SOLID Principles with Kotlin Examples

Index

1. Introduction

If you are familiar with Object-Oriented Programming,then you've probably heard about the SOLID principles.

The SOLID Principles are five principles of Object-Oriented class design.

They are a set of rules and best practices to follow while designing a class structure.

2. What Are SOLID Principles?

Basically, SOLID is an acronym for five design principles,
which main goal is to make software designs **easier to read, maintain** and work with.
It's been introduced around 2004, by Michael Feathers. Nevertheless,
it's just a subset of many principles promoted by Robert C. Martin, also known as an **"Uncle Bob"**.

2.1 Single-Responsibility Principle

2.2 Open-Closed Principle

2.3 Liskov Substitution Principle

2.4 Interface Segregation Principle

2.5 Dependency Inversion Principle

2.1 Single-Responsibility Principle

2.1.1 **Theory**

The Single-responsibility principle states:

A class should have only single responsibility.

In other words, a class should have only one reason to exists and moreover- be responsible for one thing. Although it seems pretty straightforward, the judgement itself is oftentimes really subjective and may vary between programmers.

2.1.2 **Violation Example**

```kotlin
/**
 *  SRP Violation
 */
class User(
    val id: Int,
    val name: String,
    val email: String,
    val phoneNumber: String
)


class AdminDashboardService {
    fun sendNotification(user: User) {
        println("Preparing email content")
        println("Sending email to ${user.email}")
    }

    fun deleteUser(user: User) {
        println("Deleting user with id ${user.id} from the database")
    }
}
```

For the purpose of simplicity, the above functions are just printing some text to the output. Nevertheless,
In the real-life scenarios the sendNotification() would be responsible for preparing an HTML content for the email and sending it to the given email address.

On the other hand, the deleteUser() would perform an SQL query deleting the record from connected database.

In such a case, we can clearly see that our service is responsible for 3 different things.

Moreover, let's imagine that:

1.The marketing team requested a change in the e-mail template because of the branding change
2.The CTO requested an email automation provider change
3.The data team requested a change in SQL query

We can clearly see that each of these requests may easily affect theoretically unrelated business functions.

2.1.3 **Solution**

```kotlin
/**
 * Solution
 */
class UserAccountService {
    fun deleteUser(user: User) {
        println("Deleting user with id ${user.id} from the database")
    }
}

class EmailContentProvider {
    fun prepareContent() {
        println("Preparing email content")
    }
}

class EmailNotificationService {
    fun sendNotification(user: User) {
        println("Sending email to ${user.email}")
    }
}
```

2.1.4 **Benefits**

Finally, let's summarize with the most important benefits that come with well designed, isolated classes with one responsibility:

- most importantly- any bug introduced to the particular class **affects less parts of the system** (and organization as a whole)
- additionally, the number of **merge conflicts** is reduced when multiple people are working with the codebase

- whatsoever, it can introduce much better **readability** than the monolithic classes

2.2 Open-Closed Principle

2.2.1 **Theory**

Let’s take a look at the open-closed principle:

> Software entities (classes, modules, functions, etc.) should be **open for extension, but closed for modification.**

2.2.2 **Example**

```
/**
 * Open close principle
 */
open class Parent {

    open fun walk() {

    }

    open fun run() {

    }
}

class Child : Parent() {

    override fun walk() {
        super.walk()
    }

    override fun run() {
        super.run()
    }
}
```

2.2.3 **Benefits**

Identically, let's see the most important advantages of the open-closed principle:

- first of all, **reusability** and **flexibility**. We can use already existing codebase to implement new features or apply changes without the need of reinventing the wheel
- moreover, the above advantage is a great **time-saver**
- additionally, modification of existing classes might introduce unwanted behavior everywhere they've been used. With the open-closed principle, we can easily **avoid this risk**

2.3 Liskov Substitution Principle

2.3.1 **Theory**

it should not **break the existing functionality.**
Rreplacement of the interface invocation with the derived method will definitely break the flow.

2.3.2 **Violation Example**

Let’s say that we’ve got two types of users in the app: standard and admin. Both types of the account can be created. Nevertheless, the admin account can not be deleted in our app (for instance it can be done only from an external one).

```kotlin
/**
 * Violation Example
 */
interface Manageable {
    fun create()
    fun delete()
}

class StandardUser : Manageable {
    override fun create() {
        println("Creating Standard User account")
    }

    override fun delete() {
        println("Deleting Standard User account")
    }
}

class AdminUser : Manageable {
    override fun create() {
        println("Creating Admin User account")
    }

    override fun delete() {
        throw RuntimeException("Admin Account can not be deleted!")
    }
}
```

This time, we've introduced more specific contract with two, separate interfaces. Definitely, the hypothetical substitution won't break the flow.

2.3.3 **Solution**

```kotlin
/**
 * Solution
 */


interface Creatable {
    fun create()
}

interface Deletable {
    fun delete()
}

class StandardUsers : Creatable, Deletable {
    override fun create() {
        println("Creating Standard User account")
    }

    override fun delete() {
        println("Deleting Standard User account")
    }
}

class AdminUsers : Creatable {
    override fun create() {
        println("Creating Admin User account")
    }
}
```

2.3.4 **Benefits**

Given the above, what does the Liskov substitution principle bring to the table?

- when our subtypes conform behaviorally to the supertypes in our code, our **code hierarchy becomes cleaner**
- furthermore, people working with the abstraction (interface in our case) can be sure that **no unexpected behavior occurs**

2.4 Interface Segregation Principle

2.4.1 **Theory**

Many client-specific interfaces are better than one general-purpose interface.

2.4.2 **Violation Example**

```kotlin
/**
 * Violation
 */
interface File {
    fun read()

    fun write()
}


class FileOperation : File {
    override fun write() {
    }

}
```

2.4.3 **Solution**

```
/**
 * Solution
 */
interface File {
    fun read(){

    }

    fun write()
}


class FileOperation : File {
    override fun write() {
    }

}
```

2.4.4 **Benefits**

Applying the interface segregation principle in our designs has plenty of perks. Let's check a few of them:

- well designed interfaces help us to **follow the other principles**. It's much easier to take care of single responsibility and as we could see- Liskov substitution

- additionally, precise contract described by the interface makes the cod**e less error-prone**
- whatsoever, it really **improves readability** of the hierarchy and the codebase itself

2.5 Dependency Inversion Principle

2.5.1 **Theory**

Depend upon abstractions, [not] concretions.

2.5.2 **Violation Example**

As we can see, the *EmailNotificationService* will send a formatted to upper case message. Although everything is working as expected, we can spot that this method depends on the specific implementation.

```
/**
 * Violation Example
 */

class EmailNotificationServices {
    fun sendEmail(message: String) {
        val formattedMessage = message.uppercase()
        println("Sending formatted message: $formattedMessage")
    }
}
```

2.5.3 **Solution**

```kotlin
/**
 * Solution
 */
class EmailNotificationServiceSolutions{
    fun sendEmail(message: String, formatter: (String) -> String) {
        val formattedMessage = formatter.invoke(message)
        println("Sending formatted message: $formattedMessage")
    }
}
```

This time, we made the EmailNotificationService independent of the formatter implementation. The only thing that this service care about is that the formatter has to return a String value. As we can see, applying this principle gives us much more flexibility.

2.5.4 **Benefits**

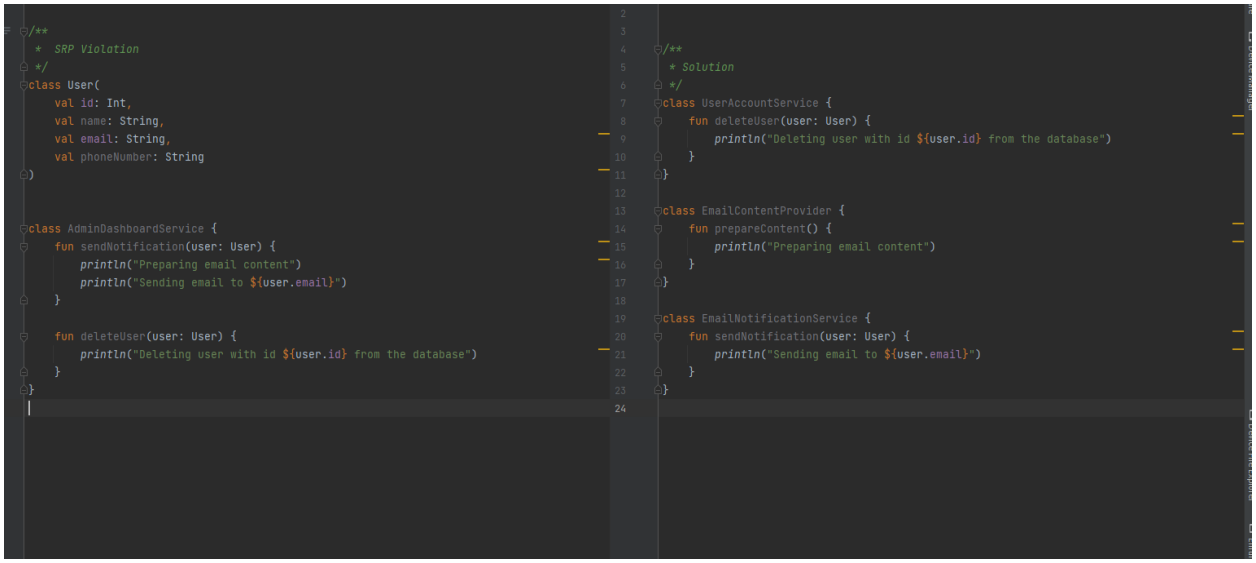
Finally, let's enumerate the dependency inversion principle benefits:
allows the codebase to be **easily expanded** and extended with new functionalities
furthermore, it improves **reusability**

3. Summary

And that would be all for this post covering the SOLID principles with Kotlin examples.

We really hope that this one will help you to understand these principles even better, no matter whether you are a beginner or an advanced programmer.

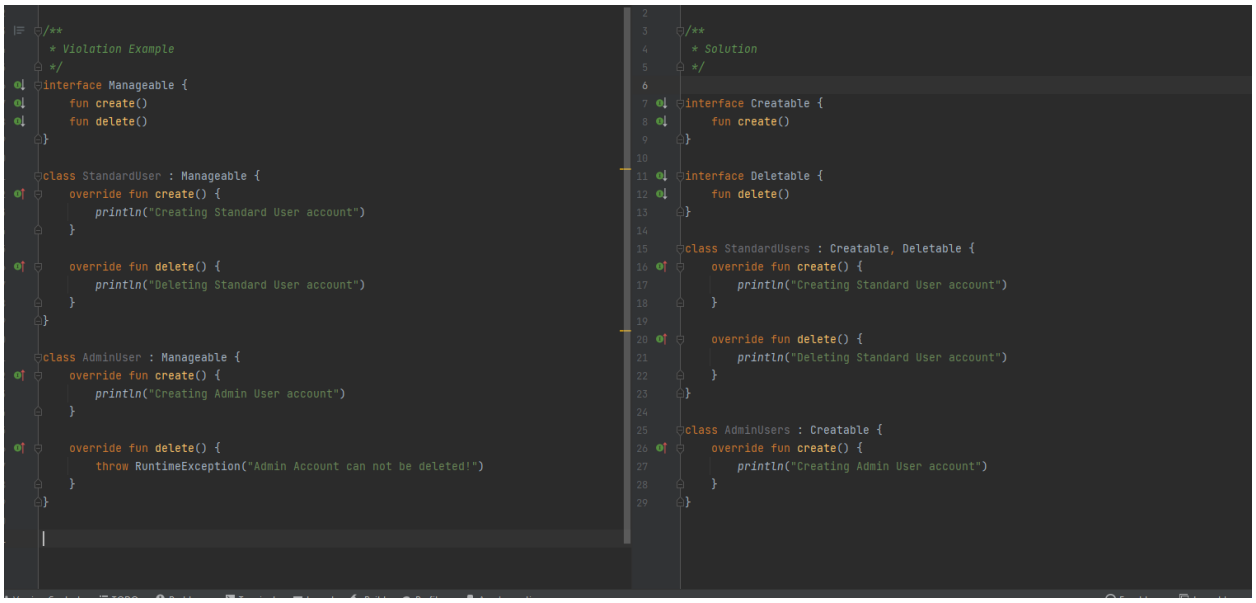
Single-Responsibility Principle Violation and Solution



```kotlin
/**
 *  SRP Violation
 */
class User(
    val id: Int,
    val name: String,
    val email: String,
    val phoneNumber: String
)

class AdminDashboardService {
    fun sendNotification(user: User) {
        println("Preparing email content")
        println("Sending email to ${user.email}")
    }

    fun deleteUser(user: User) {
        println("Deleting user with id ${user.id} from the database")
    }
}
```

```kotlin
/**
 * Solution
 */
class UserAccountService {
    fun deleteUser(user: User) {
        println("Deleting user with id ${user.id} from the database")
    }
}

class EmailContentProvider {
    fun prepareContent() {
        println("Preparing email content")
    }
}

class EmailNotificationService {
    fun sendNotification(user: User) {
        println("Sending email to ${user.email}")
    }
}
```

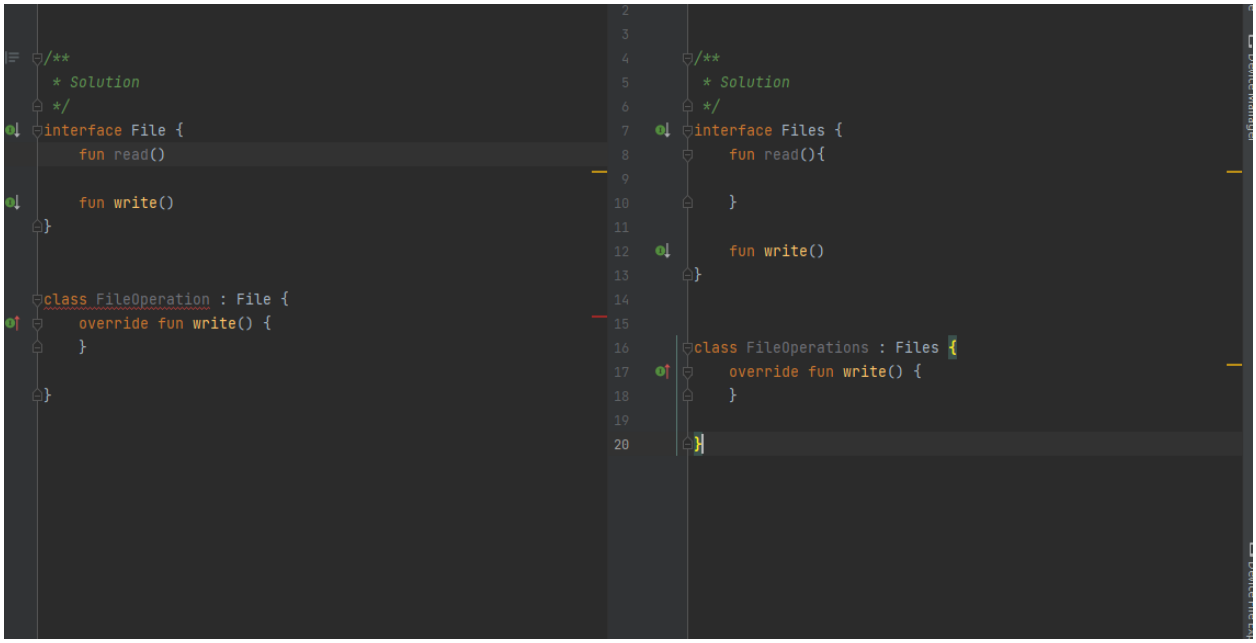
Liskov Substitution Violation and Solution



```kotlin
/**
 * Violation Example
 */
interface Manageable {
    fun create()
    fun delete()
}

class StandardUser : Manageable {
    override fun create() {
        println("Creating Standard User account")
    }

    override fun delete() {
        println("Deleting Standard User account")
    }
}

class AdminUser : Manageable {
    override fun create() {
        println("Creating Admin User account")
    }

    override fun delete() {
        throw RuntimeException("Admin Account can not be deleted!")
    }
}
```

```kotlin
/**
 * Solution
 */

interface Creatable {
    fun create()
}

interface Deletable {
    fun delete()
}

class StandardUsers : Creatable, Deletable {
    override fun create() {
        println("Creating Standard User account")
    }

    override fun delete() {
        println("Deleting Standard User account")
    }
}

class AdminUsers : Creatable {
    override fun create() {
        println("Creating Admin User account")
    }
}
```

Interface Segregation Violation and Solution



```kotlin
/**
 * Solution
 */
interface File {
    fun read()

    fun write()
}


class FileOperation : File {
    override fun write() {
    }

}
```

```kotlin
/**
 * Solution
 */
interface Files {
    fun read(){

    }

    fun write()
}


class FileOperations : Files {
    override fun write() {
    }

}
```

Dependency Inversion Violation and Solution

```kotlin
/**
 * Violation Example
 */

class EmailNotificationServices {
    fun sendEmail(message: String) {
        val formattedMessage = message.uppercase()
        println("Sending formatted message: $formattedMessage")
    }
}
```

```kotlin
/**
 * Solution
 */
class EmailNotificationServiceSolutions{
    fun sendEmail(message: String, formatter: (String) -> String
        val formattedMessage = formatter.invoke(message)
        println("Sending formatted message: $formattedMessage")
    }
}
```

References:

https://codersee.com/solid-principles-with-kotlin-examples/