S.O.L.I.D. Principles: Simplified Explanation & Example

Franz Andel · Follow

Published in Tunaiku Tech · 12 min read · Jun 4, 2020

Photo by Michele Bitetto on Unsplash

You guys might have heard about S.O.L.I.D. before, for those of you who still have something missing about this concept maybe because the explanations aren't telling the point or using a complicated wording. Or for those of you who are totally new and never heard of this before. Now you're on the right page because I'm going to give you the Simplified Version of this concept. Of course with the violation & correct example. Let's Start!

First of all, before jumping into each Principle's implementation, we need to understand what S.O.L.I.D. Principles is, why it is being used & when you are going to implement this. Also, one thing to note is that I'll consider you are familiar with OOP Concept because S.O.L.I.D. Principles come after OOP. Otherwise, please visit this link before going any further.

Prerequisite

Object-Oriented Programming (OOP)

What S.O.L.I.D. Principles is?

S.O.L.I.D. Principles is a Software Development Principle for programming with OOP Paradigm. These Principles were introduced by Robert C. Martin (Uncle Bob) in his paper *Design Principles and Design Patterns (2000)*.

Yes, you're not reading it wrongly nor I have a typo. It was written in 2000, 20 years ago since this article was published. But these Principles are still quite popular and are still being used widely in the world of OOP Paradigm.

Why S.O.L.I.D. Principles?

Because it makes your software :

- More Understandable
  When you come back to your code six months later, you still understand what you wrote back then.
- More Flexible
  It's quite easy when you need to add features in your code because all codes are loosely coupled.
- More Maintainable
  With the help of points 1 & 2, you will be easier in maintaining your code.

When S.O.L.I.D. Principles are needed?

> *Everytime you code with OOP Paradigm*

Meaning, these Principles are not tied to specific Programming Languages like Kotlin, Swift, Golang, etc. This is a Principles that you're going to have in your mind when you're writing codes that support OOP Paradigm.

Up until here, you should have enough knowledge about the theory of using S.O.L.I.D. Principles. Now it's about the time to dive into each Principle! Here I'll provide you the Simplified Version of each Principle & 2 Examples which consist of Violation and Correct Example.

The examples are in Kotlin. For those of you who have no experience in Kotlin, bear with me because I'm providing a general description & the most simple code as I can. Also, this is not about the language, it's more about the concept.

. . .

S. in S.O.L.I.D. Principles

First, let's start with what does this S. stands for :

S. = Single Responsibility Principle

> *From Wikipedia :*
> *The **single-responsibility principle** (**SRP**) is a computer-programming principle that states that every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class, module or function.*

Don't worry I'll give you the Simplified Version :

> *Simplified Version :*
> *A Class should only have 1 purpose.*

Yeah, that's all the point of the Single Responsibility Principle. Now I'll provide you the Example :

Violation Example :

```
class AgeCalculator {

    private val currentYear =
Calendar.getInstance().get(Calendar.YEAR)

    fun calculate(birthYear: Int): String {
        return (currentYear - birthYear).toString()
    }

    fun isValid(birthYear: Int): Boolean {
        return currentYear > birthYear
```

```
    }
}
```

From the highlighted code, we can see that in `AgeCalculator` class contains `isValid()` function which has different responsibility as what stated in the class name.

This is what violates the Single Responsibility Principle. Meaning, this class contains 2 functions with 2 responsibilities which are :

1. `calculate()` for calculating age
2. `isValid()` for validating age

So, what's the impact if you leave this code as it is?

Imagine this class grows, and have 10 functions which have 10 responsibility each. When you have got something to modify, you would go to that class. First, you will face 10 functions with their own responsibility which going to make you spend more time understanding the code. Second, you will violate the next principle which is Open-Closed Principle because you're modifying the existing class. Open-Closed Principle will be discussed after this principle.

So how do we refactor previous code to the correct one? Let's see

Correct Example :

- First, we left `calculate()` function in `AgeCalculator` class.

```
class AgeCalculator {
    fun calculate(birthYear: Int): String {
        val currentYear = Calendar.getInstance().get(Calendar.YEAR)
        return (currentYear - birthYear).toString()
    }
}
```

- Then, we move `isValid()` function to separate `AgeValidator` class.

```
class AgeValidator {
    fun isValid(birthYear: Int): Boolean {
        val currentYear = Calendar.getInstance().get(Calendar.YEAR)
        return currentYear > birthYear
    }
}
```

Here, we can see that each class has only 1 responsibility which would make us easier to understand the code and won't violate the next Open-Closed Principle.

O. in S.O.L.I.D. Principles

Let's start with what O. stands for :

O. = Open — Closed Principle

> *From Wikipedia :*
> *In object-oriented programming, the* ***open/closed principle*** *states "*software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*"; that is, such an entity can allow its behaviour to be extended without modifying its source code.*

Here's the Simplified Version :

> *Simplified Version :*
> *A Class should be Open for Extension but Closed for Modification.*

Maybe some of you might be wondering why Extension is preferred rather than Modification. Actually, this Principle is trying to prevent us from seeing too much code in a class.

Imagine you have a class with 1000 lines of code, and you need to edit something from there. You might have felt overwhelmed the moment you think of it and haven't even started editing the stuff that you're going to edit.

Please note that Closed for Modification here doesn't mean we totally may not edit stuff in classes, but the real meaning is, we can still edit stuff but with minimum changes. Here's the example :

Violation Example :

- First, we have `ShoesBrand` enum class.

```
enum class ShoesBrand {
    NIKE, ADIDAS
}
```

- Then, we have ShoesFactory class which have `getShoesPrice` function that accepts `ShoesBrand` as parameter.

```
class ShoesFactory {

    // How if there's a new brand of shoes?
    // If there are 10 ShoesBrand with Complex Calculation in it,
this class will be very Complex

fun getShoesPrice(shoesBrand: ShoesBrand): Int {
        return when (shoesBrand) {
            ShoesBrand.NIKE -> 2000
            ShoesBrand.ADIDAS -> 1500
        }
    }
}
```

- At last, we have another class which uses `ShoesFactory` as an Object. Then call the `getShoesPrice` method with `ShoesBrand.NIKE` as argument.

```
val shoesFactory = ShoesFactory()
tvViolationShoesType.text =
shoesFactory.getShoesPrice(ShoesBrand.NIKE).toString()
```

The highlighted code is prone to be edited. Imagine if your Project Manager comes and says, please update Nike price to 2500 and add another Puma brand with a price of 3000.

With the above code, you will modify the highlighted code and that's what violates this principle.

So how do we refactor previous code to the correct one? Let's see

Correct Example :

- First, We have `ShoesBrand` abstract class with `getShoesPrice` abstract function.

```
abstract class ShoesBrand {
    abstract fun getShoesPrice(): Int
}
```

- Then, we have `NikeShoes` class which extends from `ShoesBrand`.

```
class NikeShoes : ShoesBrand() {
    override fun getShoesPrice() = 2500
}
```



```
class AdidasShoes : ShoesBrand() {
    override fun getShoesPrice() = 1500
}
```

- Then, we have `PumaShoes` class which also extends from `ShoesBrand`.

```
class PumaShoes : ShoesBrand() {
    override fun getShoesPrice() = 3000
}
```

- At last, we have another class which uses `NikeShoes` as an Object with the type of `ShoesBrand`. Then we call `getShoesPrice` method to show the shoes

price into TextView.

```
val shoes: ShoesBrand = NikeShoes()
tvShoesType.text = shoes.getShoesPrice().toString()
```

Now, with this code, we can easily update Nike Price without bothering to see other brands of Shoes, and here is what I mean we still need to edit the class but with minimum changes.

Lastly, we can easily extend from `ShoesBrand` if there's a new Brand. Just like what we did in `PumaShoes` class.

L. in S.O.L.I.D. Principles

Here, we've arrived at Principle that causes brain explosion to most of us. But don't worry I'll give you the best-simplified version of this Principle. Let's start with what L. stands for

L. = Liskov Substitution Principle

> *From Wikipedia :*
> *In object-oriented programming, **behavioral subtyping** is the principle that subclasses should satisfy the expectations of clients accessing subclass objects through references of superclass type, not just as regards syntactic safety (absence of method-not-found errors and such) but also as regards behavioral correctness.*

I know a brain explosion there. Here's the Simplified Version :

> *Simplified Version :*
> *Abstraction should be able to provide all Child Class needs.*

If you are still confused with my simplified version definition, try to look at the example below first, then come back to this definition and hopefully things will start making sense.

Violation Example :

- We have `Jobs` interface with `doWork()` function.

```
interface Jobs {
    fun doWork()
}
```

- Then, we have `Programmer` class which implements `Jobs` interface.

```
class Programmer : Jobs {

    override fun doWork() {
        println("Working on Software Development Project")
    }
}
```

- Then, we have `Salesman` class which also implements `Jobs` interface.

```
class Salesman : Jobs {

    override fun doWork() {
        println("Working on Selling Goods")
    }
```

```
    fun talkToCustomer() {
        println("I talk to Customer Everyday")
    }
}
```

- Lastly, we have another class which uses `Programmer` as an Object with the type of `Jobs`. Then call the `doWork()` method.

```
val jobs: Jobs = Programmer()
jobs.doWork()
```

This is okay because `doWork()` method is supported by `Jobs` interface. So this does not violate the Principle yet.

So where's the Violating one? Please be patient, the violating part is about to come :

- Lastly, really the last one, the code is pretty much the same as before. We just replace `Salesman` class into `Programmer` class.

```
val jobs: Jobs = Salesman()
jobs.doWork()
if (jobs is Salesman) {
    jobs.talkToCustomer()
}
```

Here comes the violation part, pay attention to the highlighted code we check if `jobs` is `Salesman` then we cast it to `Salesman` and then call the `talkToCustomer()` method.

This is what Liskov Substitution Principle told us not to do. `Jobs` interface should be able to provide what is needed by the child class.

But there is something to note here, all things that are defined in an interface should be implemented correctly by the child class otherwise we'll violate the next principle which is Interface Segregation Principle. Interface Segregation Principle will be discussed after this principle.

Let's see how we refactor previous code into the correct one :

Correct Example :

- First, we have `Jobs` interface with `doWork()` and `talkToCustomer()` functions.

```
interface Jobs {
    fun doWork()
    fun talkToCustomer()
}
```

- Then, we have `Programmer` class which implements `Jobs` interface.

```
class Programmer : Jobs {

    override fun doWork() {
        println("Working on Software Development Project")
    }

    override fun talkToCustomer() {
        println("I seldom talk to Customer")
    }
}
```

- Then, we have `Salesman` class which also implements `Jobs` interface.

```
class Salesman : Jobs {

    override fun doWork() {
        println("Working on Selling Goods")
    }

    override fun talkToCustomer() {
        println("I talk to Customer Everyday")
    }
}
```

- Lastly, we have another class which uses `Salesman` as an Object with the type of `Jobs`. Then call the `doWork()` and `talkToCustomer()` methods.

```
val jobs: Jobs = Salesman()
jobs.doWork()
jobs.talkToCustomer()
```

Here you can see that `Jobs` interface has got `talkToCustomer()` function. So, in order to use `talkToCustomer()` method, there is no need for us to check if `jobs` is `Salesman` or not anymore.

Hopefully reaching this part, your brain is not exploding anymore.

I. in S.O.L.I.D. Principles

We've reached the fourth Principle, the second last one. Let's start with what I. stands for

I. = Interface Segregation Principle

> *From Wikipedia :*
> *In the field of software engineering, the* ***interface-segregation principle*** *(**ISP**) states that no client should be forced to depend on methods it does not use.*

The most simple Definition of Wikipedia so far. But here's the Simplified Version :

> *Simplified Version :*
> *Small Interface is better than Big Interface.*

This principle is the next stage after the Liskov Substitution Principle where functions that we define in abstraction should be implemented correctly. Meaning, we should not force child class to implement things that they can't do.

As we know, everything that is defined in Interface or abstract function should be implemented by the child class, otherwise, it would output a compiler error. So, we're encouraged to separate interface into a smaller one. Here's the example :

Violation Example :

- First, we have `OnClickListener` interface with `onClick()` and `onLongClick()` functions.

```
interface OnClickListener {
    fun onClick()
    fun onLongClick()
}
```

- Then, we have `ISViolationActivity` class which implements `OnClickListener` interface.

```
class ISViolationActivity : AppCompatActivity(), OnClickListener {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_interface_segregation)

        btnChangeName.setOnClickListener {
            onClick()
        }
    }

    @SuppressLint("SetTextI18n")
    override fun onClick() {
        tvName.text = "Franz Andel"
    }

    override fun onLongClick() {
        // This function is not used, but is forced to be implemented
    }
}
```

The interface consists of 2 functions in which if we only need one of the function inside that interface which is `onClick()` function, but we are forced to implement both of the functions. And this is what violates the principle. We should separate Interface into a smaller one so we can choose which to be used based on what we need.

Here's how we refactor previous code into the correct one :

Correct Example :

- First, we have `OnClickListener` interface with `onClick()` function.

```
interface OnClickListener {
    fun onClick()
}
```

- Then, we have `OnLongClickListener` interface with `onLongClick()` function.

```
interface OnLongClickListener {
    fun onLongClick()
}
```

- Lastly, we have `ISBestPracticeActivity` class which implements `OnClickListener` interface.

```
// Make a Smaller Interface, so Child Class can choose which
Interface to be Implemented

class ISBestPracticeActivity : AppCompatActivity(), OnClickListener {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_interface_segregation)

        btnChangeName.setOnClickListener {
            onClick()
        }
    }

    @SuppressLint("SetTextI18n")
    override fun onClick() {
        tvName.text = "Franz Andel"
    }
}
```

Now with this code, we can easily choose which interface that we need and then implement it.

In the example, we only need `onClick()` function so we only implement `OnClickListener` interface, but in the future, there might be a chance for you to need `onLongClick()` function too. This is simply solved by implementing both `OnClickListener` & `OnLongClickListener` interface. As you may know, in Java or Kotlin we can implement more than 1 interface.

D. in S.O.L.I.D. Principles

Finally, we've reached the last Principle. Let's start with what D. stands for

D. = Dependency Inversion Principle

> *From Wikipedia :*
> *In object-oriented design, the **dependency inversion principle** is a specific form of decoupling software modules.*

The shortest definition of Wikipedia at last. Here's the Simplified Version :

> *Simplified Version :*
> *A Class should depend on Abstraction, not Implementation.*

Let's get into the example to make things clearer :

Violation Example :

- First, we have `Dollar` class with `currency()` function.

```
class Dollar {
    fun currency() = "$"
```

```
}
```

- Then, we have `Rupiah` class with `moneyCurrency()` function.

```
class Rupiah {
    fun moneyCurrency() = "Rp."
}
```

- Lastly, we have `DIViolationActivity` class with a setter function `setConversion` that accepts `Dollar` class as parameter and to show currency to the TextView, we call `dollar.currency()` method.

```
class DIViolationActivity : AppCompatActivity() {

    // How if you want to change dollarViolation to RupiahViolation
    // while dollarViolation is used in many places?

    private lateinit var dollar: Dollar

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_dependency_inversion)

        setConversion(Dollar())
        setUI()
    }

    private fun setConversion(dollar: Dollar) {
        this.dollar = dollar
    }

    @SuppressLint("SetTextI18n")
    private fun setUI() {
        tvMyMoney.text = "${dollar.currency()}5.000"
    }
}
```

Here we have 2 implementation classes with their own currency. In order to call `currency()` method in `Dollar`, we make an object of it.

Now, imagine how if your Project Manager comes and says, please change the currency of our money into Rupiah.

With this code, you definitely need to change :

- the variable of `dollar` into `Rupiah`,
- `setConversion` argument into `Rupiah()`,
- `setConversion` function's parameter into `Rupiah`
- `dollar.currency()` into `rupiah.moneyCurrency()`.

We can conclude that there are 4 efforts there. And this is what violates the Principle.

Here's how we refactor the previous 4 efforts code into 1 effort code :

Correct Example :

- First, we have `MoneyConversion` interface with `currency()` function.

```
interface MoneyConversion {
    fun currency(): String
}
```

- Then, we have `Dollar` class which implements `MoneyConversion` interface.

```
class Dollar: MoneyConversion {
    override fun currency() = "$"
}
```

- Then, we have `Rupiah` class which also implements `MoneyConversion` interface.

```
class Rupiah: MoneyConversion {
    override fun currency() = "Rp."
}
```

- Lastly, we have `DIBestPracticeActivity` class with a setter function `setConversion` that accepts `MoneyConversion` interface as parameter and to show currency to the TextView, we call `moneyConversion.currency()` method.

```
class DIBestPracticeActivity : AppCompatActivity() {

    private lateinit var moneyConversion: MoneyConversion

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_dependency_inversion)

        // It can be easily changed to Rupiah
        setConversion(Dollar())
        setUI()
    }

    private fun setConversion(moneyConversion: MoneyConversion) {
        this.moneyConversion = moneyConversion
    }

    @SuppressLint("SetTextI18n")
    private fun setUI() {
        tvMyMoney.text = "${moneyConversion.currency()}5.000"
```

```
    }
}
```

Now, we've made `DIBestPracticeActivity` class depends on its Abstraction which is `MoneyConversion` interface.

With this code, we can easily change `setConversion()` argument to `Rupiah()`. And this is only 1 effort.

. . .

Conclusion

This concept will boost your productivity if implemented correctly because your code will be loosely coupled & easier to read.

If you need to look directly into the full source code. You can clone it from here :

franzandel/SOLID_Tutorial

The reason why we should follow SOLID Principles (With Violation & Best Practice Example) - franzandel/SOLID_Tutorial

github.com

I hope you get a better understanding of this concept after reading this. If you do have any suggestion please leave it in the comment section.

Solid Principles | Solid | Engineering | Oop

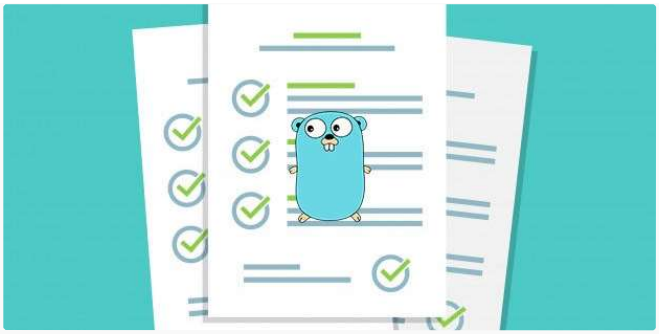
Written by Franz Andel

Follow

17 Followers · Writer for Tunaiku Tech

Technology Enthusiast who loves learning New Concept & Of course, apply it!

More from Franz Andel and Tunaiku Tech



Renaldi in Tunaiku Tech

Go Validator v10

Validation? Easy Peasy!

4 min read · Oct 20, 2020

261 2

Christian Ing Sunardi in Tunaiku Tech

Creating Secondary Entry Points for your Angular Library

Implement ng-packagr's secondary entry points

9 min read · Jun 4, 2020

550 6

Andre Simamora in Tunaiku Tech

Your First Web Component with Vue.js

Creating web component is complicated? Let's make it easier by using Vue.js

8 min read · Feb 18, 2020

500 5

Andre Simamora in Tunaiku Tech

Create Your First Web Component with Vanilla JavaScript

Creating a Web Component with frameworks such as Angular and Vue may look ordinary....

3 min read · Apr 2, 2020

277

See all from Franz Andel | See all from Tunaiku Tech

Recommended from Medium

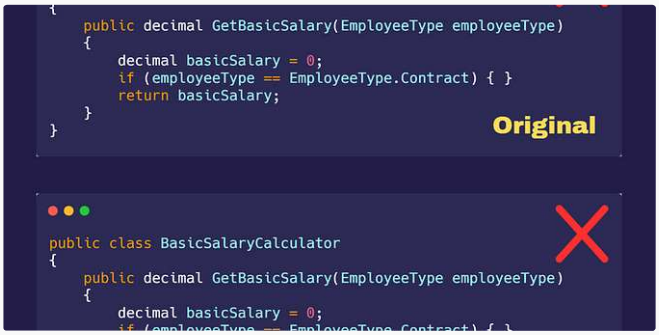
Srinivasan Baskar... in Cloudnloud Tech Communi...

Understanding the SOLID Principles with Real-time Examples

The SOLID principles are a set of five design principles that were introduced by Robert C....

4 min read · Oct 4, 2023

12



Muhammad Waseem in Become .NET Pro !

Pro EP 60 : Open/Closed Principle of SOLID

OCP is second principle of SOLID , most important one and it is linked with SRP.

2 min read · Aug 22, 2023

1

Lists

Leadership

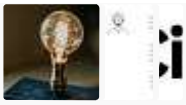
41 stories · 201 saves

Leadership upgrades

7 stories · 61 saves

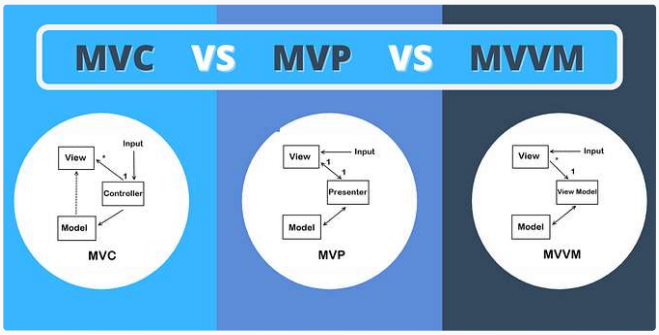
Stories to Help You Grow as a Software Developer

19 stories · 719 saves



Good Product Thinking

11 stories · 434 saves



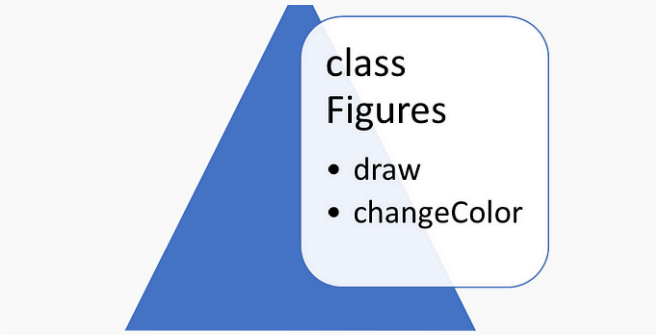
Gulshan N in Stackademic

Understanding MVC, MVVM, and MVP: A Comprehensive...

Introduction

4 min read · Oct 6, 2023

5

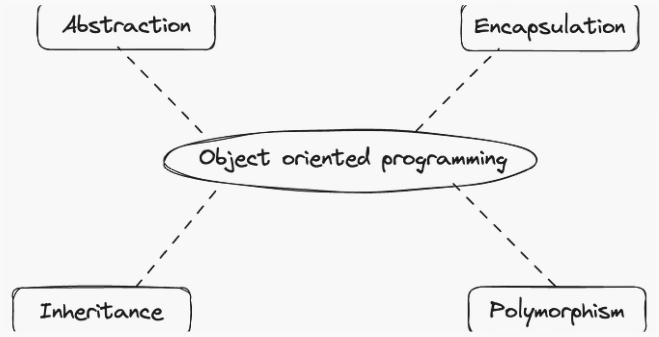


Ankit Pande

SOLID

S— Single Responsibility Principal

3 min read · Aug 21, 2023

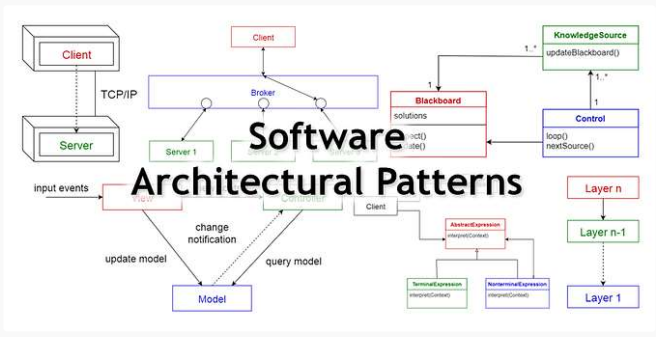


Dileep Sreepathi

OOPs Concepts C# — Object Oriented Programming

The better way to explain Object Oriented Programming is to think of the OOPs in real-...

5 min read · Jul 24, 2023



Vijini Mallawaarachchi in Towards Data Science

10 Common Software Architectural Patterns in a nutshell

Ever wondered how large enterprise scale systems are designed? Before major softwar...

5 min read · Sep 4, 2017

 33  1   39K  127 

See more recommendations