

Programación Orientada a Objetos

Patrones de diseño

Strategy, Observer, Factory y Composite

Gerson Garrido

Semana 14

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

1. Qué es un patrón de diseño *(design pattern)*, y qué no
2. **Strategy**: elegir un algoritmo mientras el programa corre
3. **Observer**: avisar sin conocer a quién
4. **Factory**: decidir qué crear en un solo lugar
5. **Composite**: una parte y un todo que se tratan igual

> **Lectura**
>
> Refactoring.Guru: Strategy, Observer, Factory Method y Composite. Eck, subsec. 9.4.3 *Expression Trees*.

SECCIÓN 01

Qué es un patrón

01



De la arquitectura al software

- Años 70: Christopher Alexander pone nombre a soluciones que se repiten en casas y ciudades.

De la arquitectura al software

- Años 70: Christopher Alexander pone nombre a soluciones que se repiten en casas y ciudades.
- 1994: Gamma, Helm, Johnson y Vlissides, *Design Patterns*: 23 patrones para programas orientados a objetos.

De la arquitectura al software

- Años 70: Christopher Alexander pone nombre a soluciones que se repiten en casas y ciudades.
- 1994: Gamma, Helm, Johnson y Vlissides, *Design Patterns*: 23 patrones para programas orientados a objetos.
- Hoy vemos cuatro. Ya usaron varios sin saber su nombre.

De la arquitectura al software

- Años 70: Christopher Alexander pone nombre a soluciones que se repiten en casas y ciudades.
- 1994: Gamma, Helm, Johnson y Vlissides, *Design Patterns*: 23 patrones para programas orientados a objetos.
- Hoy vemos cuatro. Ya usaron varios sin saber su nombre.

Un patrón tiene tres partes

1. Un **problema** que se repite en muchos programas.
2. Una **solución con nombre**: cómo repartir responsabilidades.
3. Sus **consecuencias**: qué se gana y qué cuesta.

Vocabulario, no receta

Idea clave

Un patrón es vocabulario compartido. Se usa cuando el problema está presente y se reconoce por el problema, no por los nombres de las clases. Un programa no es mejor por tener más patrones.

Vocabulario, no receta

Idea clave

Un patrón es vocabulario compartido. Se usa cuando el problema está presente y se reconoce por el problema, no por los nombres de las clases. Un programa no es mejor por tener más patrones.

De cada patrón veremos cuatro cosas:

1. el **problema** que resuelve,
2. su **estructura**,
3. un **ejemplo** en Java,
4. y cuándo no usarlo: todo patrón agrega clases e indirección.

SECCIÓN 02

Strategy

02

El problema: un `switch` que crece

ShippingSwitch.java

```java
static Money cost(String kind, Parcel parcel) {
  return switch (kind) {
    case "flat" -> Money.of(1000, "PEN");
    case "weight" -> {
      long kg = (parcel.grams() + 999) / 1000;
      yield Money.of(500 + 300 * kg, "PEN");
    }
    case "free-above" -> ...
    default -> throw new IllegalArgument...;
  };
}
```

Tarifas de envío de una tienda en línea.

El problema: un `switch` que crece

ShippingSwitch.java

```java
static Money cost(String kind, Parcel parcel) {
  return switch (kind) {
    case "flat" -> Money.of(1000, "PEN");
    case "weight" -> {
      long kg = (parcel.grams() + 999) / 1000;
      yield Money.of(500 + 300 * kg, "PEN");
    }
    case "free-above" -> ...
    default -> throw new IllegalArgument...;
  };
}
```

Tarifas de envío de una tienda en línea.

- Tarifa nueva: editar un método que ya funciona. Abierto/cerrado.

El problema: un `switch` que crece

ShippingSwitch.java

```java
static Money cost(String kind, Parcel parcel) {
  return switch (kind) {
    case "flat" -> Money.of(1000, "PEN");
    case "weight" -> {
      long kg = (parcel.grams() + 999) / 1000;
      yield Money.of(500 + 300 * kg, "PEN");
    }
    case "free-above" -> ...
    default -> throw new IllegalArgument...;
  };
}
```

Tarifas de envío de una tienda en línea.

- Tarifa nueva: editar un método que ya funciona. Abierto/cerrado.
- Los S/ 5 y S/ 3 están enterrados en el `case`.

El problema: un `switch` que crece

ShippingSwitch.java

```java
static Money cost(String kind, Parcel parcel) {
  return switch (kind) {
    case "flat" -> Money.of(1000, "PEN");
    case "weight" -> {
      long kg = (parcel.grams() + 999) / 1000;
      yield Money.of(500 + 300 * kg, "PEN");
    }
    case "free-above" -> ...
    default -> throw new IllegalArgument...;
  };
}
```

Tarifas de envío de una tienda en línea.

- Tarifa nueva: editar un método que ya funciona. Abierto/cerrado.
- Los S/ 5 y S/ 3 están enterrados en el `case`.
- `"wieght"` se descubre recién al ejecutar.

La solución: una interfaz, una clase por algoritmo

ShippingRate.java · FlatRate.java · ByWeight.java

```java
interface ShippingRate { Money cost(Parcel p); }
record FlatRate(Money amount) ... {
    public Money cost(Parcel p) {return amount;}
}
record ByWeight(Money base, Money perKilo) ... {
    public Money cost(Parcel p) {
        long kilos = (p.grams() + 999) / 1000;
        return base.plus(perKilo.times(kilos));
    }
}
```

...: implements ShippingRate.

- La **strategy**: una interfaz con un método.

La solución: una interfaz, una clase por algoritmo

ShippingRate.java · FlatRate.java · ByWeight.java

```java
interface ShippingRate { Money cost(Parcel p); }
record FlatRate(Money amount) ... {
    public Money cost(Parcel p) {return amount;}
}
record ByWeight(Money base, Money perKilo) ... {
    public Money cost(Parcel p) {
        long kilos = (p.grams() + 999) / 1000;
        return base.plus(perKilo.times(kilos));
    }
}
```

...: `implements ShippingRate`.

- La **strategy**: una interfaz con un método.
- Cada algoritmo, una clase. Sin más estado que sus parámetros: records.

La solución: una interfaz, una clase por algoritmo

ShippingRate.java · FlatRate.java · ByWeight.java

```java
interface ShippingRate { Money cost(Parcel p); }
record FlatRate(Money amount) ... {
    public Money cost(Parcel p) {return amount;}
}
record ByWeight(Money base, Money perKilo) ... {
    public Money cost(Parcel p) {
        long kilos = (p.grams() + 999) / 1000;
        return base.plus(perKilo.times(kilos));
    }
}
```

`...`: `implements ShippingRate`.

- La **strategy**: una interfaz con un método.
- Cada algoritmo, una clase. Sin más estado que sus parámetros: records.
- Los números ya no están enterrados: son parámetros. Dos tiendas, dos objetos.

Strategies que delegan, y el context

FreeAbove.java

```java
record FreeAbove(Money threshold,
        ShippingRate otherwise)
        implements ShippingRate {
  public Money cost(Parcel p) {
    var sub = p.subtotal();
    if (sub.isLessThan(threshold)) {
      return otherwise.cost(p);
    }
    return Money.zero();
  }
}
```

Por debajo del mínimo, delega en otra tarifa.

Strategies que delegan, y el context

FreeAbove.java

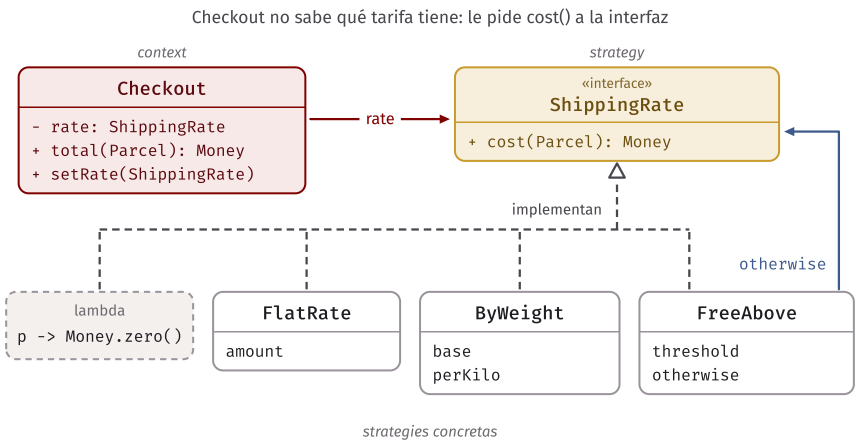
```java
record FreeAbove(Money threshold,
        ShippingRate otherwise)
        implements ShippingRate {
  public Money cost(Parcel p) {
    var sub = p.subtotal();
    if (sub.isLessThan(threshold)) {
      return otherwise.cost(p);
    }
    return Money.zero();
  }
}
```

Checkout.java · el context

```java
class Checkout {
  private ShippingRate rate;
  Checkout(ShippingRate rate) {
    this.rate = rate;
  }
  void setRate(ShippingRate r)
     {...}
  Money total(Parcel p) {
    var ship = rate.cost(p);
    return p.subtotal().plus(ship);
  }
}
```

Por debajo del mínimo, delega en otra tarifa.

Estructura de Strategy



Ejemplo: tres tarifas y una lambda

ShippingDemo.java · fragmento

```java
ShippingRate pickup = p -> Money.zero();
var checkout = new Checkout(rates.get(1));
IO.println("total por peso: "
        + checkout.total(heavy));
checkout.setRate(pickup);
IO.println("total en tienda: "
        + checkout.total(heavy));
```

bash · final de la salida

```
$ java -cp build ShippingDemo
...
total por peso: S/ 197.00
total en tienda: S/ 180.00
```

Ejemplo: tres tarifas y una lambda

ShippingDemo.java · fragmento

```
1  ShippingRate pickup = p -> Money.zero();
2  var checkout = new Checkout(rates.get(1));
3  IO.println("total por peso: "
4          + checkout.total(heavy));
5  checkout.setRate(pickup);
6  IO.println("total en tienda: "
7          + checkout.total(heavy));
```

bash · final de la salida

```
$ java -cp build ShippingDemo
...
total por peso: S/ 197.00
total en tienda: S/ 180.00
```

- `heavy`: 3200 g son 4 kilos empezados, $5 + 4 \times 3 = \text{S/}\ 17$ de envío. La strategy cambia mientras el programa corre.

Ejemplo: tres tarifas y una lambda

ShippingDemo.java · fragmento

```
1  ShippingRate pickup = p -> Money.zero();
2  var checkout = new Checkout(rates.get(1));
3  IO.println("total por peso: "
4          + checkout.total(heavy));
5  checkout.setRate(pickup);
6  IO.println("total en tienda: "
7          + checkout.total(heavy));
```

bash · final de la salida

```
$ java -cp build ShippingDemo
...
total por peso: S/ 197.00
total en tienda: S/ 180.00
```

- `heavy`: 3200 g son 4 kilos empezados, $5 + 4 \times 3 = \text{S/}\ 17$ de envío. La strategy cambia mientras el programa corre.
- Un solo método abstracto: interfaz funcional. Una strategy simple puede ser una lambda.

Ya lo conocían: `Comparator`

- `list.sort(comparador)`: `sort` es el context; el algoritmo de ordenamiento no cambia.

Ya lo conocían: `Comparator`

- `list.sort(comparador)`: `sort` es el context; el algoritmo de ordenamiento no cambia.
- Lo que cambia es cómo comparar dos elementos, y eso se lo pasan ustedes.

Ya lo conocían: `Comparator`

- `list.sort(comparador)`: `sort` es el context; el algoritmo de ordenamiento no cambia.
- Lo que cambia es cómo comparar dos elementos, y eso se lo pasan ustedes.
- Las reglas de precio y descuento de P2: una interfaz y una clase por promoción.

Ya lo conocían: `Comparator`

- `list.sort(comparador)`: `sort` es el context; el algoritmo de ordenamiento no cambia.
- Lo que cambia es cómo comparar dos elementos, y eso se lo pasan ustedes.
- Las reglas de precio y descuento de P2: una interfaz y una clase por promoción.

En Java

`Comparator` para `sort` y `TreeMap`; el `Predicate` de `removeIf` y `filter`; la `Function` de `map`. Cada interfaz funcional que reciben las colecciones y los streams es una strategy.

¿Strategy o herencia?

¿Y si `Checkout` fuera abstracta, con `FlatCheckout`, `WeightCheckout`...?

Con herencia

Con Strategy

¿Strategy o herencia?

¿Y si `Checkout` fuera abstracta, con `FlatCheckout`, `WeightCheckout`...?

Con herencia

- La tarifa queda fija al crear el objeto.

Con Strategy

- `setRate` la cambia en cualquier momento.

¿Strategy o herencia?

¿Y si `Checkout` fuera abstracta, con `FlatCheckout`, `WeightCheckout`...?

Con herencia

- La tarifa queda fija al crear el objeto.
- Dos cosas que varían (envío y forma de pago): una subclase por cada combinación.

Con Strategy

- `setRate` la cambia en cualquier momento.
- Cada strategy varía por su lado.

¿Strategy o herencia?

¿Y si `Checkout` fuera abstracta, con `FlatCheckout`, `WeightCheckout`...?

Con herencia

- La tarifa queda fija al crear el objeto.
- Dos cosas que varían (envío y forma de pago): una subclase por cada combinación.
- Para probar la tarifa hay que crear un `Checkout`.

Con Strategy

- `setRate` la cambia en cualquier momento.
- Cada strategy varía por su lado.
- La strategy se prueba sola.

¿Strategy o herencia?

¿Y si `Checkout` fuera abstracta, con `FlatCheckout`, `WeightCheckout`...?

Con herencia

- La tarifa queda fija al crear el objeto.
- Dos cosas que varían (envío y forma de pago): una subclase por cada combinación.
- Para probar la tarifa hay que crear un `Checkout`.

Con Strategy

- `setRate` la cambia en cualquier momento.
- Cada strategy varía por su lado.
- La strategy se prueba sola.

La regla de la semana 7: componer cuando lo que varía es un comportamiento, no el tipo de objeto.

Cuándo no usar Strategy

Error común

Una interfaz, una clase y una factory para **un algoritmo que es uno solo** y nadie ha pedido cambiar. Una tienda que siempre cobra S/ 10 de envío necesita una constante, no `ShippingRate`.

Cuándo no usar Strategy

Error común

Una interfaz, una clase y una factory para **un algoritmo que es uno solo** y nadie ha pedido cambiar. Una tienda que siempre cobra S/ 10 de envío necesita una constante, no `ShippingRate`.

- Se justifica con **dos o más** algoritmos, o con un cambio concreto en camino.

Cuándo no usar Strategy

Error común

Una interfaz, una clase y una factory para **un algoritmo que es uno solo** y nadie ha pedido cambiar. Una tienda que siempre cobra S/ 10 de envío necesita una constante, no `ShippingRate`.

- Se justifica con **dos o más** algoritmos, o con un cambio concreto en camino.
- Si el segundo aparece después: extraer interfaz y mover el código (semana 13) toma minutos con las pruebas en su lugar.

Cuándo no usar Strategy

Error común

Una interfaz, una clase y una factory para **un algoritmo que es uno solo** y nadie ha pedido cambiar. Una tienda que siempre cobra S/ 10 de envío necesita una constante, no `ShippingRate`.

- Se justifica con **dos o más** algoritmos, o con un cambio concreto en camino.
- Si el segundo aparece después: extraer interfaz y mover el código (semana 13) toma minutos con las pruebas en su lugar.
- Tres variantes fijas que no se combinan: un `switch` sobre un `enum` puede ser más claro.

SECCIÓN 03

Observer

03



El problema: un subject que conoce a todos

Thermometer.java · versión directa

```java
public void record(double celsius) {
    display.show(place, celsius);
    alarm.check(place, celsius);
    // mañana: log, SMS, tablero...
}
```

Un termómetro vigila la refrigeradora de vacunas de una posta: la cadena de frío exige entre 2 y 8 °C.

El problema: un subject que conoce a todos

Thermometer.java · versión directa

```java
public void record(double celsius) {
    display.show(place, celsius);
    alarm.check(place, celsius);
    // mañana: log, SMS, tablero...
}
```

Un termómetro vigila la refrigeradora de vacunas de una posta: la cadena de frío exige entre 2 y 8 °C.

- El termómetro conoce la clase concreta de cada interesado.

El problema: un subject que conoce a todos

Thermometer.java · versión directa

```java
public void record(double celsius) {
    display.show(place, celsius);
    alarm.check(place, celsius);
    // mañana: log, SMS, tablero...
}
```

Un termómetro vigila la refrigeradora de vacunas de una posta: la cadena de frío exige entre 2 y 8 °C.

- El termómetro conoce la clase concreta de cada interesado.
- Cada interesado nuevo: abrir el termómetro, un campo y una línea más.

El problema: un subject que conoce a todos

Thermometer.java · versión directa

```java
public void record(double celsius) {
    display.show(place, celsius);
    alarm.check(place, celsius);
    // mañana: log, SMS, tablero...
}
```

Un termómetro vigila la refrigeradora de vacunas de una posta: la cadena de frío exige entre 2 y 8 °C.

- El termómetro conoce la clase concreta de cada interesado.
- Cada interesado nuevo: abrir el termómetro, un campo y una línea más.
- No se puede probar sin pantalla ni alarma.

El problema: un subject que conoce a todos

Thermometer.java · versión directa

```java
public void record(double celsius) {
    display.show(place, celsius);
    alarm.check(place, celsius);
    // mañana: log, SMS, tablero...
}
```

Un termómetro vigila la refrigeradora de vacunas de una posta: la cadena de frío exige entre 2 y 8 °C.

- El termómetro conoce la clase concreta de cada interesado.
- Cada interesado nuevo: abrir el termómetro, un campo y una línea más.
- No se puede probar sin pantalla ni alarma.
- Lo estable depende de lo que cambia: acoplamiento al revés.

La solución: una lista de observers

Thermometer.java · esqueleto

```java
class Thermometer {
  private final List<TemperatureListener>
      listeners = new ArrayList<>();
  // addListener(l), removeListener(l)
  void record(double celsius) {
    var reading = new Reading(place, celsius);
    for (var l : List.copyOf(listeners)) {
      l.onReading(reading);
    }
  }
}
```

- El **subject** guarda una lista de **observers**.

La solución: una lista de observers

Thermometer.java · esqueleto

```java
class Thermometer {
  private final List<TemperatureListener>
      listeners = new ArrayList<>();
  // addListener(l), removeListener(l)
  void record(double celsius) {
    var reading = new Reading(place, celsius);
    for (var l : List.copyOf(listeners)) {
      l.onReading(reading);
    }
  }
}
```

- El **subject** guarda una lista de **observers**.
- Del observer conoce solo una interfaz:

```
void onReading(Reading r)
```

La solución: una lista de observers

Thermometer.java · esqueleto

```java
class Thermometer {
  private final List<TemperatureListener>
      listeners = new ArrayList<>();
  // addListener(l), removeListener(l)
  void record(double celsius) {
    var reading = new Reading(place, celsius);
    for (var l : List.copyOf(listeners)) {
      l.onReading(reading);
    }
  }
}
```

- El **subject** guarda una lista de **observers**.
- Del observer conoce solo una interfaz:
  `void onReading(Reading r)`
- Un cliente registra los observers con `addListener`.

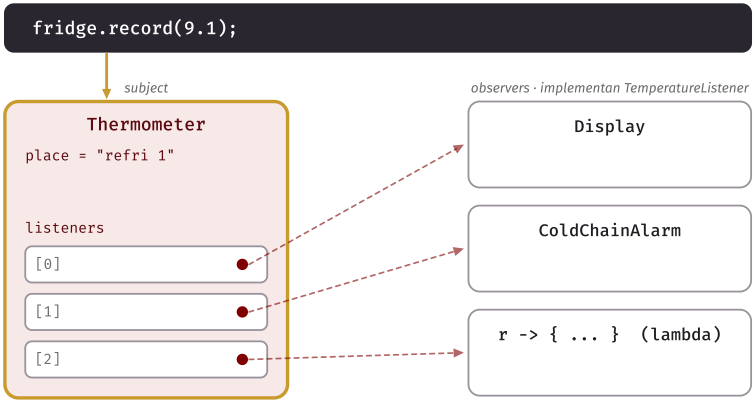
La solución: una lista de observers

Thermometer.java · esqueleto

```java
class Thermometer {
  private final List<TemperatureListener>
      listeners = new ArrayList<>();
  // addListener(l), removeListener(l)
  void record(double celsius) {
    var reading = new Reading(place, celsius);
    for (var l : List.copyOf(listeners)) {
      l.onReading(reading);
    }
  }
}
```

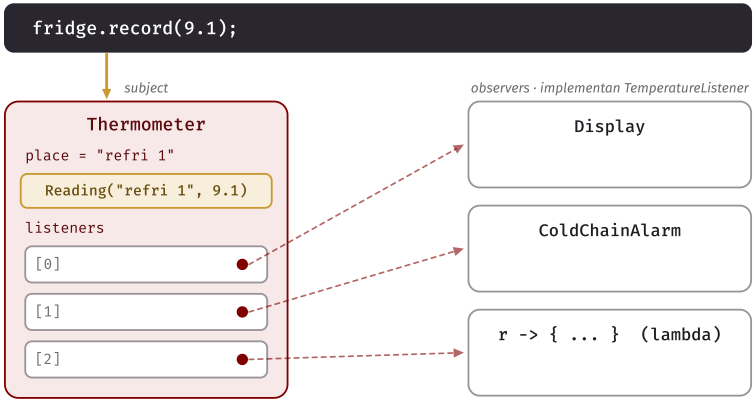
- El **subject** guarda una lista de **observers**.
- Del observer conoce solo una interfaz:
  `void onReading(Reading r)`
- Un cliente registra los observers con `addListener`.
- `Thermometer` compila sin `Display` ni `ColdChainAlarm`.

Observer en acción



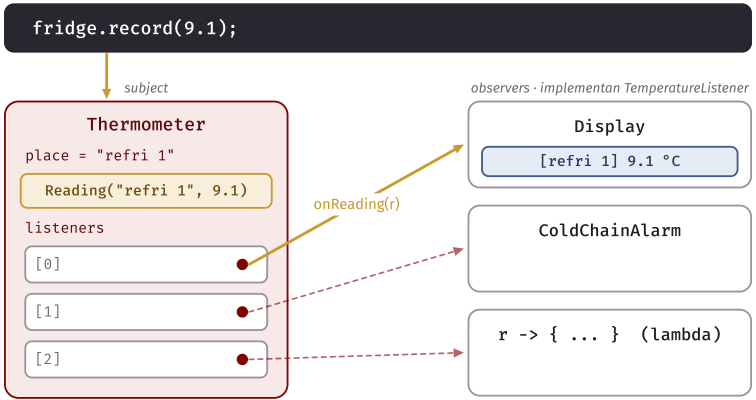
1. llega una lectura nueva al subject

Observer en acción



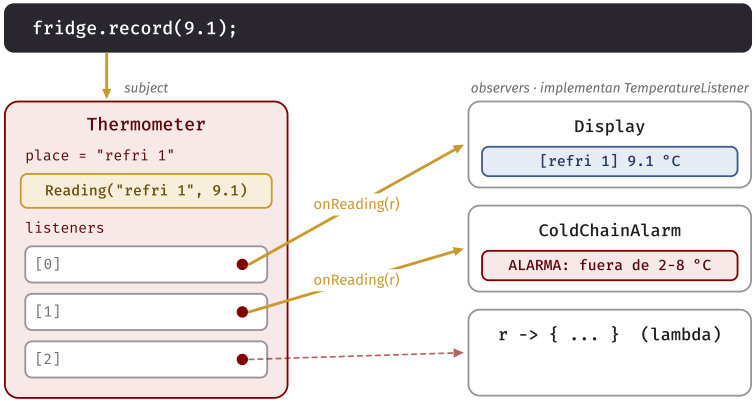
2. el subject arma el aviso: un Reading con lugar y grados

Observer en acción



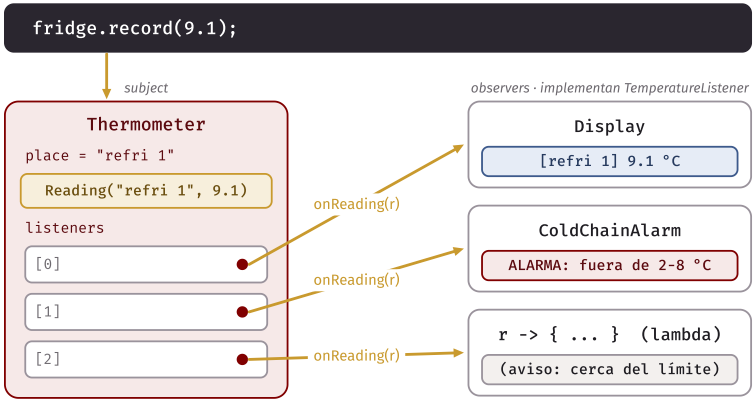
3. recorre su lista y avisa al primer observer

Observer en acción



4. el mismo mensaje al segundo: la alarma se enciende

Observer en acción



el subject solo conoce TemperatureListener: no sabe qué hace cada observer

Ejemplo: la refrigeradora de vacunas

ObserverDemo.java · fragmento

```java
var fridge = new Thermometer("refri 1");
var alarm = new ColdChainAlarm();
fridge.addListener(new Display());
fridge.addListener(alarm);
fridge.addListener(r -> {  // una lambda
    if (r.celsius() > 7.5) {
        IO.println("  (aviso: ...)");
    }
});
fridge.record(4.5);
fridge.record(7.8);  fridge.record(9.1);
```

bash

```
$ java -cp build ObserverDemo
[refri 1] 4.5 °C
[refri 1] 7.8 °C
  (aviso: cerca del límite)
[refri 1] 9.1 °C
ALARMA en refri 1: 9.1 °C
    fuera de 2-8 °C
  (aviso: cerca del límite)
alarmas: 1
```

Los detalles que importan (1)

¿Quién se da de baja?

Un observer que nadie da de baja sigue recibiendo avisos, y el subject lo mantiene vivo: el garbage collector no puede liberarlo. Olvidar `removeListener` es un memory leak en Java.

Los detalles que importan (1)

¿Quién se da de baja?

Un observer que nadie da de baja sigue recibiendo avisos, y el subject lo mantiene vivo: el garbage collector no puede liberarlo. Olvidar `removeListener` es un memory leak en Java.

¿Qué viaja en el aviso?

Push: el subject manda los datos (`Reading`). **Pull**: manda solo "cambié" y cada observer pregunta lo que necesita.

Un record como evento permite agregar un campo (la hora) sin cambiar la interfaz.

Los detalles que importan (2): un observer que falla

Thermometer.java · dentro de record

```java
RuntimeException failure = null;
for (var l : List.copyOf(listeners)) {
    try {
        l.onReading(reading);
    } catch (RuntimeException e) {
        if (failure == null) failure = e;
        else failure.addSuppressed(e);
    }
}
if (failure != null) throw failure;
```

En el código, con llaves en cada `if`.

- Con el `for` simple, la excepción sale de `record` y los que faltaban no se enteran.

Los detalles que importan (2): un observer que falla

Thermometer.java · dentro de record

```java
RuntimeException failure = null;
for (var l : List.copyOf(listeners)) {
    try {
        l.onReading(reading);
    } catch (RuntimeException e) {
        if (failure == null) failure = e;
        else failure.addSuppressed(e);
    }
}
if (failure != null) throw failure;
```

En el código, con llaves en cada `if`.

- Con el `for` simple, la excepción sale de `record` y los que faltaban no se enteran.
- Si la pantalla falla, la alarma no suena.

Los detalles que importan (2): un observer que falla

Thermometer.java · dentro de record

```java
RuntimeException failure = null;
for (var l : List.copyOf(listeners)) {
    try {
        l.onReading(reading);
    } catch (RuntimeException e) {
        if (failure == null) failure = e;
        else failure.addSuppressed(e);
    }
}
if (failure != null) throw failure;
```

En el código, con llaves en cada `if`.

- Con el `for` simple, la excepción sale de `record` y los que faltaban no se enteran.
- Si la pantalla falla, la alarma no suena.
- Aquí: avisar a todos y relanzar la primera, con las demás suprimidas (semana 9).

Los detalles que importan (2): un observer que falla

Thermometer.java · dentro de record

```java
RuntimeException failure = null;
for (var l : List.copyOf(listeners)) {
    try {
        l.onReading(reading);
    } catch (RuntimeException e) {
        if (failure == null) failure = e;
        else failure.addSuppressed(e);
    }
}
if (failure != null) throw failure;
```

En el código, con llaves en cada `if`.

- Con el `for` simple, la excepción sale de `record` y los que faltaban no se enteran.
- Si la pantalla falla, la alarma no suena.
- Aquí: avisar a todos y relanzar la primera, con las demás suprimidas (semana 9).
- Es una decisión, no la única. Lo importante: decidirla y probarla.

Los detalles que importan (3): darse de baja durante el aviso

FirstAlert.java

```java
public void onReading(Reading r) {
    if (r.celsius() > limit) {
        IO.println("primer aviso: ...");
        subject.removeListener(this);
    }
}
```

bash · última línea de ObserverDemo

```
ingenuo: java.util.
    ConcurrentModificationException
```

- `NaiveThermometer` recorre la lista original.

Los detalles que importan (3): darse de baja durante el aviso

FirstAlert.java

```java
public void onReading(Reading r) {
    if (r.celsius() > limit) {
        IO.println("primer aviso: ...");
        subject.removeListener(this);
    }
}
```

bash · última línea de ObserverDemo

```
ingenuo: java.util.
    ConcurrentModificationException
```

- `NaiveThermometer` recorre la lista original.
- El observer la modifica desde adentro de `onReading`.

Los detalles que importan (3): darse de baja durante el aviso

FirstAlert.java

```java
public void onReading(Reading r) {
    if (r.celsius() > limit) {
        IO.println("primer aviso: ...");
        subject.removeListener(this);
    }
}
```

bash · última línea de ObserverDemo

```
ingenuo: java.util.
    ConcurrentModificationException
```

- `NaiveThermometer` recorre la lista original.
- El observer la modifica desde adentro de `onReading`.

Error común

Modificar una lista mientras se la recorre con for-each, aunque haya un solo hilo. Solución: recorrer `List.copyOf(listeners)`.

En Java

- `java.util.Observable` y `Observer`, desde Java 1.0: deprecated desde Java 9. El aviso no dice qué cambió y el orden no está especificado.

En Java

- `java.util.Observable` y `Observer`, desde Java 1.0: deprecated desde Java 9. El aviso no dice qué cambió y el orden no está especificado.
- Hoy: una interfaz de observer a la medida, como `TemperatureListener`.

En Java

- `java.util.Observable` y `Observer`, desde Java 1.0: deprecated desde Java 9. El aviso no dice qué cambió y el orden no está especificado.
- Hoy: una interfaz de observer a la medida, como `TemperatureListener`.
- O `java.beans.PropertyChangeSupport`: `addPropertyChangeListener`, `firePropertyChange`. Hace el trabajo del subject para cualquier clase.

En Java

- `java.util.Observable` y `Observer`, desde Java 1.0: deprecated desde Java 9. El aviso no dice qué cambió y el orden no está especificado.
- Hoy: una interfaz de observer a la medida, como `TemperatureListener`.
- O `java.beans.PropertyChangeSupport`: `addPropertyChangeListener`, `firePropertyChange`. Hace el trabajo del subject para cualquier clase.
- Las interfaces gráficas usan Observer en todas partes: un botón es un subject; quien escucha sus clics, un observer. Semana 16.

Probar el subject con un spy

ThermometerTest.java · fragmento

```java
class RecordingListener implements ... {
    List<Reading> received = new ArrayList<>();
    // onReading(r) hace received.add(r)
}
@Test void removedListenerHearsNothingMore() {
    var t = new Thermometer("refri");
    var spy = new RecordingListener();
    t.addListener(spy);     t.record(4.0);
    t.removeListener(spy);  t.record(9.0);
    assertEquals(1, spy.received.size());
}
```

- El **spy** de la semana 7: recuerda lo que recibió.

Probar el subject con un spy

ThermometerTest.java · fragmento

```java
class RecordingListener implements ... {
    List<Reading> received = new ArrayList<>();
    // onReading(r) hace received.add(r)
}
@Test void removedListenerHearsNothingMore() {
    var t = new Thermometer("refri");
    var spy = new RecordingListener();
    t.addListener(spy);     t.record(4.0);
    t.removeListener(spy);  t.record(9.0);
    assertEquals(1, spy.received.size());
}
```

- El **spy** de la semana 7: recuerda lo que recibió.
- Sin pantalla ni alarma.

Probar el subject con un spy

ThermometerTest.java · fragmento

```java
class RecordingListener implements ... {
    List<Reading> received = new ArrayList<>();
    // onReading(r) hace received.add(r)
}
@Test void removedListenerHearsNothingMore() {
    var t = new Thermometer("refri");
    var spy = new RecordingListener();
    t.addListener(spy);     t.record(4.0);
    t.removeListener(spy);  t.record(9.0);
    assertEquals(1, spy.received.size());
}
```

- El **spy** de la semana 7: recuerda lo que recibió.
- Sin pantalla ni alarma.
- El subject se prueba solo porque depende de una interfaz.

Cuándo no usar Observer

- **Un solo interesado**, siempre el mismo: una llamada directa se lee mejor que una lista de un elemento.

Cuándo no usar Observer

- **Un solo interesado**, siempre el mismo: una llamada directa se lee mejor que una lista de un elemento.
- **El orden importa**: si la alarma debe actuar antes que el registro, depender del orden de registro es frágil.

Cuándo no usar Observer

- **Un solo interesado**, siempre el mismo: una llamada directa se lee mejor que una lista de un elemento.
- **El orden importa**: si la alarma debe actuar antes que el registro, depender del orden de registro es frágil.
- **Cadenas de avisos**: un observer cambia otro subject, que avisa a otros… El código ya no se lee de arriba abajo.

Cuándo no usar Observer

- **Un solo interesado**, siempre el mismo: una llamada directa se lee mejor que una lista de un elemento.
- **El orden importa**: si la alarma debe actuar antes que el registro, depender del orden de registro es frágil.
- **Cadenas de avisos**: un observer cambia otro subject, que avisa a otros… El código ya no se lee de arriba abajo.

> **Observer paga cuando…**
>
> …el conjunto de interesados cambia, o el subject no debe depender de ellos (están en otra capa).

SECCIÓN 04

Factory

04

Factories: sacar la decisión de crear

`new Clase(...)` ata el código a una clase concreta. Casi siempre está bien. Una factory sirve cuando crear necesita:

Factories: sacar la decisión de crear

`new Clase(...)` ata el código a una clase concreta. Casi siempre está bien. Una factory sirve cuando crear necesita:

- **validar** o normalizar antes,

Factories: sacar la decisión de crear

`new Clase(...)` ata el código a una clase concreta. Casi siempre está bien. Una factory sirve cuando crear necesita:

- **validar** o normalizar antes,
- elegir la clase concreta **según un dato**,

Factories: sacar la decisión de crear

`new Clase(...)` ata el código a una clase concreta. Casi siempre está bien. Una factory sirve cuando crear necesita:

- **validar** o normalizar antes,
- elegir la clase concreta **según un dato**,
- hacerse **en un solo lugar**, porque se repite.

Factories: sacar la decisión de crear

`new Clase(...)` ata el código a una clase concreta. Casi siempre está bien. Una factory sirve cuando crear necesita:

- **validar** o normalizar antes,
- elegir la clase concreta **según un dato**,
- hacerse **en un solo lugar**, porque se repite.

Tres formas

1. un **static factory method**,
2. una **factory**,
3. **Factory Method**, el del libro de 1994.

Static factory methods: `Money`

¿Qué gana frente a `new`?

Money.java · fragmento

```java
public final class Money {
  private static final Money ZERO_PEN = ...;
  static Money of(long cents, String cur) {
    if (cents < 0) { throw ... }
    String code = cur.strip().toUpperCase();
    if (cents == 0 && code.equals("PEN")) {
      return ZERO_PEN;
    }
    return new Money(cents, code);
  }
}
```

Constructor `private`. Además: `parse("12.50")` y `zero()`.

Static factory methods: `Money`

Money.java · fragmento

```java
public final class Money {
  private static final Money ZERO_PEN = ...;
  static Money of(long cents, String cur) {
    if (cents < 0) { throw ... }
    String code = cur.strip().toUpperCase();
    if (cents == 0 && code.equals("PEN")) {
      return ZERO_PEN;
    }
    return new Money(cents, code);
  }
}
```

Constructor `private`. Además: `parse("12.50")` y `zero()`.

¿Qué gana frente a `new`?

- **Tiene nombre**: `parse`, `zero`.

Static factory methods: `Money`

Money.java · fragmento

```java
public final class Money {
  private static final Money ZERO_PEN = ...;
  static Money of(long cents, String cur) {
    if (cents < 0) { throw ... }
    String code = cur.strip().toUpperCase();
    if (cents == 0 && code.equals("PEN")) {
      return ZERO_PEN;
    }
    return new Money(cents, code);
  }
}
```

Constructor `private`. Además: `parse("12.50")` y `zero()`.

¿Qué gana frente a `new`?

- **Tiene nombre**: `parse`, `zero`.
- Puede devolver un objeto que ya existe.

Static factory methods: `Money`

Money.java · fragmento

```java
public final class Money {
  private static final Money ZERO_PEN = ...;
  static Money of(long cents, String cur) {
    if (cents < 0) { throw ... }
    String code = cur.strip().toUpperCase();
    if (cents == 0 && code.equals("PEN")) {
      return ZERO_PEN;
    }
    return new Money(cents, code);
  }
}
```

Constructor `private`. Además: `parse("12.50")` y `zero()`.

¿Qué gana frente a `new`?

- **Tiene nombre**: `parse`, `zero`.
- Puede devolver un objeto que ya existe.
- Puede validar y normalizar: `" pen "` se vuelve `"PEN"`.

Static factory methods: `Money`

Money.java · fragmento

```java
public final class Money {
  private static final Money ZERO_PEN = ...;
  static Money of(long cents, String cur) {
    if (cents < 0) { throw ... }
    String code = cur.strip().toUpperCase();
    if (cents == 0 && code.equals("PEN")) {
      return ZERO_PEN;
    }
    return new Money(cents, code);
  }
}
```

Constructor `private`. Además: `parse("12.50")` y `zero()`.

¿Qué gana frente a `new`?

- **Tiene nombre**: `parse`, `zero`.
- Puede devolver un objeto que ya existe.
- Puede validar y normalizar: `" pen "` se vuelve `"PEN"`.
- Puede devolver un subtipo.

En Java: `Integer.valueOf` y `List.of`

jshell

```
jshell> Integer.valueOf(127) == Integer.valueOf(127)
$1 ==> true
jshell> Integer.valueOf(128) == Integer.valueOf(128)
$2 ==> false
jshell> List.of(1, 2).getClass()
$3 ==> class java.util.ImmutableCollections$List12
jshell> List.of(1, 2, 3).getClass()
$4 ==> class java.util.ImmutableCollections$ListN
```

En Java: `Integer.valueOf` y `List.of`

jshell

```
jshell> Integer.valueOf(127) == Integer.valueOf(127)
$1 ==> true
jshell> Integer.valueOf(128) == Integer.valueOf(128)
$2 ==> false
jshell> List.of(1, 2).getClass()
$3 ==> class java.util.ImmutableCollections$List12
jshell> List.of(1, 2, 3).getClass()
$4 ==> class java.util.ImmutableCollections$ListN
```

- `valueOf` guarda en caché al menos de −128 a 127 (puede guardar más). No dependan de `==`: se compara con `equals` (semana 5).

En Java: `Integer.valueOf` y `List.of`

jshell

```
jshell> Integer.valueOf(127) == Integer.valueOf(127)
$1 ==> true
jshell> Integer.valueOf(128) == Integer.valueOf(128)
$2 ==> false
jshell> List.of(1, 2).getClass()
$3 ==> class java.util.ImmutableCollections$List12
jshell> List.of(1, 2, 3).getClass()
$4 ==> class java.util.ImmutableCollections$ListN
```

- `valueOf` guarda en caché al menos de −128 a 127 (puede guardar más). No dependan de `==`: se compara con `equals` (semana 5).
- `List.of` devuelve clases distintas según cuántos elementos recibe. Ninguna de las dos cosas se puede hacer con un constructor.

Una factory: la decisión en un solo lugar

Notification.java · fragmento

```java
interface Notification {
  String send();

  static Notification of(String channel,
      String to, String text) {
    return switch (channel) {
      case "email" -> new EmailNotification(..);
      case "sms"   -> new SmsNotification(..);
      default -> throw new IllegalArg...;
    };
  }
}
```

- Quien llama recibe un `Notification`: no sabe qué clase hay detrás.

Una factory: la decisión en un solo lugar

Notification.java · fragmento

```java
interface Notification {
  String send();

  static Notification of(String channel,
      String to, String text) {
    return switch (channel) {
      case "email" -> new EmailNotification(..);
      case "sms"   -> new SmsNotification(..);
      default -> throw new IllegalArg...;
    };
  }
}
```

- Quien llama recibe un `Notification`: no sabe qué clase hay detrás.

Error común

Un `switch` que crea objetos, **copiado** en el lector de archivos, en el menú y en las pruebas. La decisión va en **un solo lugar**.

Factory Method: crear dentro de un algoritmo

LineImporter.java · resumido

```java
abstract class LineImporter<T> {
  final List<T> importAll(List<String> lines) {
    List<T> result = new ArrayList<>();
    for (String line : lines) {
      // saltar vacías y #, partir, verificar
      result.add(create(fields));
    }
    return result;
  }
  protected abstract T create(String[] fields);
}
```

- No es un método estático: es un método al que cada subclase le hace override.

Factory Method: crear dentro de un algoritmo

LineImporter.java · resumido

```java
abstract class LineImporter<T> {
  final List<T> importAll(List<String> lines) {
    List<T> result = new ArrayList<>();
    for (String line : lines) {
      // saltar vacías y #, partir, verificar
      result.add(create(fields));
    }
    return result;
  }
  protected abstract T create(String[] fields);
}
```

- No es un método estático: es un método al que cada subclase le hace override.
- El recorrido es siempre el mismo: `importAll` es `final`.

Factory Method: crear dentro de un algoritmo

LineImporter.java · resumido

```java
abstract class LineImporter<T> {
  final List<T> importAll(List<String> lines) {
    List<T> result = new ArrayList<>();
    for (String line : lines) {
      // saltar vacías y #, partir, verificar
      result.add(create(fields));
    }
    return result;
  }
  protected abstract T create(String[] fields);
}
```

- No es un método estático: es un método al que cada subclase le hace override.
- El recorrido es siempre el mismo: `importAll` es `final`.
- Lo único que cambia es qué objeto se crea con los campos: `create`.

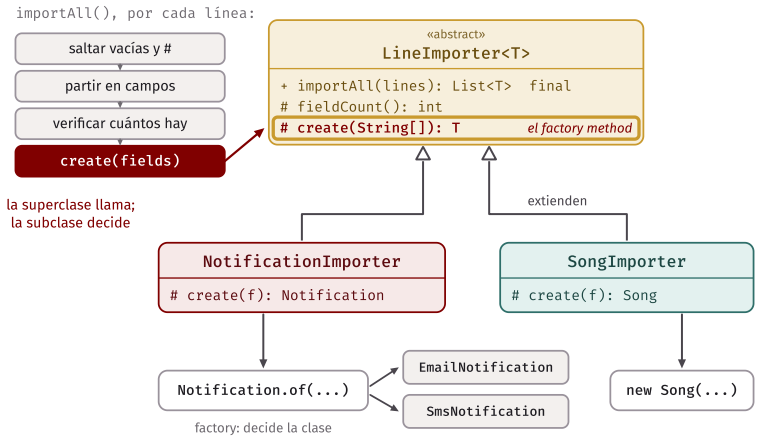
Factory Method: crear dentro de un algoritmo

LineImporter.java · resumido

```java
abstract class LineImporter<T> {
  final List<T> importAll(List<String> lines) {
    List<T> result = new ArrayList<>();
    for (String line : lines) {
      // saltar vacías y #, partir, verificar
      result.add(create(fields));
    }
    return result;
  }
  protected abstract T create(String[] fields);
}
```

- No es un método estático: es un método al que cada subclase le hace override.
- El recorrido es siempre el mismo: `importAll` es `final`.
- Lo único que cambia es qué objeto se crea con los campos: `create`.
- Nos va a servir para leer archivos la semana 15.

Estructura de Factory Method



Una subclase decide

NotificationImporter.java

```java
class NotificationImporter
    extends LineImporter<Notification> {
  protected int fieldCount() { return 3; }

  protected Notification create(String[] f){
    return Notification.of(f[0].strip(),
        f[1].strip(), f[2].strip());
  }
}
```

bash · FactoryDemo

```
email a alice@uni.pe: Su
    pedido salió hoy
SMS a 987654321: Su pedido
    llega maña...
email a bob@uni.pe: Gracias
    por su compra
```

Una subclase decide

NotificationImporter.java

```java
class NotificationImporter
    extends LineImporter<Notification> {
  protected int fieldCount() { return 3; }

  protected Notification create(String[] f){
    return Notification.of(f[0].strip(),
        f[1].strip(), f[2].strip());
  }
}
```

bash · FactoryDemo

```
email a alice@uni.pe: Su
    pedido salió hoy
SMS a 987654321: Su pedido
    llega maña...
email a bob@uni.pe: Gracias
    por su compra
```

Las líneas vacías y los comentarios los salta la superclase; un canal `fax` da `error: canal desconocido: fax`. `SongImporter` crea canciones con el mismo recorrido.

Tres cosas parecidas, con nombres distintos

Nombre	Ejemplo	Qué hace
Static factory method	`Money.of`	Reemplaza a un constructor

Tres cosas parecidas, con nombres distintos

Nombre	Ejemplo	Qué hace
Static factory method	`Money.of`	Reemplaza a un constructor
Factory	`Notification.of`	Decide en un solo lugar qué clase crear según un dato

Tres cosas parecidas, con nombres distintos

Nombre	Ejemplo	Qué hace
Static factory method	`Money.of`	Reemplaza a un constructor
Factory	`Notification.of`	Decide en un solo lugar qué clase crear según un dato
Factory Method	`LineImporter.create`	La superclase lo llama; cada subclase decide qué crear

Tres cosas parecidas, con nombres distintos

Nombre	Ejemplo	Qué hace
Static factory method	`Money.of`	Reemplaza a un constructor
Factory	`Notification.of`	Decide en un solo lugar qué clase crear según un dato
Factory Method	`LineImporter.create`	La superclase lo llama; cada subclase decide qué crear

Cuando alguien dice “factory”, pregunten cuál de las tres.

Cuándo no usar una factory

- Una sola clase y un constructor que no valida nada: `new` es lo más claro. `new ArrayList<>()` no necesita factory.

Cuándo no usar una factory

- Una sola clase y un constructor que no valida nada: `new` es lo más claro. `new ArrayList<>()` no necesita factory.
- “Factories de factories” para no depender de `new` en un programa pequeño: indirección sin cliente.

Cuándo no usar una factory

- Una sola clase y un constructor que no valida nada: `new` es lo más claro. `new ArrayList<>()` no necesita factory.
- "Factories de factories" para no depender de `new` en un programa pequeño: indirección sin cliente.
- Factory Method paga con un algoritmo común de verdad y al menos dos subclases.

Cuándo no usar una factory

- Una sola clase y un constructor que no valida nada: `new` es lo más claro. `new ArrayList<>()` no necesita factory.
- "Factories de factories" para no depender de `new` en un programa pequeño: indirección sin cliente.
- Factory Method paga con un algoritmo común de verdad y al menos dos subclases.
- El static factory method cuesta: no se distingue de otros `static`, y una clase con constructor privado no se puede extender.

SECCIÓN 05

Composite

05

El problema: listas que contienen listas

- Una lista de reproducción contiene canciones y otras listas.

El problema: listas que contienen listas

- Una lista de reproducción contiene canciones y otras listas.
- “Fiesta” contiene “Criollas”, dos canciones y “Cumbias”.

El problema: listas que contienen listas

- Una lista de reproducción contiene canciones y otras listas.
- “Fiesta” contiene “Criollas”, dos canciones y “Cumbias”.
- Queremos la duración de cualquier cosa: una canción, una lista, una lista de listas.

El problema: listas que contienen listas

- Una lista de reproducción contiene canciones y otras listas.
- “Fiesta” contiene “Criollas”, dos canciones y “Cumbias”.
- Queremos la duración de cualquier cosa: una canción, una lista, una lista de listas.

Sin el patrón

Cada operación pregunta a cada paso qué tiene en la mano: ¿canción o lista? Y lo repite para duración, contar, imprimir…

La solución: una interfaz para la leaf y el composite

Playable.java · Song.java

```java
sealed interface Playable
        permits Song, Playlist {
    String title();
    int seconds();
    default String length() { ... }  // "m:ss"
}
record Song(String title, String artist,
        int seconds) implements Playable { }
```

- **Component**: la interfaz común.

La solución: una interfaz para la leaf y el composite

Playable.java · Song.java

```java
sealed interface Playable
        permits Song, Playlist {
    String title();
    int seconds();
    default String length() { ... }  // "m:ss"
}
record Song(String title, String artist,
        int seconds) implements Playable { }
```

- **Component**: la interfaz común.
- **Leaf**: Song. Los métodos que genera el record ya implementan la interfaz.

La solución: una interfaz para la leaf y el composite

Playable.java · Song.java

```java
sealed interface Playable
        permits Song, Playlist {
    String title();
    int seconds();
    default String length() { ... }  // "m:ss"
}
record Song(String title, String artist,
        int seconds) implements Playable { }
```

- **Component**: la interfaz común.
- **Leaf**: `Song`. Los métodos que genera el record ya implementan la interfaz.
- `length()`: un método `default`, escrito una vez para los dos casos.

El composite: `Playlist`

Playlist.java

```java
record Playlist(String title,
        List<Playable> entries)
        implements Playable {
    public int seconds() {
        int total = 0;
        for (Playable entry : entries) {
            total += entry.seconds();
        }
        return total;
    }
}
```

- Guarda components, sin saber cuáles son leaves, y les pide la respuesta.

El composite: `Playlist`

Playlist.java

```java
record Playlist(String title,
        List<Playable> entries)
        implements Playable {
    public int seconds() {
        int total = 0;
        for (Playable entry : entries) {
            total += entry.seconds();
        }
        return total;
    }
}
```

- Guarda components, sin saber cuáles son leaves, y les pide la respuesta.
- `entry.seconds()`: una canción (caso base) u otra lista (caso recursivo). El dynamic dispatch elige.

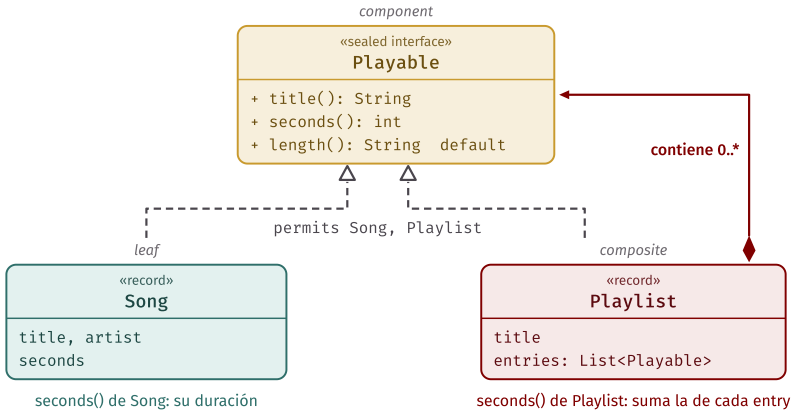
El composite: `Playlist`

Playlist.java

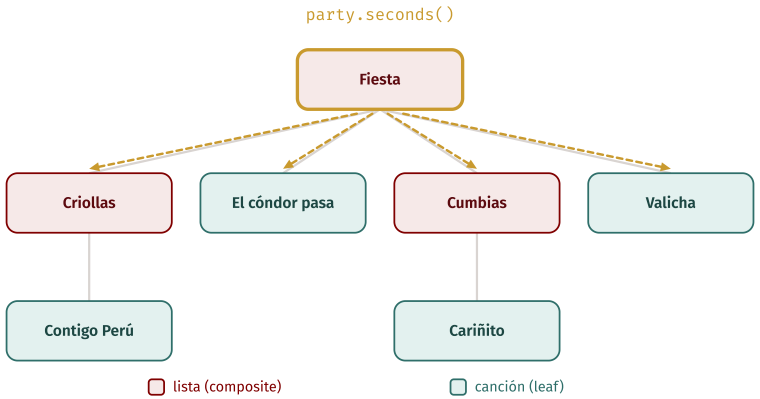
```java
record Playlist(String title,
        List<Playable> entries)
        implements Playable {
    public int seconds() {
        int total = 0;
        for (Playable entry : entries) {
            total += entry.seconds();
        }
        return total;
    }
}
```

- Guarda components, sin saber cuáles son leaves, y les pide la respuesta.
- `entry.seconds()`: una canción (caso base) u otra lista (caso recursivo). El dynamic dispatch elige.
- El constructor compacto hace `List.copyOf`: el árbol es inmutable. Si fuera mutable, cuidado con una lista que se contenga a sí misma.

Estructura de Composite

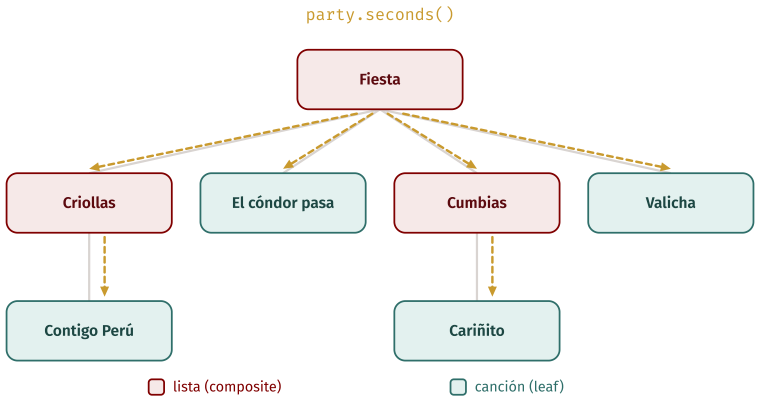


La duración sube por el árbol



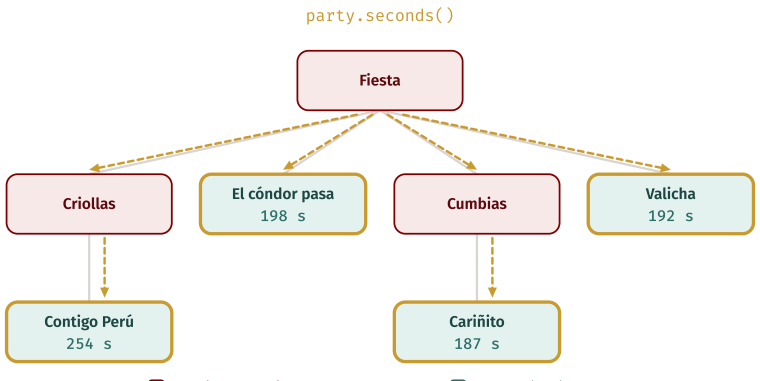
1. la raíz pide seconds() a cada uno de sus hijos

La duración sube por el árbol



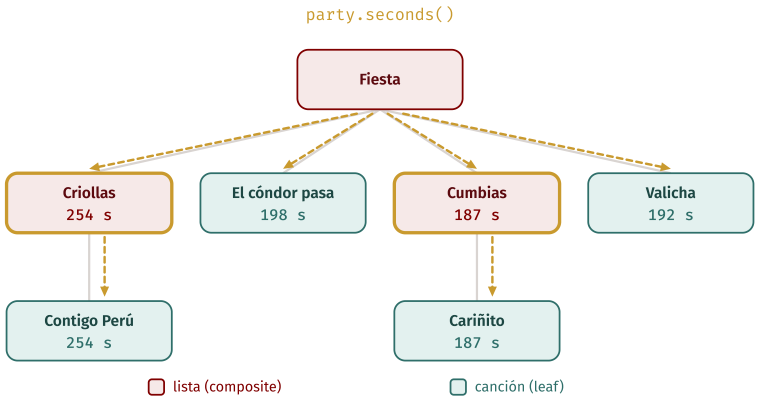
2. las listas hijas hacen lo mismo con los suyos

La duración sube por el árbol



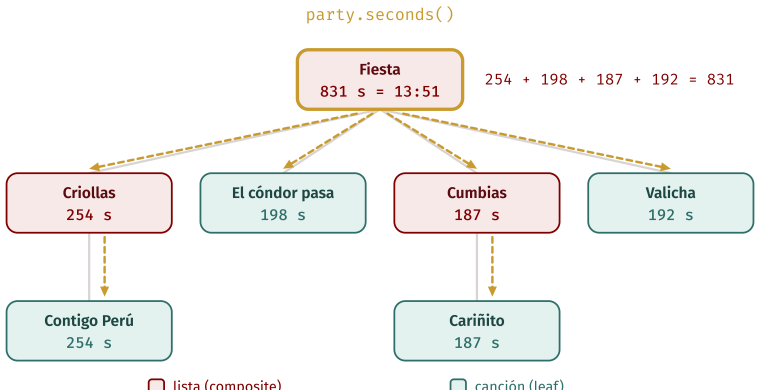
3. las leaves (canciones) responden su duración: el caso base

La duración sube por el árbol



4. cada lista suma lo que respondieron sus hijos

La duración sube por el árbol



5. la raíz suma y tiene el total, sin preguntar el tipo de nadie

Ejemplo: la lista “Fiesta”

CompositeDemo.java · fragmento

```java
var criollas = Playable.playlist(
        "Criollas", contigoPeru);
var cumbias = Playable.playlist(
        "Cumbias", carinito);
var party = Playable.playlist("Fiesta",
        criollas, condorPasa,
        cumbias, valicha);
IO.print(Playables.tree(party));
```

bash

```
$ java -cp build CompositeDemo
Fiesta  13:51
  Criollas  4:14
    Contigo Perú  4:14
  El cóndor pasa  3:18
  Cumbias  3:07
    Cariñito  3:07
  Valicha  3:12
```

`Playable.playlist`: un static factory method con varargs.

Operaciones nuevas con un `switch` sobre el tipo sealed

Playables.countSongs

```
return switch (p) {
  case Song s -> 1;
  case Playlist list -> {
    int n = 0;
    for (var e : list.entries()) {
      n += countSongs(e);
    }
    yield n;
  }
};
```

	Método en cada clase	**`switch` sobre el tipo sealed**
Tipo nuevo (un podcast)	Una clase nueva	Cambiar cada `switch`

Operaciones nuevas con un `switch` sobre el tipo sealed

Playables.countSongs

```
1  return switch (p) {
2    case Song s -> 1;
3    case Playlist list -> {
4      int n = 0;
5      for (var e : list.entries()) {
6        n += countSongs(e);
7      }
8      yield n;
9    }
10 };
```

	Método en cada clase	**`switch` sobre el tipo sealed**
Tipo nuevo (un podcast)	Una clase nueva	Cambiar cada `switch`
Operación nueva	Cambiar la interfaz y cada clase	Un método nuevo

Operaciones nuevas con un `switch` sobre el tipo sealed

Playables.countSongs

```
return switch (p) {
  case Song s -> 1;
  case Playlist list -> {
    int n = 0;
    for (var e : list.entries()) {
      n += countSongs(e);
    }
    yield n;
  }
};
```

	Método en cada clase	**`switch` sobre el tipo sealed**
Tipo nuevo (un podcast)	Una clase nueva	Cambiar cada `switch`
Operación nueva	Cambiar la interfaz y cada clase	Un método nuevo

La tensión de abierto/cerrado (semana 13). Lo que todo component sabe: un método. Lo que se agrega después: un `switch`.

Cuándo no usar Composite

- **Los datos no forman un árbol**: una lista de canciones sin listas adentro es un `List<Song>`.

Cuándo no usar Composite

- **Los datos no forman un árbol**: una lista de canciones sin listas adentro es un `List<Song>`.
- **La leaf y el composite no comparten operaciones con sentido**.

Cuándo no usar Composite

- **Los datos no forman un árbol**: una lista de canciones sin listas adentro es un `List<Song>`.
- **La leaf y el composite no comparten operaciones con sentido**.
- **Cada operación distingue los casos de todos modos**: un `switch` sobre el tipo sealed lo dice más claro.

Cuándo no usar Composite

- **Los datos no forman un árbol**: una lista de canciones sin listas adentro es un `List<Song>`.
- **La leaf y el composite no comparten operaciones con sentido**.
- **Cada operación distingue los casos de todos modos**: un `switch` sobre el tipo sealed lo dice más claro.

Error común

Poner `add(Playable)` en `Playable` "para tratar todo igual": una canción tendría un `add` que lanza una excepción. Es la violación de **Liskov** de la semana 7.

SECCIÓN 06

Patrones juntos

06



Tres patrones en un programa

CompositeDemo.java · fragmento

```java
var imported = new SongImporter().importAll(
  List.of("Contigo Perú;Arturo Cavero;254",
    "El cóndor pasa;Daniel Alomía Robles;198",
    "Cariñito;Los Hijos del Sol;187",
    "Valicha;Miguel Ángel Hurtado;192"));
// ... se arma el árbol party ...
List<Song> songs = Playables.songs(party);
songs.sort(Comparator.comparingInt(
        Song::seconds));
```

bash

```
3:07  Cariñito
3:12  Valicha
3:18  El cóndor pasa
4:14  Contigo Perú
```

Tres patrones en un programa

CompositeDemo.java · fragmento

```java
var imported = new SongImporter().importAll(
  List.of("Contigo Perú;Arturo Cavero;254",
    "El cóndor pasa;Daniel Alomía Robles;198",
    "Cariñito;Los Hijos del Sol;187",
    "Valicha;Miguel Ángel Hurtado;192"));
// ... se arma el árbol party ...
List<Song> songs = Playables.songs(party);
songs.sort(Comparator.comparingInt(
        Song::seconds));
```

bash

```
3:07  Cariñito
3:12  Valicha
3:18  El cóndor pasa
4:14  Contigo Perú
```

- **Factory Method**: `SongImporter`.

Tres patrones en un programa

CompositeDemo.java · fragmento

```java
var imported = new SongImporter().importAll(
  List.of("Contigo Perú;Arturo Cavero;254",
    "El cóndor pasa;Daniel Alomía Robles;198",
    "Cariñito;Los Hijos del Sol;187",
    "Valicha;Miguel Ángel Hurtado;192"));
// ... se arma el árbol party ...
List<Song> songs = Playables.songs(party);
songs.sort(Comparator.comparingInt(
        Song::seconds));
```

bash

```
3:07  Cariñito
3:12  Valicha
3:18  El cóndor pasa
4:14  Contigo Perú
```

- **Factory Method**: `SongImporter`.
- **Composite**: el árbol de listas.

Tres patrones en un programa

CompositeDemo.java · fragmento

```java
var imported = new SongImporter().importAll(
  List.of("Contigo Perú;Arturo Cavero;254",
    "El cóndor pasa;Daniel Alomía Robles;198",
    "Cariñito;Los Hijos del Sol;187",
    "Valicha;Miguel Ángel Hurtado;192"));
// ... se arma el árbol party ...
List<Song> songs = Playables.songs(party);
songs.sort(Comparator.comparingInt(
        Song::seconds));
```

bash

```
3:07  Cariñito
3:12  Valicha
3:18  El cóndor pasa
4:14  Contigo Perú
```

- **Factory Method**: `SongImporter`.
- **Composite**: el árbol de listas.
- **Strategy**: el `Comparator`.

SECCIÓN 07

Laboratorio

07

Laboratorio: patrones en el proyecto

1. Encontrar una **strategy sin nombre** en su proyecto (precios, descuentos, orden de reportes) y hacerla explícita con una interfaz y pruebas.

Laboratorio: patrones en el proyecto

1. Encontrar una **strategy sin nombre** en su proyecto (precios, descuentos, orden de reportes) y hacerla explícita con una interfaz y pruebas.
2. Un **observer** para alertas de stock bajo: el inventario notifica y una alerta registra; probarlo con un spy.

Laboratorio: patrones en el proyecto

1. Encontrar una **strategy sin nombre** en su proyecto (precios, descuentos, orden de reportes) y hacerla explícita con una interfaz y pruebas.
2. Un **observer** para alertas de stock bajo: el inventario notifica y una alerta registra; probarlo con un spy.
3. **Paquetes de productos** con Composite; probar el precio de un paquete anidado.

Laboratorio: patrones en el proyecto

1. Encontrar una **strategy sin nombre** en su proyecto (precios, descuentos, orden de reportes) y hacerla explícita con una interfaz y pruebas.
2. Un **observer** para alertas de stock bajo: el inventario notifica y una alerta registra; probarlo con un spy.
3. **Paquetes de productos** con Composite; probar el precio de un paquete anidado.
4. Una **factory** para crear productos desde texto, con pruebas listas para la semana 15.

Laboratorio: patrones en el proyecto

1. Encontrar una **strategy sin nombre** en su proyecto (precios, descuentos, orden de reportes) y hacerla explícita con una interfaz y pruebas.
2. Un **observer** para alertas de stock bajo: el inventario notifica y una alerta registra; probarlo con un spy.
3. **Paquetes de productos** con Composite; probar el precio de un paquete anidado.
4. Una **factory** para crear productos desde texto, con pruebas listas para la semana 15.

El código de los apuntes está en `code/week-14`: `make test` corre las cuatro demostraciones y 22 pruebas.

Resumen

- Un **patrón**: problema que se repite, solución con nombre, consecuencias. Vocabulario, no receta.
- **Strategy**: cada algoritmo detrás de una interfaz; se cambia mientras el programa corre. Puede ser una lambda. `Comparator` es una strategy.
- **Observer**: el subject avisa a una lista de observers que conoce por su interfaz. Decidir bajas, qué viaja, fallas y bajas durante el aviso. Se prueba con un spy.
- **Factories**: static factory method (nombre, caché, validación, subtipo), factory (un solo lugar), **Factory Method** (la subclase decide qué crear).
- **Composite**: leaf y composite con la misma interfaz; el composite le pregunta a sus hijos.
- Cuándo no: un solo algoritmo, un solo interesado, una sola clase o datos que no son un árbol.

Para la próxima clase

- Semana 15: archivos y E/S con `java.nio.file`, el patrón **Decorator**, CSV y serialización.
- Lecturas: Eck, secs. 11.1 y 11.2.1; dev.java, *The Java I/O API*.
- **P4** se entrega la semana 15; ese día se publica **P5**.

Pregunta para pensar

`LineImporter` lee de una `List<String>`. ¿Qué tendría que cambiar para leer de un archivo de verdad? ¿Y si el archivo no existe?