

Clases Conceptuales: “Entity vs. Value Object”

- *Las Entidades* o *clases* del dominio de mi problema tienen un identificador, son modificables y comparables por *Identidad*.

Value Object:

- Son comparables por contenido (igualdad estructural), no tienen identificador.



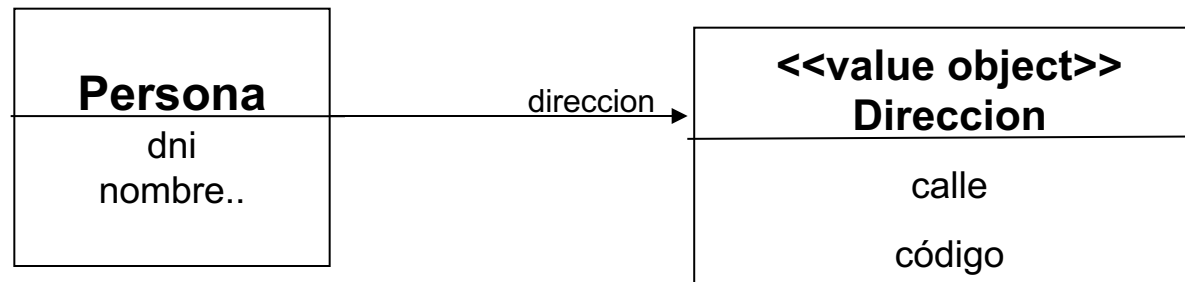
- No viven por si mismos, necesitan una entidad base, son intercambiables (un billete de 100 Pesos AR lo puedo cambiar por otro). Persisten adjunto a su base, no separadamente.
- Inmutables (No le defino *setters*).

Clases Conceptuales: “Entity vs. Value Object”

- Deben ser inmutables. No puede cambiar en su vida. Caso contrario NO es un Value Object.
- ¿Cómo identificar un *Value Object* en el modelo?

Cuando necesitamos por ej. modelar Moneda, Fecha, Dirección, que puedan tener cierto comportamiento (getters..)

¿Cómo representar “Value Objects” en el modelo?



<<Value Object>> delante del nombre

Ejemplo en Java

```
1 package vo;
2
3 import java.util.Objects;
4
5 public class DireccionPostal {
6
7     private final String calle;
8     private final String altura;
9     private final String codigoPostal;
10    private final String localidad;
11
12    public DireccionPostal(String calle, String altura,
13        String codigoPostal, String localidad) {
14        //pueden incluirse validaciones de los parámetros.
15        this.calle = calle;
16        this.altura = altura;
17        this.codigoPostal = codigoPostal;
18        this.localidad = localidad;
19    }
20
21    // Getters Dependiendo del contexto.
22    public String getCalle() {
23        return calle;
24    }
25 }
```

Ejemplo en Java

```
1 package vo;
2
3 import java.util.Objects;
4
5 public class DireccionPostal {
6
7     private final String calle;
8     private final String altura;
9     private final String codigoPostal;
10    private final String localidad;
11
12    public DireccionPostal(String calle, String altura, String codigoPostal, String localidad) {
13        //pueden incluirse valores nulos
14        this.calle = calle;
15        this.altura = altura;
16        this.codigoPostal = codigoPostal;
17        this.localidad = localidad;
18    }
19
20    // Getters Dependiendo del contexto.
21    public String getCalle() {
22        return calle;
23    }
24
25    @Override
26    public boolean equals(Object o) {
27        if (this == o) return true;
28        if (o == null || getClass() != o.getClass()) return false;
29        DireccionPostal that = (DireccionPostal) o;
30        return calle.equals(that.calle) &&
31            altura.equals(that.altura) &&
32            codigoPostal.equals(that.codigoPostal) &&
33            localidad.equals(that.localidad);
34    }
35
36    @Override
37    public int hashCode() {
38        return Objects.hash(calle, altura, codigoPostal, localidad);
39    }
40
41 }
```

Ejemplo en Java

```
1 package vo;
2
3 import static org.junit.jupiter.api.Assertions.*;
4
5 import org.junit.jupiter.api.BeforeEach;
6 import org.junit.jupiter.api.Test;
7
8 class DireccionPostalTest {
9
10     private DireccionPostal casa, home, biblioPublica;
11
12     @BeforeEach
13     public void setUp() {
14         this.casa = new DireccionPostal("50 y 120", "s/n", "1900", "La Plata");
15         this.home = new DireccionPostal("50 y 120", "s/n", "1900", "La Plata");
16         this.biblioPublica = new DireccionPostal("Plaza Rocha", "137", "1900", "La Plata");
17     }
18
19     @Test
20     void testEquals() {
21         assertEquals(this.casa, this.home);
22         assertNotEquals(this.casa, this.biblioPublica);
23     }
24
25 }
26
```

Agregando Heurísticas para Asignación de responsabilidades (HAR)

Descripción: cuando el comportamiento varía según el tipo, asigne la responsabilidad a los tipos/las clases para las que varía el comportamiento.

Ejemplo: Se debe soportar distintas bonificaciones de pago con tarjeta de crédito.
(ya visto previamente)

... Como la bonificación del pago varía según el tipo de tarjeta, deberíamos asignarle la responsabilidad de la bonificación a los distintos tipos de tarjeta.

- Nos permite sustituir objetos que tienen idéntica interfaz.

“No hables con extraños”

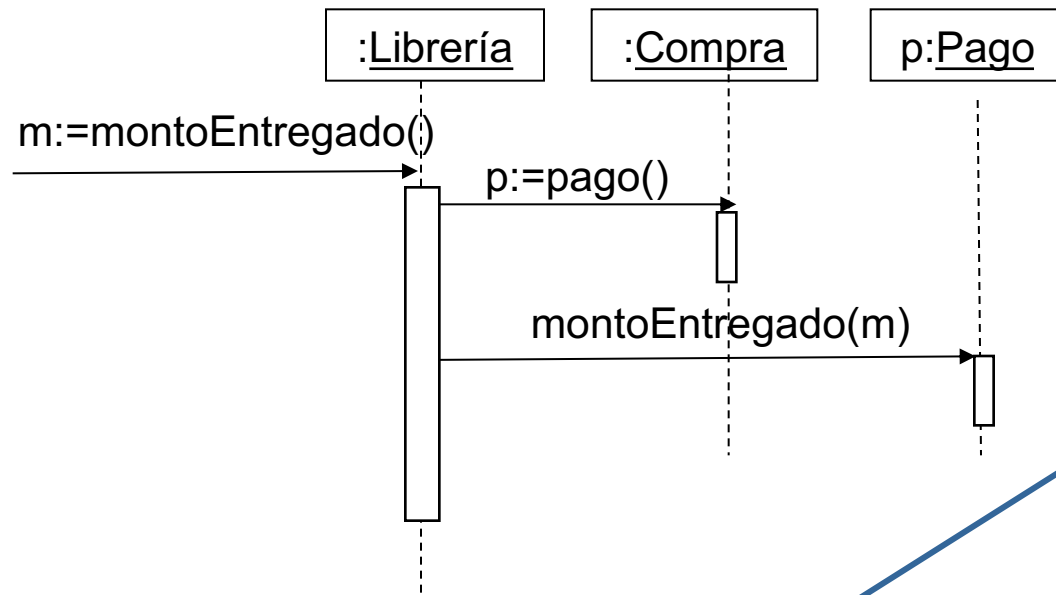
Descripción: Evite diseñar objetos que recorren largos caminos de estructura y envían mensajes (hablan) a objetos distantes o indirectos (extraños).

Dentro de un método sólo pueden enviarse mensajes a objetos conocidos:

- Self/this
- un parámetro del método
- un objeto que esté asociado a self/this
- un miembro de una colección que sea atributo de self/this
- un objeto creado dentro del método

Los demás objetos son extraños (strangers)

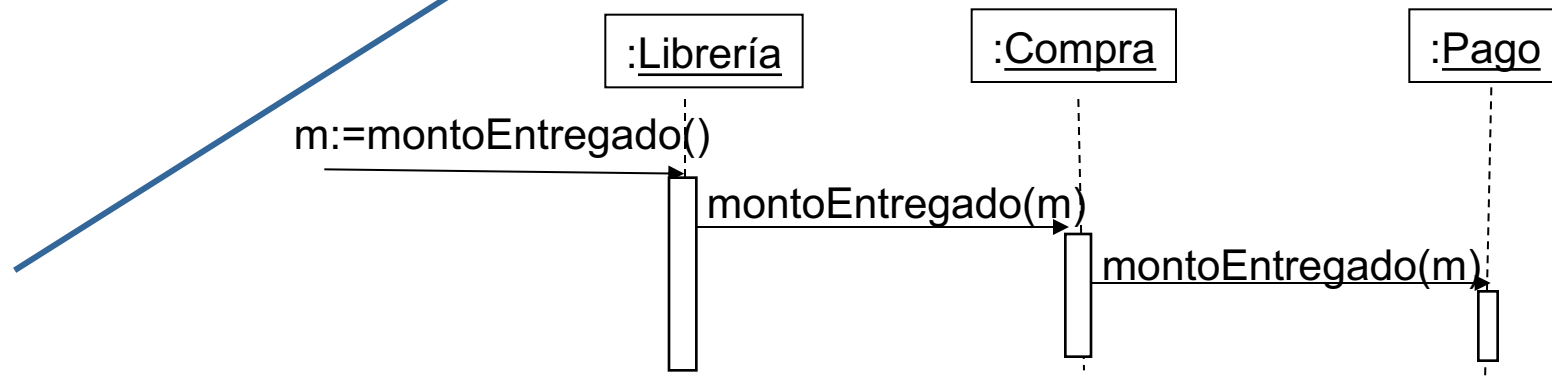
“No hables con extraños”



¿Dónde se da la envidia?
¿De qué tipo?

Mejor

Peor



Reuso de Código

Herencia vs. Composición

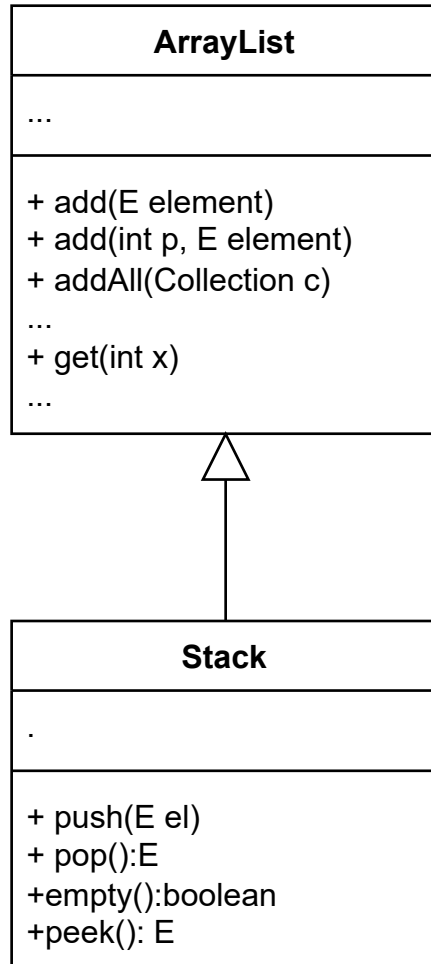
- Herencia **total**: debo conocer todo el código que se hereda -> Reutilización de **Caja Blanca**
- Usualmente debemos redefinir
- Los cambios en la superclase se propagan automáticamente a las subclases
- Herencia de Estructura vs. Herencia de comportamiento
- Es útil para extender la funcionalidad del dominio de aplicación

- Los objetos se componen en forma Dinámica -> Reutilización de **Caja Negra**
- Los objetos pueden reutilizarse a través de su **interfaz** (sin conocer el código)
- A través de las relaciones de composición se pueden delegar responsabilidades entre los objetos

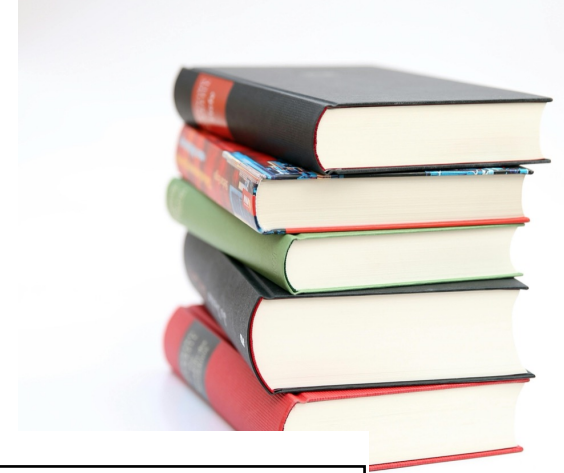
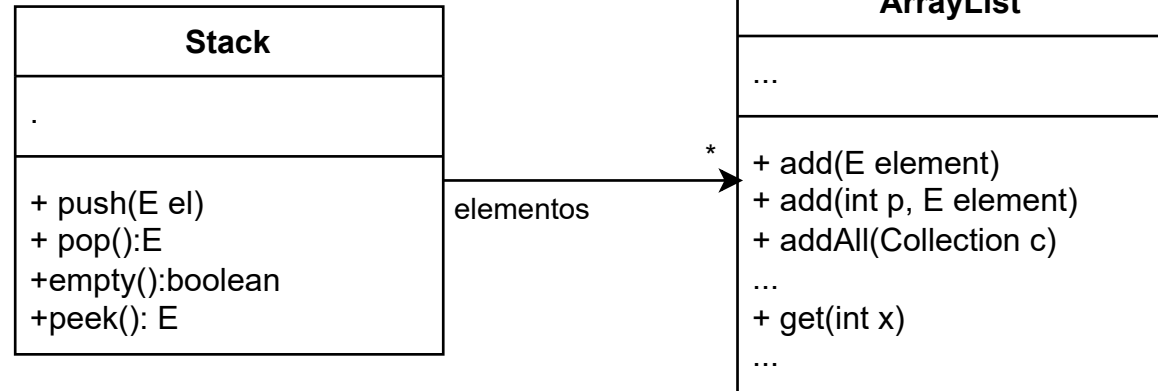
- Las clases y los objetos creados mediante herencia están *estrechamente acoplados* ya que cambiar algo en la superclase afecta directamente a la/las subclases.
- Las clases y los objetos creados a través de la composición están *débilmente acoplados*, lo que significa que se pueden cambiar más fácilmente los componentes sin afectar el objeto contenedor.

Veamos un ejemplo

Herencia



Composición



Cómo hacer un mal uso de la herencia

```
import java.util.ArrayList;

public class Stack<T> extends ArrayList<T> {

    public void push(T object) {
        this.add(object);
    }

    public T pop() {
        return this.remove(this.size() - 1);
    }

    public boolean empty() {
        return this.size() == 0;
    }

    public T peek() {
        return this.get(this.size() - 1);
    }
}
```



Cómo hacer un mal uso de la herencia

- Esta clase *Stack* funcionará como una pila, pero su **interfaz** es **voluminosa**; está formada por mensajes que hay que **anular o redefinir!!**
- La interfaz pública de esta clase no es solo push y pop, (esperable para una Pila), también incluye
 - **add en cualquier posición por índice,**
 - **remove de una posición a otra,**
 - **clear** y muchos otros mensajes heredados de ArrayList que **son inapropiados** para una Pila.

El ejemplo tiene errores de diseño

- uno **semántico**:

"una pila es una ArrayList" no es cierto; Stack no es un subtipo adecuado de ArrayList (No cumple con **Is-a**). Se supone que una pila aplica el último en entrar, primero en salir, una restricción que se satisface con la interfaz push/pop, pero que no se cumple con la interfaz de ArrayList que es mucho más extensa.

- Otro **mecánico**:

heredar de ArrayList viola el encapsulamiento; usar ArrayList para contener la colección de objetos de la pila es una opción de implementación que puede y debe ocultarse.

Composición, no Herencia

```
import java.util.ArrayList;

public class Stack<T> {
    private ArrayList<T> elementos

    public void push(T object) {
        elementos.add(object);
    }

    public T pop() {
        return elementos.remove(elementos.size() - 1);
    }

    public boolean empty() {
        return elementos.size() == 0;
    }

    public T peek() {
        return elementos.get(elementos.size() - 1);
    }
}
```



Composición, no herencia...

- Mecánicamente, heredar de ArrayList no cumple con el encapsulamiento; en cambio,
- **componer con ArrayList para contener la colección de objetos de la pila es una opción de implementación que permite ocultarla públicamente.**
- En este caso, en vez de heredar, el uso o composición permite reuso y mantiene el encapsulamiento!!

Heurísticas para Diseño “ágil” Orientado a Objetos

(Principios S O L I D)

Principios para Diseño “ágil” Orientado a Objetos

Principios **S O L I D**

Relacionados a las HAR, para un buen estilo de DOO.
Promueven Alta Cohesión y Bajo Acoplamiento.

*Robert C. Martin (2014) Agile Software Development, Principles, Patterns, and Practices.
Pearson New International Edition (Capítulos 8 al 12)*

S SRP: The Single-Responsibility Principle

Principio de Responsabilidad única. Una clase debería cambiar por una sola razón.

Debería ser responsable de únicamente una tarea, y ser modificada por una sola razón (alta cohesión).

O OCP: The Open-Closed Principle

Entidades de software (clases, módulos, funciones, etc.) deberían ser “abiertas” para extensión, y “cerradas” para modificación.

Abierto a extensión: ser capaz de añadir nuevas funcionalidades.

Cerrado a modificación: al añadir la nueva funcionalidad no se debe cambiar el diseño existente.

L LSP: The Liskov Substitution Principle

Los objetos de un programa deben ser intercambiables por instancias de sus subtipos sin alterar el correcto funcionamiento del programa.

Es decir, que si el programa utiliza una clase (clase A), y ésta es extendida (clases B, C, D, etc...) el programa tiene que poder utilizar cualquiera de sus subclases y seguir siendo válido. Uso correcto de herencia (Is-a) y polimorfismo.

I ISP: The Interface-Segregation Principle

Las clases que tienen interfaces “voluminosas” son clases cuyas interfaces no son cohesivas.

Las clases no deberían verse forzadas a depender de interfaces que no utilizan. Cuando creamos interfaces (protocolos) para definir comportamientos, las clases que las implementan, no deben estar forzadas a incluir métodos que no va a utilizar.

D DIP: The Dependency-Inversion Principle

- a. Los módulos de alto nivel de abstracción no deben depender de los de bajo nivel.*
- b. Las abstracciones no deben depender de detalles. Los detalles deben depender de las abstracciones.*

Módulos de alto nivel: se refieren a los objetos que definen qué es y qué hace el sistema.

Módulos de bajo nivel: no están directamente relacionados con la lógica de negocio del programa (no definen el dominio). Por ejemplo, el mecanismo de persistencia o el acceso a red .

Abstracciones: se refieren a protocolos (o interfaces) o clases abstractas.

Detalles: son las implementaciones concretas, (cuál mecanismo de persistencia, etc).

Ser capaz de «invertir» una dependencia es lo mismo que ser capaz de «intercambiar» una implementación concreta por otra implementación concreta cualquiera, respecto a la misma abstracción.

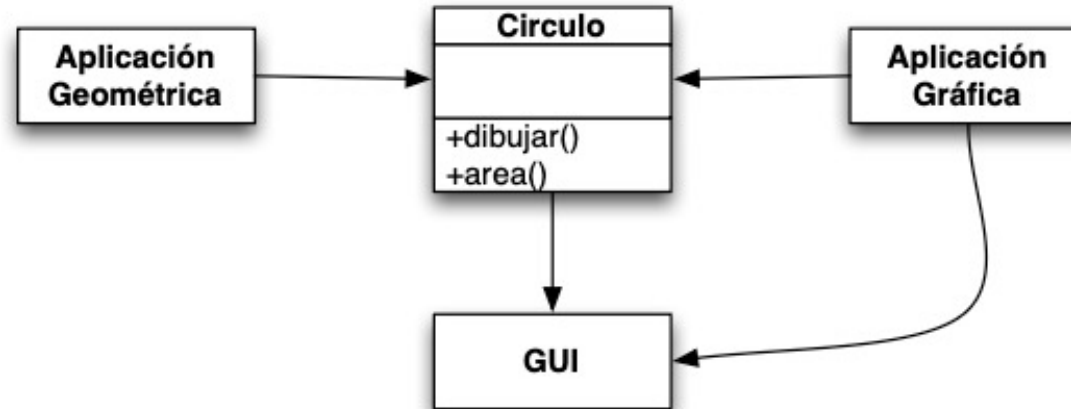


FIGURA: Clase Círculo

- La aplicación geométrica utiliza solamente el comportamiento relacionado a computar el area del círculo y jamas lo dibujará.
- La Aplicación Gráfica utiliza la opción de dibujar en la interfaz de usuario (GUI) al círculo.

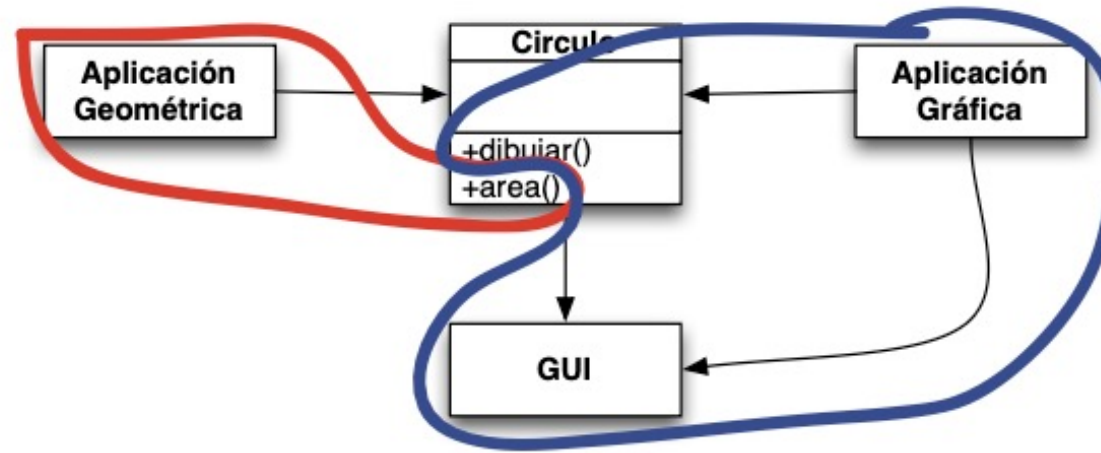


FIGURA: Clase Círculo

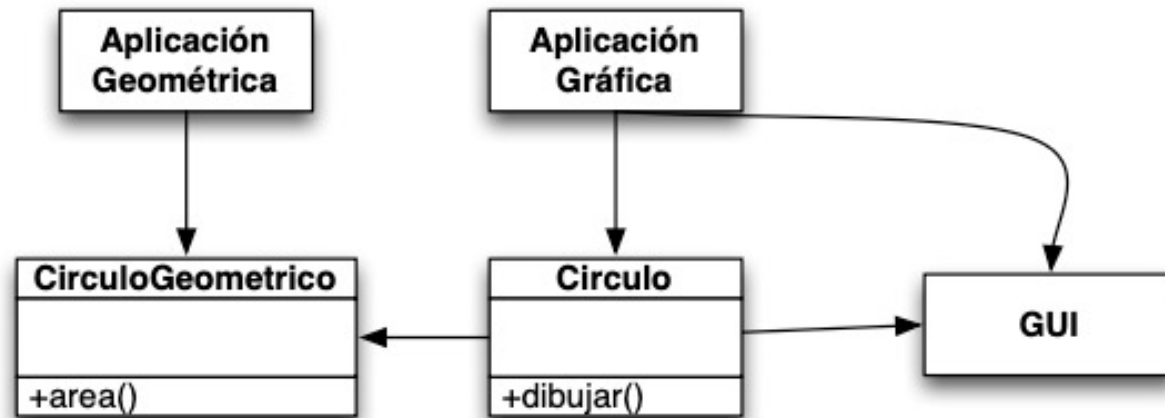
EL DISEÑO VIOLA EL PRINCIPIO DE SRP YA QUE TIENE DOS RESPONSABILIDADES

- La primera modela la representación geométrica del círculo
- La segunda es para renderizar el círculo en la interfáz gráfica.

LA VIOLACIÓN DEL SRP GENERA VARIOS PROBLEMAS

- debemos incluir la GUI en la aplicación de cálculo geométrico. En algunos lenguajes, como Java o .net debe ser ensamblada para poder realizar el deploy con la aplicación geométrica.
- si un cambio en la aplicación Gráfica causa que el círculo debe cambiar por alguna razón, entonces el cambio nos obliga a recompilar, re testear, y re-deployar la aplicación de computación geométrica. Si olvidamos hacer esto, la aplicación puede romperse de formas poco predecibles.

Single Responsibility - Solución



Un mejor diseño es separar en las responsabilidades en dos clases totalmente diferentes. Este diseño mueve la porción computacional del círculo a la clase `CirculoGeometrico`. Ahora los cambios sobre la manera en que se dibuja el rectangulo no pueden afectar a la aplicación geométrica.

Single Responsibility - Conclusiones

- El principio de Única Responsabilidad (SRP) es uno de los principios más simples de comprender pero el más difícil de aplicar. Juntar responsabilidades es algo que realizamos naturalmente.
- Encontrar y separar esas responsabilidades es algo más relacionado a lo que el diseño de software en verdad realiza. El resto de los principios que discutiremos vuelven de una forma a este.

- "Todas las variables de instancia deben ser privadas"
- "Deben evitarse las variables globales"
- Hacer uso de identificadores de tipos no es una buena práctica (ej. Variable de Instancia String tipo)

"Todos los sistemas cambian durante su ciclo de vida. Debemos considerar esto para hacer nuestros sistemas lo más durables posible"

— Ivar Jacobson

ABIERTAS PARA SER EXTENDIDAS

Significa que el comportamiento se debe poder extender.

CERRADAS PARA LA MODIFICACIÓN

El código de la clase es inviolable. Nadie puede alterar el código fuente de la misma.

ABIERTAS PARA SER EXTENDIDAS

Significa que el comportamiento se debe poder extender.

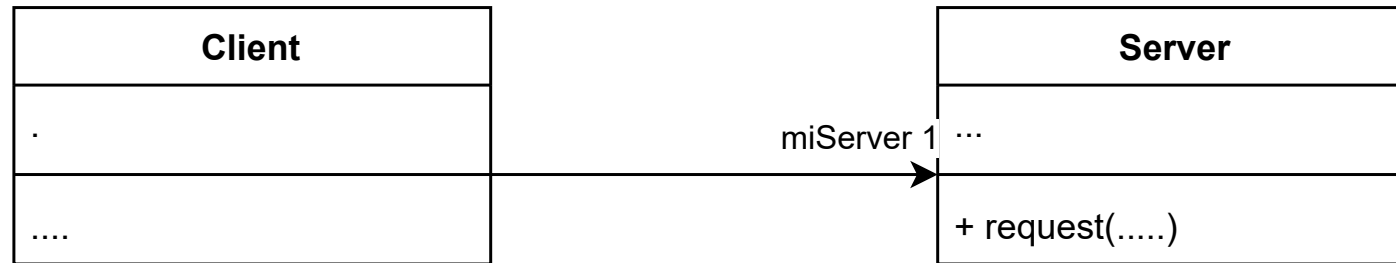
CERRADAS PARA LA MODIFICACIÓN

El código de la clase es inviolable. Nadie puede alterar el código fuente de la misma.

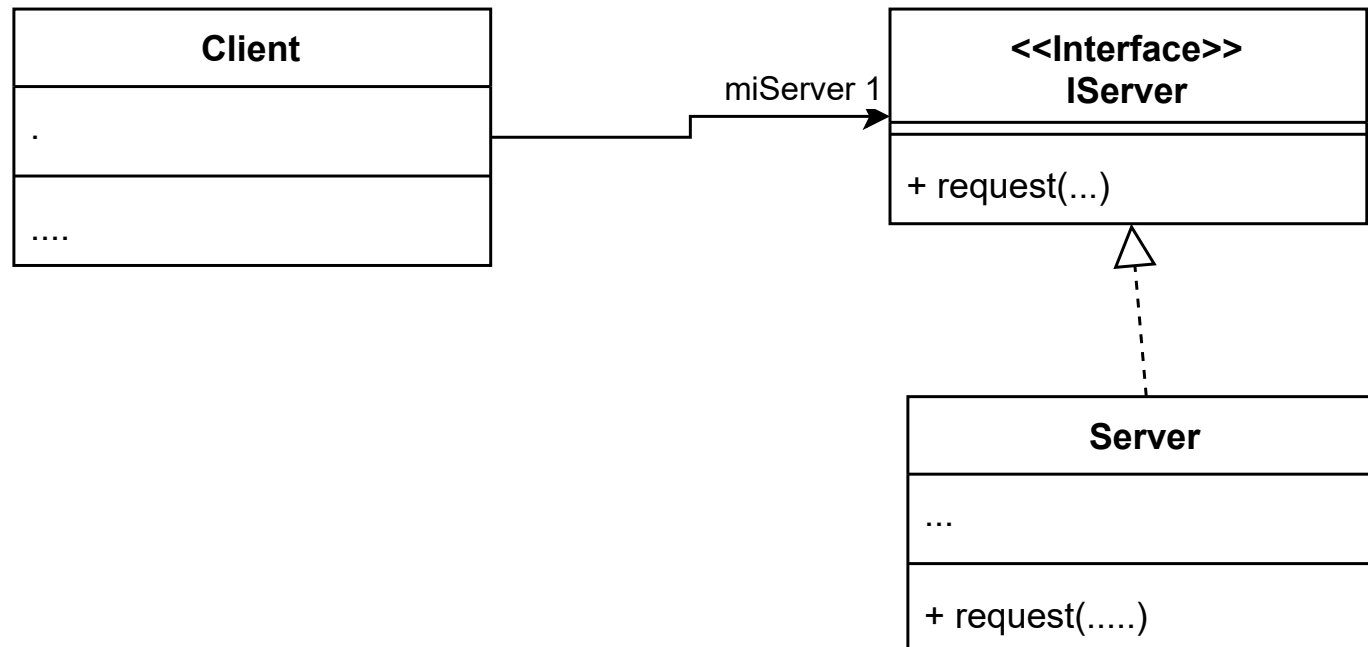
Open - Closed Principle

```
1. public class Mensaje {
2.     private String contenido;
3.     private String tipo; // "normal", "censurado",
        "encriptado"
4.
5.     public Mensaje(String contenido, String tipo) {
6.         this.contenido = contenido;
7.         this.tipo = tipo;
8.     }
9.
10.    public String getTipo() {
11.        return tipo;
12.    }
13.
14.    public String procesar() {
15.        String resultado = contenido;
16.
17.        if (tipo.equals("censurado") ||
            tipo.equals("encriptado")) {
18.            String[] malasPalabras = {"tonto", "feo"};
19.            for (String palabra : malasPalabras) {
```

```
20.                String censura =
                """.repeat(palabra.length());
21.                resultado = resultado.replaceAll("(?i)" +
                palabra, censura);
22.            }
23.        }
24.
25.        if (tipo.equals("encriptado")) {
26.            resultado = new
                StringBuilder(resultado).reverse().toString();
27.        }
28.
29.        return resultado;
30.    }
31.}
32.
```

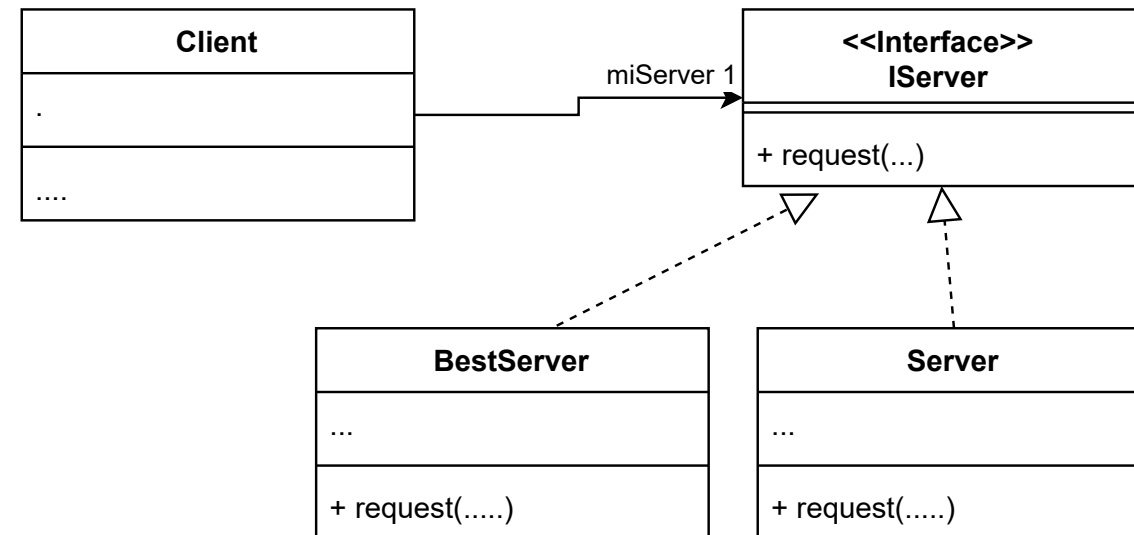
¿Cuál respeta mejor el OCP?



Open - Closed Principle - Mirando el diseño y el código

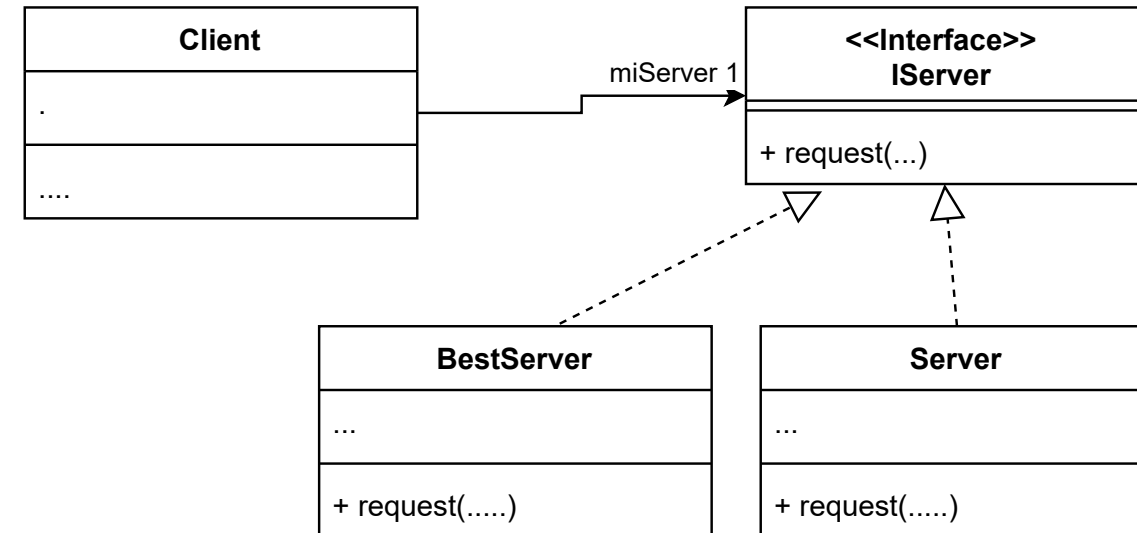
```
4 package model;
5
6 public interface IServer {
7
8     public void request(String value);
9
10 }
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
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
```

```
4 package model;
5
6 public class Server implements IServer{
7
8     @Override
9     public void request(String value) {
10         //....
11
12     }
13
14 }
15
```



Open - Closed Principle - Mirando el diseño y el código

```
4 package model;
5
5 package model;
6
7 public class Client {
8
9     IServer miServer;
10
11     public Client() {
12         this.miServer=new Server();
13     }
14
15     public void generarRequest(String string) {
16         this.miServer.request(string);
17     }
18
19 }
20
```



Open - Closed Principle - Mirando el diseño y el código

```
4 package model;
3 package model;
4
5 public class Client {
6
7     IServer miServer;
8
9     public Client(IServer server) {
10         this.miServer=server;
11     }
12
13     public void generarRequest(String string) {
14         this.miServer.request(string);
15     }
16
17 }
18
```

