій клас працювати через нього.
3. Так код стане гнучкішим: можна легко змінити реалізацію, не переписуючи весь додаток.

Приклад із життя: у програмі є клас `ReportService`, який безпосередньо використовує клас `MySQLDatabase` для збереження даних.

Bad Practice:

```
2 references
1 public class MySQLDatabase {           C#
    1 reference
2     public void SaveData(string data) {
3         Console.WriteLine($"Зберігаю
4             дані в MySQL: {data}");
5     }
6
7 public class ReportService {
    1 reference
8     private MySQLDatabase _database =
9         new MySQLDatabase(); // Пряма
10        залежність! Погано!
11
12     public void GenerateReport() {
13         _database.SaveData("Звіт 2024");
14     }
15 }
```

```
1 class MySQLDatabase {
2     fun saveData(data: String) {      Kotlin
3         println("Зберігаю дані в MySQL:
4             $data")
5     }
6
7 class ReportService {
8     private val database = MySQLDatabase
9         () // Пряма залежність! Погано!
10
11     fun generateReport() {
12         database.saveData("Звіт 2024")
13     }
14
15     public void GenerateReport() {
16         _database.SaveData("Звіт 2024");
17     }
18 }
```

Чому це погано?

Якщо ми захочемо використовувати PostgreSQL або FileStorage, доведеться переписувати ReportService.

Код жорстко залежить від конкретної реалізації.

Рішення: абстракція!

Good Practice:

```
1 // easy-expandable code
2 public interface IDatabase {
3     void SaveData(string data);
4 }
5
6 public class MySQLDatabase : IDatabase {
7     public void SaveData(string data) {
8         Console.WriteLine($"Звіт в MySQL: {data}");
9     }
10 }
11
12 public class PostgreSQLDatabase : IDatabase {
13     public void SaveData(string data) {
14         Console.WriteLine($"Звіт в PostgreSQL: {data}");
15     }
16 }
17
18 public class FileStorage : IDatabase {
19     public void SaveData(string data) {
20         File.WriteAllText("report.txt", data);
21         Console.WriteLine($"Звіт в файл: {data}");
22     }
23 }
```

```
25 public class ReportService {
26     private readonly IDatabase _database;
27
28     public ReportService(IDatabase database) { // Залежність через інтерфейс!
29         _database = database;
30     }
31
32     public void GenerateReport() {
33         _database.SaveData("Звіт 2024");
34     }
35 }
36
37
38 // dependency injection
39 var mySqlReport = new ReportService(new MySQLDatabase());
40 mySqlReport.GenerateReport(); // "Звіт в MySQL: Звіт 2024"
41
42 var postgresReport = new ReportService(new PostgreSQLDatabase());
43 postgresReport.GenerateReport(); // "Звіт в PostgreSQL: Звіт 2024"
44
45 var fileReport = new ReportService(new FileStorage());
46 fileReport.GenerateReport(); // Звіт в файл "report.txt"
```

C#

Ін'єкція залежностей - реалізація DIP на практиці

```
1 interface Database {
2     fun saveData(data: String)
3 }
4
5 class MySQLDatabase : Database {
6     override fun saveData(data: String) {
7         println("Звіт в MySQL: $data")
8     }
9 }
10
11 class PostgreSQLDatabase : Database {
12     override fun saveData(data: String) {
13         println("Звіт в PostgreSQL: $data")
14     }
15 }
16
17 class FileStorage : Database {
18     override fun saveData(data: String) {
19         File("report.txt").writeText(data)
20         println("Звіт в файл: $data")
21     }
22 }
```

```
24 class ReportService(private val database: Database) { // Залежність через
    інтерфейс!
25     fun generateReport() {
26         database.saveData("Звіт 2024")
27     }
28 }
29
30
31
32 // dependency injection
33 fun main() {
34     val mySqlReport = ReportService(MySQLDatabase())
35     mySqlReport.generateReport() // "Звіт в MySQL: Звіт 2024"
36
37     val postgresReport = ReportService(PostgreSQLDatabase())
38     postgresReport.generateReport() // "Звіт в PostgreSQL: Звіт 2024"
39
40     val fileReport = ReportService(FileStorage())
41     fileReport.generateReport() // Звіт у файл "report.txt"
42 }
```

Kotlin

Чому SOLID корисний?

SOLID корисний, тому що він допомагає створювати гнучкий, зрозумілий і підтримуваний код, який легко масштабувати та модифікувати. Ось основні переваги цих принципів:

1. Покращує якість коду

- Зменшує зв'язність (low coupling) – модулі залежать від абстракцій, а не від конкретних реалізацій.
- Підвищує зв'язність (high cohesion) – кожен клас/метод виконує одну чітку роль.
- Робить код передбачуваним – легше знайти помилки та внести зміни.

2. Спрощує підтримку та розширення

- Open/Closed Principle (OCP) – новий функціонал додається через розширення, а не зміну існуючого коду.
- Dependency Inversion (DIP) – код залежить від інтерфейсів, тому заміна реалізацій (наприклад, для тестування) стає простою.

3. Полегшує тестування

- Легкі моки та заглушки – завдяки інтерфейсам і DI.
- Менше залежностей = простіші юніт-тести.

4. Допомогає уникнути поширених антипатернів

- God Object (клас, що робить усе) – розбивається за Single Responsibility.
- Fragile Base Class (крихка база) – уникнення через Liskov Substitution.
- Interface Pollution (інтерфейси з тисячами методів) – запобігає Interface Segregation.

5. Підходить для великих проектів

У корпоративному ПЗ (наприклад, банківські системи, enterprise-рішення) SOLID критично важливий, бо:

- Дозволяє розподіляти роботу між командами.
- Зменшує ризик поломки коду при змінах.

Коли SOLID може бути "некорисним"?

- У дрібних скриптах або прототипах – надмірна архітектура лише ускладнить розробку.
- Можна переборщити з абстракцією.
- Нові парадигми – Функціональне програмування, композиція над успадкуванням (як у Go або Rust) роблять деякі принципи (наприклад, LSP) менш актуальними.
- Мікросервіси та серверлес – У розподілених системах акцент зміщується на незалежні компоненти, а не на класичні ООП-структури.
- Сучасні мови та фреймворки – Деякі мови (наприклад, Kotlin, TypeScript) пропонують альтернативні підходи до побудови архітектури (лямбда-типи, extention funcs, делегування замість успадковування, data-classes...)

Підсумки:

1. Один клас - одна відповідальність
2. Додавати новий функціонал через нові класи, а не зміни в старих.
3. Підкласи можна замінити на батьківські (нащадки не повинні ламати поведінку батьківського)
4. Один інтерфейс - один набір дій
5. Ін'єкція залежностей - Залежності мають бути абстрактними (інтерфейси), а не конкретними реалізаціями.

Що почитати?

Загалом:

1. Іван Бранець, dou.ua([посилання](#))
2. Сергій Немчинський, dou.ua([посилання](#))


С#:

3. Damodara Naidu, c-sharpcorner.com([посилання](#))
4. Danny, freecodecamp.org([посилання](#))

С++:

5. Олександра Шершень, medium.com([посилання](#))
6. Samuel Oloruntoba та Anish Singh Walia, digitalocean.com([посилання](#))

Kotlin:

7. xpendence , понятнее про SOLID ([link](#))
8. Huawei Developers ,  Kotlin SOLID Principles([link](#))