

Programación Orientada a Objetos

Principios de diseño y refactoring

SOLID y cambios en pasos pequeños

Gerson Garrido

Semana 13

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

1. ¿Qué es un buen diseño? Cohesión y acoplamiento
2. Los cinco principios **SOLID**, con un ejemplo cada uno
3. Cuándo *no* aplicarlos
4. Refactoring: cambiar la estructura sin cambiar el comportamiento
5. Un ejemplo completo: cinco pasos con las pruebas en verde

> **Lectura**
>
> Eck, sec. 4.1. R. C. Martin, *The Single Responsibility Principle*, *The Open Closed Principle*, *Solid Relevance*. M. Fowler, *Definition Of Refactoring*, *Code Smell* y el catálogo de refactoring.

SECCIÓN 01

Diseño

01

¿Qué es un buen diseño?

- Dos programas que hacen lo mismo pueden ser muy distintos el día que hay que cambiarlos.
- Bodega cambió en cada entrega: precios por peso, descuentos, ventas, excepciones, un `Map`...
- **Diseño**: cómo se reparten las responsabilidades entre las clases y cómo dependen unas de otras.
- Un buen diseño hace baratos los cambios probables. No hay un diseño correcto en abstracto.

Cohesión y acoplamiento

Cohesión *(cohesion)*

Cuánto pertenece junto lo que está en una clase.

Baja cohesión: grupos de métodos que no comparten campos; clases `Manager`, `Utils`.

Acoplamiento *(coupling)*

Cuánto tiene que saber una clase de otra.

Una interfaz acopla menos que una clase concreta; un método, menos que un campo.

Objetivo: alta cohesión, bajo acoplamiento

Lo que cambia junto, junto. Lo que cambia por separado, que sepa lo menos posible de lo demás.

SECCIÓN 02

SOLID

02



SOLID en una diapositiva

	Principio	En una frase
S	Responsabilidad única	Una clase, una razón para cambiar
O	Abierto/cerrado	Agregar comportamiento sin editar lo que funciona
L	Sustitución de Liskov	Un subtipo cumple el contrato del supertipo
I	Segregación de interfaces	Nadie depende de métodos que no usa
D	Inversión de dependencias	El alto nivel depende de interfaces, no de detalles

Reunidos por Robert C. Martin; el acrónimo es de Michael Feathers. Heurísticas, no leyes.

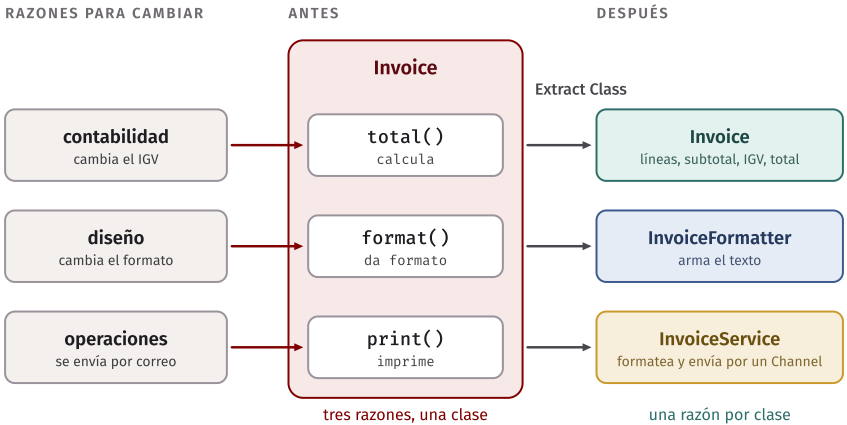
S: una factura que hace de todo

invoice/before/Invoice.java · recortado

```java
public int total() {                    // razón 1
    int subtotal = 0;
    for (Line line : lines) {
        subtotal += line.quantity()
                * line.unitPrice();
    }
    return subtotal + (subtotal * 18 + 50) /
     100;
}
public String format() { ... }          // razón 2
public void print() {                   // razón 3
    System.out.print(format());
}
```

- Corta, se entiende, funciona.
- ¿Quién va a pedir cambios?
- Contabilidad: el IGV.
- Diseño: el formato.
- Operaciones: enviar por correo.

S: tres razones para cambiar, tres clases



S: después, `Invoice` solo calcula

invoice/after/Invoice.java · recortado

```java
public class Invoice {
    private static final int TAX_PERCENT = 18;
    // customer, lines, add(...), lines() ...
    public int subtotal() { ... }
    public int tax() {
        return (subtotal() * TAX_PERCENT
                + 50) / 100;
    }
    public int total() {
        return subtotal() + tax();
    }
}
```

- `Invoice`: datos y reglas. Nada de texto.
- `InvoiceFormatter`: arma el texto.
- `InvoiceService`: formatea y envía por un `Channel`.
- Cada pedido de cambio tiene una sola dirección.

Error común: “responsabilidad única” no es “un método”

Una razón para cambiar, no una cosa pequeña

`Invoice` después del cambio tiene seis métodos públicos, y está bien: todos sirven a la misma razón.

- Partir una clase cohesiva en cinco clases de un método no la mejora: reparte una idea en cinco archivos.
- La pregunta útil: si mañana cambia X, ¿cuántas clases toco? ¿Y si cambia Y?

O: abierto a la extensión, cerrado a la modificación

el mismo switch, en varios métodos

```
1  int fine = switch (member.type()) {
2      case STUDENT -> ...;
3      case TEACHER -> ...;
4      case VISITOR -> ...;
5  };
```

Un tipo de socio nuevo: buscar y editar cada `switch`.

- Con polimorfismo (semana 6): un tipo nuevo es una clase nueva; el código que la usa no se toca.
- En la Sesión B quitamos un `switch` así: *Replace Conditional with Polymorphism*.
- Pero no todo `switch` es malo…

O: ningún diseño está cerrado a todo

Diseño	Tipo nuevo	Operación nueva
Jerarquía con override (sem. 6)	una clase nueva	editar todas las clases
`sealed` + `switch` exhaustivo (sem. 8)	editar cada `switch`	un método nuevo

- Con `sealed`, el compilador señala cada `switch` incompleto.
- Elegir es apostar: en un catálogo llegan *tipos* nuevos; en `Expr` (semana 8), *operaciones* nuevas.

L: sustitución de Liskov, con contratos

ArchiveChannel.java

```java
// Channel promete enviar;
// este canal se niega.
class ArchiveChannel
        implements Channel {
    @Override
    public void send(String to,
                     String text) {
        throw new
          UnsupportedOperationException();
    }
}
```

- Compila: la excepción es unchecked.
- `InvoiceService` confía en que `send` envía: falla en ejecución.
- Un subtipo no exige más, no garantiza menos y no lanza lo que el supertipo no anuncia (semana 9).
- Señales: un override que lanza `UnsupportedOperationException`; `instanceof` en el cliente.

I: segregación de interfaces

antes

```
interface OfficeMachine {
    void print(String doc);
    String scan();
    void fax(String n, String doc);
}
// BasicPrinter implementa las tres:
// scan() y fax() lanzan excepción
```

después

```
interface Printer { ... }
interface DocumentScanner { ... }
interface Fax { ... }

class BasicPrinter
        implements Printer { ... }
class AllInOne implements Printer,
        DocumentScanner, Fax { ... }
```

Interfaces según quién las usa; una clase puede implementar varias. Caso extremo: las interfaces funcionales de la semana 12.

D: el alto nivel crea el detalle

InvoiceService.java · versión acoplada

```java
public class InvoiceService {
    private final ConsoleChannel channel = new ConsoleChannel();
    ...
}
```

- **Alto nivel**: las reglas (emitir una factura). **Bajo nivel**: los detalles (consola, archivo, correo).
- Enviar por correo: editar `InvoiceService`.
- Probarlo: no se puede evitar que escriba en la consola.

D: la abstracción, y la dependencia por el constructor

invoice/after/InvoiceService.java

```java
public class InvoiceService {
    private final InvoiceFormatter formatter;
    private final Channel channel;

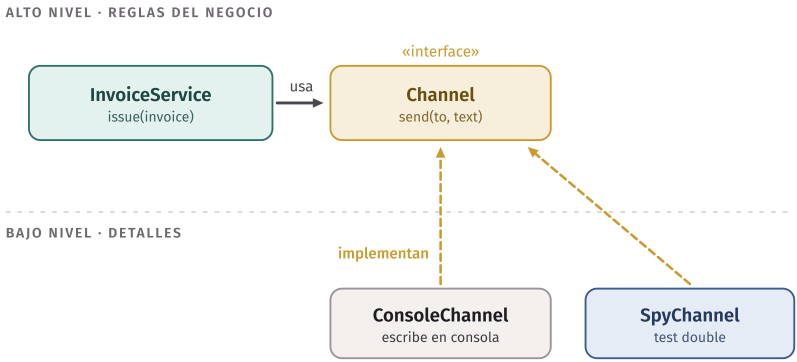
    public InvoiceService(InvoiceFormatter f,
                          Channel channel) {
        this.formatter = f;
        this.channel = channel;
    }
    public void issue(Invoice invoice) {
        channel.send(invoice.customer(),
                formatter.format(invoice));
    }
}
```

Channel.java

```java
interface Channel {
  void send(String to,
            String text);
}
```

- El alto nivel define la interfaz; los detalles la implementan.
- **Inyección de dependencias** por el constructor.

D: ¿qué se invierte?



Las dos capas dependen de Channel; la flecha del detalle ahora apunta hacia arriba

D: las piezas se arman en `main`

invoice/after/InvoiceDemo.java

```java
var invoice = new Invoice("alice");
invoice.add("cuaderno", 3, 450);
invoice.add("lapicero", 2, 150);
var service = new InvoiceService(
        new InvoiceFormatter(),
        new ConsoleChannel());
service.issue(invoice);
```

bash

```
[para alice]
Factura para alice
   3 x cuaderno     S/ 13.50
   2 x lapicero     S/ 3.00
Total con IGV: S/ 19.47
```

El `new` no desaparece: se muda al borde del programa. No todo `new` es un problema; se inyecta lo que habla con el mundo (consola, archivos, red, reloj) o lo que una prueba quiere reemplazar.

D: probar con un spy

invoice/after/SpyChannel.java

```java
class SpyChannel implements Channel {
    final List<String> recipients =
            new ArrayList<>();
    final List<String> texts =
            new ArrayList<>();
    @Override
    public void send(String to, String
      t) {
        recipients.add(to);
        texts.add(t);
    }
}
```

InvoiceServiceTest.java

```java
var spy = new SpyChannel();
var service = new InvoiceService(
        new InvoiceFormatter(),
    spy);
service.issue(sample());
assertEquals(List.of("alice"),
        spy.recipients);
```

Un test double escrito a mano, como en la semana 7: recuerda lo que le pidieron enviar.

Cuándo no aplicar SOLID

Error común: SOLID a ciegas

Cien líneas, diez interfaces de un método con una sola implementación, una factory por clase. Cada interfaz es una indirección más que leer.

- `Channel` se justifica: consola, spy y (pronto) correo.
- Un principio se aplica cuando hay un cambio o una prueba concreta que lo pide.

Las cuatro reglas del diseño simple (Kent Beck)

1. Pasa las pruebas. 2. Revela su intención. 3. Sin duplicación. 4. El menor número de elementos.

SECCIÓN 03

Refactoring

03



Refactoring

Refactoring (M. Fowler)

Cambiar la estructura interna del código sin cambiar su comportamiento observable, para que sea más fácil de entender y de modificar.

- Sin cambiar el comportamiento: ni siquiera sus rarezas. Arreglar un error es otro cambio.
- En pasos pequeños: compila y pasa las pruebas después de cada uno.
- ¿Cómo se sabe que no cambió? Con **pruebas**. Sin esa red de seguridad es una apuesta arriesgada: primero, characterization tests.

El ciclo

1. Pruebas en verde.
2. Un cambio pequeño.
3. Correr las pruebas.
4. ¿Pasan? Commit con el nombre del refactoring.
5. ¿Fallan? Deshacer y probar un paso más chico.

bash

```
$ make test
$ git commit -am "Extract Function"
```

bash

```
$ git restore .
```

Deshacer pierde dos minutos. Depurar un cambio grande que rompió algo puede costar una tarde.

Code smells

Code smell	Cómo se ve	Refactoring
Método largo	hay que leerlo con el dedo	Extract Function
Clase grande	métodos que no se usan entre sí	Extract Class
Código duplicado	el mismo bloque en dos lugares	Extract Function
Lista larga de parámetros	`register(a, b, c, d, e)`	agrupar en un record
Feature envy	usa más los datos de otra clase	Move Function
`switch` repetido por tipo	el mismo `switch` en varios lados	Replace Conditional w. Polymorphism
Nombres que no dicen nada	`d`, `tmp`, `proc()`	Rename

Un code smell no es un error: el programa funciona. Es una pista de que el próximo cambio va a costar.

SECCIÓN 04

Un ejemplo completo

04

El mostrador de la biblioteca

Multas por devolver tarde, en céntimos:

- Por día: libro 50, DVD 100.
- **Estudiante**: 2 días de gracia, tope de S/ 10.00 por préstamo.
- **Docente**: no paga libros; DVD, tarifa normal.
- **Visitante**: el doble, sin gracia ni tope.

Los datos: records y enums

```
MemberType: STUDENT, TEACHER,
VISITOR
ItemKind: BOOK, DVD
Member(name, type)
Loan(title, kind, due, returned)
```

Todo el trabajo lo hace `LibraryDesk`: `statement` y `totalFine`.

`stepo`: funciona, tiene pruebas…

library/stepo/LibraryDesk.java · el bucle de statement, recortado

```java
for (Loan x : l) {
    int d = (int) ChronoUnit.DAYS.between(x.due(), x.returned());
    if (d < 0) { d = 0; }
    int a = 0;
    switch (m.type()) {
        case STUDENT:
            if (d > 2) { a = (d - 2) * (x.kind() == ItemKind.DVD ? 100 : 50); }
            if (a > 1000) { a = 1000; }
            break;
        case TEACHER: ...   case VISITOR: ...
    }
    t += a;   s += String.format("  %-18s %3d días  S/ %d.%02d\n", ...);
}
```

`totalFine` repite el cálculo entero. El `switch` es del estilo antiguo, con `break`.

…y tiene code smells. ¿Cuáles ven?

- **Método largo**: `statement` lee, calcula, aplica reglas, formatea y suma.
- **Código duplicado**: `totalFine` copia el cálculo. ¿Y si cambia el tope?
- **Nombres que no dicen nada**: `s`, `t`, `x`, `d`, `a`, `m`, `l`.
- **Feature envy**: los días de atraso solo usan datos de `Loan`.
- **`switch` repetido por tipo**: dos, y uno más por cada regla nueva.
- **Dos responsabilidades**: calcular y dar formato.

> **Meta**
>
> Llegar a un diseño sin esos code smells sin cambiar el comportamiento, en cinco pasos. Cada paso es un paquete: `library/step0` … `step5`.

Paso 0: characterization tests

library/step0/LibraryDeskTest.java

```java
@Test
void studentStatement() {
    var alice = new Member("alice",
            MemberType.STUDENT);
    String expected = """
        Multas de alice (STUDENT)
          Clean Code          5 días  S/ 1.50
          Matrix              1 días  S/ 0.00
          SICP                0 días  S/ 0.00
        Total: S/ 1.50
        """;
    assertEquals(expected,
        desk.statement(alice, ALICE_LOANS));
}
```

- Llamar al código, mirar qué devuelve, exigir exactamente eso.
- Siete pruebas: cada regla, el tope, el caso vacío, `totalFine`.
- `1 días` es un error. La prueba lo exige igual; se arregla después, en su propio commit.

Las mismas pruebas en todos los pasos

Probar la interfaz pública

`statement`, `totalFine` y los constructores de `Member` y `Loan`, nada interno.

bash · make test, recortado

```
== las pruebas de step1..step5 son idénticas a las de step0
== pruebas de library.step0: 7 tests successful
...
== pruebas de library.step5: 7 tests successful
```

El `Makefile` compara los archivos de prueba (salvo `package`) y prueba cada paso aparte.

Cinco pasos, siete pruebas en verde

PASOS (UN COMMIT CADA UNO)

0	**Characterization tests**	se fija lo que el código hace hoy
1	**Rename**	d, a, t, x → daysLate, fine, total, loan
2	**Extract Function**	fineFor(), daysLate(), money(): sin duplicado
3	**Move Function**	cada cálculo a la clase que tiene los datos
4	**Extract Class**	el formato se va a StatementFormatter
5	**Replace Conditional with Polymorphism**	cada tipo de socio, su propia regla

MAKE TEST

7 de 7 pruebas pasan

las mismas pruebas en todos los pasos

EL CÓDIGO

líneas de LibraryDesk	26
switch por tipo de socio	0
líneas de producción en total	121

el total puede subir:
refactorizar no es acortar

El comportamiento no cambió; la estructura, sí.

Paso 1: Rename

library/step1/LibraryDesk.java

```java
for (Loan loan : loans) {
    int daysLate = (int) ChronoUnit.DAYS
            .between(loan.due(), loan.returned())
     ;
    if (daysLate < 0) {
        daysLate = 0;
    }
    int fine = 0;
    switch (member.type()) {
        case STUDENT:
            if (daysLate > 2) {
                fine = (daysLate - 2) * (...);
            }
            ...
```

- `s` → `text`, `t` → `total`, `x` → `loan`, `d` → `daysLate`, `a` → `fine`...
- Ninguna línea cambió de lugar.
- Ahora el código se lee como la regla.
- El editor renombra de forma segura; a mano, una variable a la vez.

Paso 2: Extract Function

library/step2/LibraryDesk.java · recortado

```java
// en statement:
for (Loan loan : loans) {
    text += String.format("...",
            loan.title(), daysLate(loan),
            money(fineFor(member, loan)));
}
// en totalFine:
for (Loan loan : loans) {
    total += fineFor(member, loan);
}
private int fineFor(Member m, Loan loan)
private int daysLate(Loan loan)
private static String money(int cents)
```

- Tres métodos con nombre.
- `totalFine` ya no copia: un solo `switch`.
- `LibraryDesk`: de 75 a 59 líneas.
- `Math.max(days, 0)` reemplaza un `if`: lo “obvio”, con las pruebas corriendo.

Paso 3: Move Function (feature envy)

library/step3/Loan.java

```java
public record Loan(String title,
        ItemKind kind, LocalDate due,
        LocalDate returned) {
    public int daysLate() {
        int days = (int) ChronoUnit.DAYS
                .between(due, returned);
        return Math.max(days, 0);
    }
}
```

- `daysLate(loan)` no usaba nada de `LibraryDesk`: se muda a `Loan`.
- `? 100 : 50`, escrito tres veces, es un dato de cada tipo de ítem: `BOOK(50)`, `DVD(100)` y `dailyRate()` en `ItemKind`.
- En un proyecto real: un commit por movimiento.

Paso 3: la multa, a `MemberType`

library/step3/MemberType.java · dentro del enum

```java
int fineFor(Loan loan) {
    int daysLate = loan.daysLate();
    int rate = loan.kind().dailyRate();
    return switch (this) {
        case STUDENT -> Math.min(Math.max(daysLate - 2, 0) * rate, 1000);
        case TEACHER -> loan.kind() == ItemKind.DVD ? daysLate * rate : 0;
        case VISITOR -> daysLate * rate * 2;
    };
}
```

- `switch` con flechas: exhaustivo, sin `default`. `Member.fineFor(loan)` delega en su tipo.
- `Math.min(Math.max(...))` reemplaza dos `if`: `studentFineIsCappedPerLoan` lo confirma.

Paso 4: Extract Class

library/step4/LibraryDesk.java · recortado

```java
private final StatementFormatter formatter =
        new StatementFormatter();
public String statement(Member member,
                        List<Loan> loans) {
    var text = new StringBuilder(
            formatter.header(member));
    for (Loan loan : loans) {
        text.append(formatter.line(loan,
                member.fineFor(loan)));
    }
    text.append(formatter.footer(
            totalFine(member, loans)));
    return text.toString();
}
```

- El formato se va a `StatementFormatter`.
- `LibraryDesk` coordina.
- ¿Y el `new`? El formato no habla con el mundo ni se cambia en una prueba.

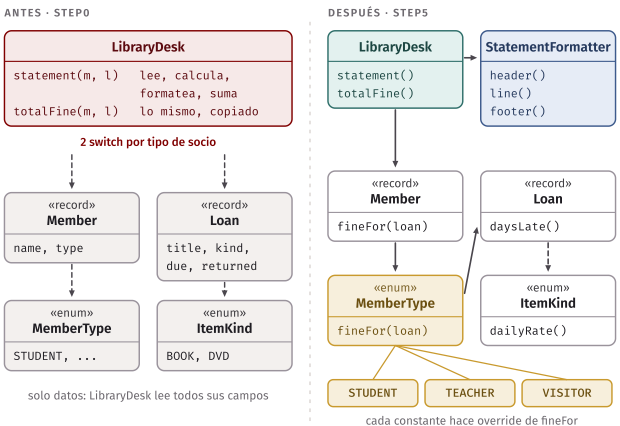
Paso 5: Replace Conditional with Polymorphism

library/step5/MemberType.java · recortado

```java
public enum MemberType {
    STUDENT {
        @Override int fineFor(Loan loan) {
            int charged = Math.max(
                    loan.daysLate() -
     GRACE_DAYS, 0);
            return Math.min(charged
                    * loan.kind().dailyRate(),
     CAP);
        }
    },
    TEACHER { ... },  VISITOR { ... };
    abstract int fineFor(Loan loan);
}
```

- Cada constante hace override de `fineFor`: dynamic dispatch, sin `switch`.
- Tipo nuevo = constante nueva; si olvida el método, no compila.
- ¿Por qué no una `sealed interface`? Cambiaría la interfaz pública, y con ella las pruebas.

Antes y después



Lo mismo, organizado de otra forma

Pregunta	Antes (step0)	Después (step5)
¿Días de atraso?	`LibraryDesk`, dos veces	`Loan.daysLate()`
¿Tarifa diaria?	tres `? 100 : 50`	`ItemKind.dailyRate()`
¿Regla del estudiante?	un `case` en dos `switch`	la constante `STUDENT`
¿Formato?	mezclado con los cálculos	`StatementFormatter`
¿Qué hace `LibraryDesk`?	todo (75 líneas)	coordina (26 líneas)

En total, de 99 a 121 líneas: refactorizar no es acortar. Bajó lo que hay que leer para cada cambio.

El mismo estado de cuenta, antes y después

Treinta días de un DVD serían S/ 28.00 para una estudiante; el tope los deja en S/ 10.00.

bash

```
$ java -cp build library.LibraryDemo
== step0
Multas de alice (STUDENT)
  Clean Code           5 días  S/ 1.50
  Matrix              30 días  S/ 10.00
Total: S/ 11.50
== step5
Multas de alice (STUDENT)
  Clean Code           5 días  S/ 1.50
  Matrix              30 días  S/ 10.00
Total: S/ 11.50
```

Error común: refactorizar y cambiar a la vez

En el paso 3, “de paso”, el tope de estudiantes sube a S/ 15.00:

MemberType.java

```
1  case STUDENT -> Math.min(
2      Math.max(daysLate - 2, 0)
        * rate,
3      1500);
```

bash · recortado

```
studentFineIsCappedPerLoan() ×
   expected: <1100> but was: <1600>
...
[          6 tests successful      ]
[          1 tests failed          ]
```

¿Qué lo rompió: el movimiento o el tope nuevo?

Con los dos en el mismo paso, no se sabe sin depurar. En dos commits, la respuesta es inmediata: primero el refactoring, con las pruebas intactas; después el cambio, con la prueba del tope actualizada a propósito.

Cuándo refactorizar

Sí

- Antes de agregar algo que el diseño actual hace difícil.
- Cuando por fin se entiende algo que costó entender.
- A la tercera repetición: la primera se escribe, la segunda se tolera.

No

- Sin pruebas (primero, los characterization tests).
- Código que no se va a volver a tocar.
- A mitad de un cambio de funcionalidad.

> **Primero hacer fácil el cambio; después, hacer el cambio fácil**
>
> Refactorizar es la primera mitad. Nunca al mismo tiempo que la segunda.

SECCIÓN 05

Laboratorio

05

Laboratorio: refactorizar

- `code/week-13/lab`: un registro de notas que funciona, tiene pruebas y está mal diseñado a propósito.
- Listar sus code smells y el principio que viola cada uno.
- Refactorizar en pasos pequeños: `make test` y un commit por paso, con el nombre del refactoring.
- Inversión de dependencias en la clase que crea sus colaboradores con `new`; probarla con un test double.
- Lo mismo en su P4: dos code smells, corregidos sin cambiar ninguna prueba existente.

Resumen

- **Diseño**: repartir responsabilidades para que los cambios probables sean baratos. Alta **cohesión**, bajo **acoplamiento**.
- **S** una razón para cambiar · **O** extender sin editar · **L** cumplir el contrato · **I** interfaces pequeñas · **D** depender de abstracciones, por el constructor.
- Los principios se aplican cuando un cambio o una prueba concreta los pide.
- **Refactoring**: misma conducta, otra estructura, en pasos pequeños con las pruebas en verde.
- Sin pruebas, primero **characterization tests** contra la interfaz pública.
- Rename, Extract Function, Move Function, Extract Class, Replace Conditional with Polymorphism.
- Nunca refactoring y cambio de funcionalidad en el mismo paso.

Para la próxima clase

- Lecturas de la semana 14: Refactoring.Guru, *Strategy* y *Observer*.
- Laboratorio: traer el repositorio de P4 con `make test` en verde.
- P4 en curso: la segunda mitad del enunciado es refactorizar con SOLID lo que P1–P3 dejaron acoplado.

Pregunta para pensar

El `Comparator` de las semanas 11 y 12 y el `Channel` de hoy tienen algo en común: un objeto que se pasa para decidir *cómo* se hace algo. ¿Qué nombre le darían a esa idea?