

SOLID Principles with Kotlin Examples

Index

- 1. Introduction**
- 2. What Are SOLID Principles?**
 - 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
- 3. Summary**

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 exist 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

```
/**
 * 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

```
/**
 * 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**.

Replacement 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).

```

/**
 * 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

```
/**
 * 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

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

    fun write()
}

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

2.4.3 Solution

```
4  |≡  /**
5     * Solution
6     */
7  o↓ interface File {
8     fun read(){
9
10    }
11
12    o↓ fun write()
13  }
14
15  o↑ class FileOperation : File {
16     override fun write() {
17     }
18  }
```

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 code **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

```
/**
 * 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

```
1  /**
2  * SRP Violation
3  */
4  class User {
5      val id: Int,
6      val name: String,
7      val email: String,
8      val phoneNumber: String
9  }
10
11 class AdminDashboardService {
12     fun sendNotification(user: User) {
13         println("Preparing email content")
14         println("Sending email to ${user.email}")
15     }
16
17     fun deleteUser(user: User) {
18         println("Deleting user with id ${user.id} from the database")
19     }
20 }
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

Liskov Substitution Violation and Solution

```
1  /**
2  * Violation Example
3  */
4  interface Manageable {
5      fun create()
6      fun delete()
7  }
8
9  class StandardUser : Manageable {
10     override fun create() {
11         println("Creating Standard User account")
12     }
13
14     override fun delete() {
15         println("Deleting Standard User account")
16     }
17 }
18
19 class AdminUser : Manageable {
20     override fun create() {
21         println("Creating Admin User account")
22     }
23
24     override fun delete() {
25         throw RuntimeException("Admin Account can not be deleted!")
26     }
27 }
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

Interface Segregation Violation and Solution

```
1  /**
2  * Solution
3  */
4  interface File {
5      fun read()
6      fun write()
7  }
8
9  class FileOperation : File {
10     override fun write() {
11     }
12 }
13
14
15
16
17
18
19
20
```

```
1  /**
2  * Solution
3  */
4  interface Files {
5      fun read(){}
6      fun write(){}
7  }
8
9  class FileOperations : Files {
10     override fun write() {
11     }
12 }
13
14
15
16
17
18
19
20
```

Dependency Inversion Violation and Solution

```
1  /**
2  * Violation Example
3  */
4  class EmailNotificationServices {
5      fun sendEmail(message: String) {
6          val formattedMessage = message.uppercase()
7          println("Sending formatted message: $formattedMessage")
8      }
9  }
10
11
12
```

```
1  /**
2  * Solution
3  */
4  class EmailNotificationServiceSolutions{
5      fun sendEmail(message: String, formatter: (String) -> String) {
6          val formattedMessage = formatter.invoke(message)
7          println("Sending formatted message: $formattedMessage")
8      }
9  }
10
11
12
```

References:

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

