

UNIVERSIDAD NACIONAL DE INGENIERÍA

Programación Orientada a Objetos

Programación orientada a eventos

y cierre: el diseño de una aplicación completa

Gerson Garrido

Semana 16

Hoy

1. Programas secuenciales y programas que **esperan**
2. El event loop y los handlers: Observer otra vez
3. El modelo como **máquina de estados**
4. Separar el modelo de la interfaz (MVC) y probar sin interfaz
5. Interfaces gráficas con Swing (lectura)
6. Cierre: una aplicación por capas, las preguntas del principio y el repaso

> **Lectura**
>
> Eck, sec. 1.2. Edición Swing: subsecs. 6.1.3, 6.3.1, 6.4.4, 13.4.1 y 13.5.2. Relectura de 5.3.4.

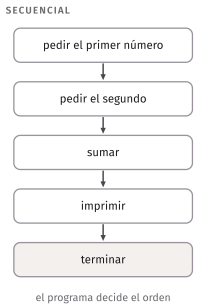
SECCIÓN 01

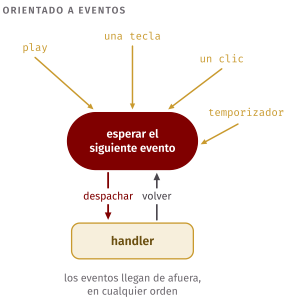
Eventos

01

Programas que esperan

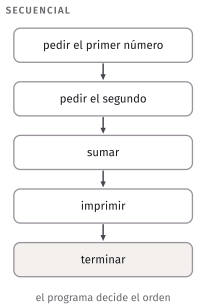
- **Secuencial**: el programa decide el orden.

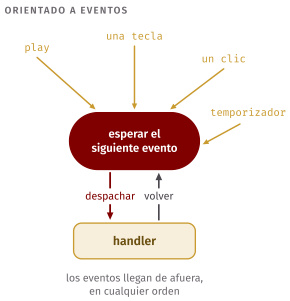




Programas que esperan

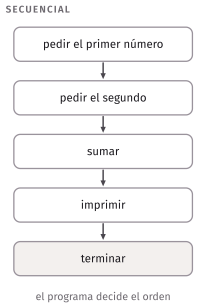
- **Secuencial**: el programa decide el orden.
- **Orientado a eventos**: el programa espera; lo que llega decide.

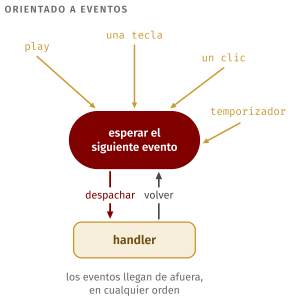




Programas que esperan

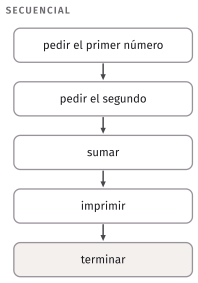
- **Secuencial**: el programa decide el orden.
- **Orientado a eventos**: el programa espera; lo que llega decide.
- **Evento**: una línea, un clic, una tecla, un temporizador.

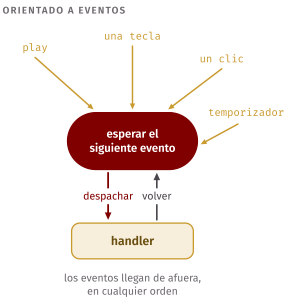




Programas que esperan

- **Secuencial**: el programa decide el orden.
- **Orientado a eventos**: el programa espera; lo que llega decide.
- **Evento**: una línea, un clic, una tecla, un temporizador.
- **Handler**: el código que reacciona a un tipo de evento.





La computadora ya funciona así

- Eck, sec. 1.2: la CPU no pregunta a cada rato si se presionó una tecla (*polling*).

La computadora ya funciona así

- Eck, sec. 1.2: la CPU no pregunta a cada rato si se presionó una tecla (*polling*).
- El hardware la interrumpe cuando pasa algo.

La computadora ya funciona así

- Eck, sec. 1.2: la CPU no pregunta a cada rato si se presionó una tecla (*polling*).
- El hardware la interrumpe cuando pasa algo.
- Los programas con ventanas repiten la idea un nivel más arriba: un event loop.

La computadora ya funciona así

- Eck, sec. 1.2: la CPU no pregunta a cada rato si se presionó una tecla (*polling*).
- El hardware la interrumpe cuando pasa algo.
- Los programas con ventanas repiten la idea un nivel más arriba: un event loop.
- Nuestro menú de la semana 15 ya era uno, pero con todo mezclado en un `switch`.

El event loop

EventLoop.java · run, resumido

```
while (running && in.hasNextLine()) {
  String line = in.nextLine().strip();
  List<String> words = ...;   // partir la línea
  var handler = handlers.get(words.get(0));
  if (handler == null) {
    out.println("comando desconocido: ...");
    continue;
  }
  try { handler.handle(words.subList(1, ...)); }
  catch (RuntimeException e) { out.println(..); }
}
```

- El despacho es una búsqueda en un mapa, no un `switch`.

El event loop

EventLoop.java · run, resumido

```java
while (running && in.hasNextLine()) {
  String line = in.nextLine().strip();
  List<String> words = ...;   // partir la línea
  var handler = handlers.get(words.get(0));
  if (handler == null) {
    out.println("comando desconocido: ...");
    continue;
  }
  try { handler.handle(words.subList(1, ...)); }
  catch (RuntimeException e) { out.println(..); }
}
```

- El despacho es una búsqueda en un mapa, no un `switch`.
- Comando nuevo: `loop.on(...)`, sin tocar `EventLoop`.

El event loop

EventLoop.java · run, resumido

```java
while (running && in.hasNextLine()) {
  String line = in.nextLine().strip();
  List<String> words = ...;   // partir la línea
  var handler = handlers.get(words.get(0));
  if (handler == null) {
    out.println("comando desconocido: ...");
    continue;
  }
  try { handler.handle(words.subList(1, ...)); }
  catch (RuntimeException e) { out.println(..); }
}
```

- El despacho es una búsqueda en un mapa, no un `switch`.
- Comando nuevo: `loop.on(...)`, sin tocar `EventLoop`.
- Un handler que falla no detiene el ciclo.

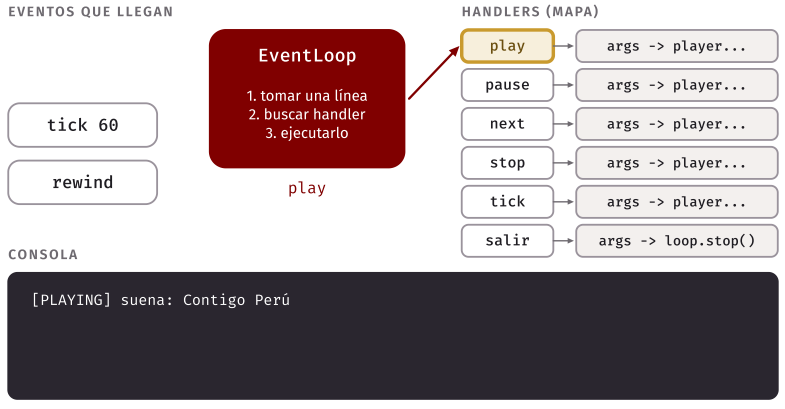
El event loop

EventLoop.java · run, resumido

```java
while (running && in.hasNextLine()) {
  String line = in.nextLine().strip();
  List<String> words = ...;   // partir la línea
  var handler = handlers.get(words.get(0));
  if (handler == null) {
    out.println("comando desconocido: ...");
    continue;
  }
  try { handler.handle(words.subList(1, ...)); }
  catch (RuntimeException e) { out.println(..); }
}
```

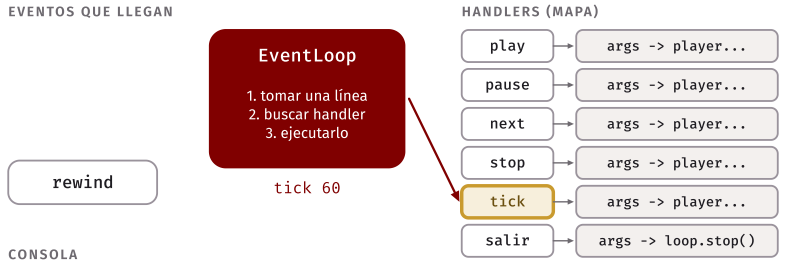
- El despacho es una búsqueda en un mapa, no un `switch`.
- Comando nuevo: `loop.on(...)`, sin tocar `EventLoop`.
- Un handler que falla no detiene el ciclo.
- Termina con el fin de la entrada o con `stop()`.

El event loop, paso a paso



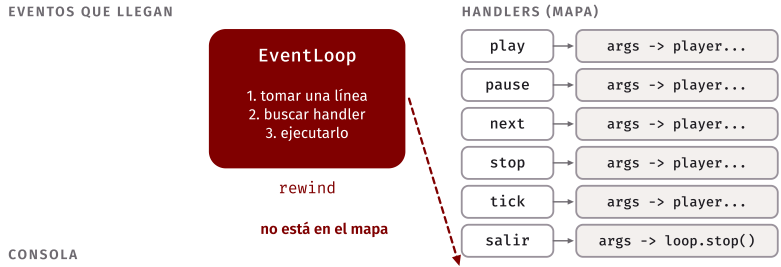
1. play: está en el mapa; su handler llama a player.play()

El event loop, paso a paso



2. tick 60: su handler recibe ["60"]; la canción sigue y no se imprime nada

El event loop, paso a paso



3. rewind: no hay handler; se informa y el ciclo sigue

El event loop, paso a paso



el ciclo vuelve a esperar: no sabe qué hace ningún comando

Handlers como objetos

tres formas del mismo handler

```java
// 1. Una clase con nombre (semana 7)
loop.on("play", new PlayHandler(player));

// 2. Una clase anónima
loop.on("play", new CommandHandler() {
    public void handle(List<String> args) {
        player.play();
    }
});
// 3. Una lambda (semana 12)
loop.on("play", args -> player.play());
```

- `CommandHandler`: una interfaz con un método.

Handlers como objetos

tres formas del mismo handler

```
1  // 1. Una clase con nombre (semana 7)
2  loop.on("play", new PlayHandler(player));
3
4  // 2. Una clase anónima
5  loop.on("play", new CommandHandler() {
6      public void handle(List<String> args) {
7          player.play();
8      }
9  });
10 // 3. Una lambda (semana 12)
11 loop.on("play", args -> player.play());
```

- `CommandHandler`: una interfaz con un método.
- Clases anónimas: en código anterior a Java 8 y en Eck.

Handlers como objetos

tres formas del mismo handler

```java
// 1. Una clase con nombre (semana 7)
loop.on("play", new PlayHandler(player));

// 2. Una clase anónima
loop.on("play", new CommandHandler() {
    public void handle(List<String> args) {
        player.play();
    }
});
// 3. Una lambda (semana 12)
loop.on("play", args -> player.play());
```

- `CommandHandler`: una interfaz con un método.
- Clases anónimas: en código anterior a Java 8 y en Eck.
- Hoy, lo habitual: una lambda.

Registrar un handler es registrarse como observer

Idea clave

El **subject** es quien produce los eventos: el ciclo, un botón, un temporizador. El **handler** es un observer con una interfaz de un método.

	Semana 14	Semana 16
Subject	`Thermometer`	`EventLoop`, un `JButton`

Registrar un handler es registrarse como observer

Idea clave

El **subject** es quien produce los eventos: el ciclo, un botón, un temporizador. El **handler** es un observer con una interfaz de un método.

	Semana 14	Semana 16
Subject	`Thermometer`	`EventLoop`, un `JButton`
Suscribirse	`addListener(l)`	`on("play", h)`, `addActionListener(h)`

Registrar un handler es registrarse como observer

Idea clave

El **subject** es quien produce los eventos: el ciclo, un botón, un temporizador. El **handler** es un observer con una interfaz de un método.

	Semana 14	Semana 16
Subject	`Thermometer`	`EventLoop`, un `JButton`
Suscribirse	`addListener(l)`	`on("play", h)`, `addActionListener(h)`
El aviso	`onReading(r)`	`handle(args)`, `actionPerformed(e)`

SECCIÓN 02

Máquinas de estados

02

El mismo evento, distintas reacciones

Estado	`play`	`pause`	`tick`
`STOPPED`	→ `PLAYING`, desde la primera	nada	se ignora
`PLAYING`	nada (“ya está sonando”)	→ `PAUSED`	cuenta segundos
`PAUSED`	→ `PLAYING`, continúa	nada	se ignora

El mismo evento, distintas reacciones

Estado	`play`	`pause`	`tick`
`STOPPED`	→ `PLAYING`, desde la primera	nada	se ignora
`PLAYING`	nada (“ya está sonando”)	→ `PAUSED`	cuenta segundos
`PAUSED`	→ `PLAYING`, continúa	nada	se ignora

- Un reproductor de música: tres **estados**, un `enum`.

El mismo evento, distintas reacciones

Estado	`play`	`pause`	`tick`
`STOPPED`	→ `PLAYING`, desde la primera	nada	se ignora
`PLAYING`	nada (“ya está sonando”)	→ `PAUSED`	cuenta segundos
`PAUSED`	→ `PLAYING`, continúa	nada	se ignora

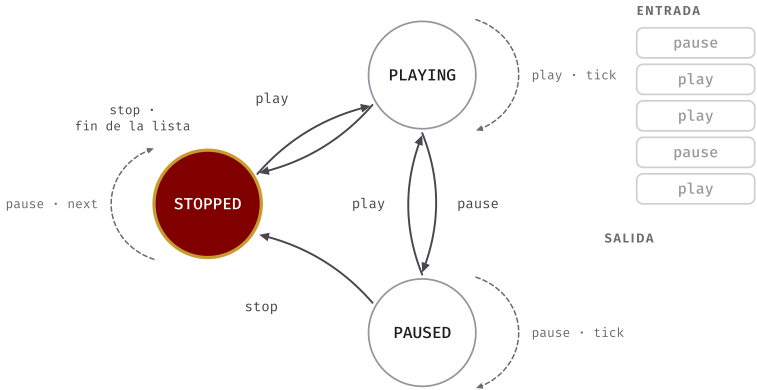
- Un reproductor de música: tres **estados**, un `enum`.
- Una **máquina de estados**: para cada par (estado, evento), a qué estado se pasa y qué se hace.

El mismo evento, distintas reacciones

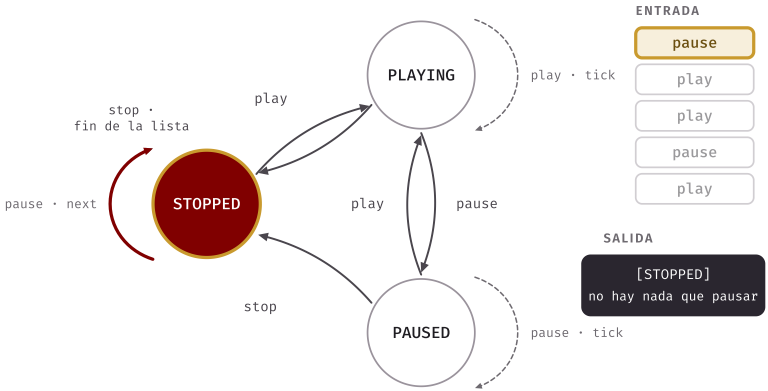
Estado	`play`	`pause`	`tick`
`STOPPED`	→ `PLAYING`, desde la primera	nada	se ignora
`PLAYING`	nada (“ya está sonando”)	→ `PAUSED`	cuenta segundos
`PAUSED`	→ `PLAYING`, continúa	nada	se ignora

- Un reproductor de música: tres **estados**, un `enum`.
- Una **máquina de estados**: para cada par (estado, evento), a qué estado se pasa y qué se hace.
- Además `next` y `stop`. El reloj también es una fuente de eventos: `tick`.

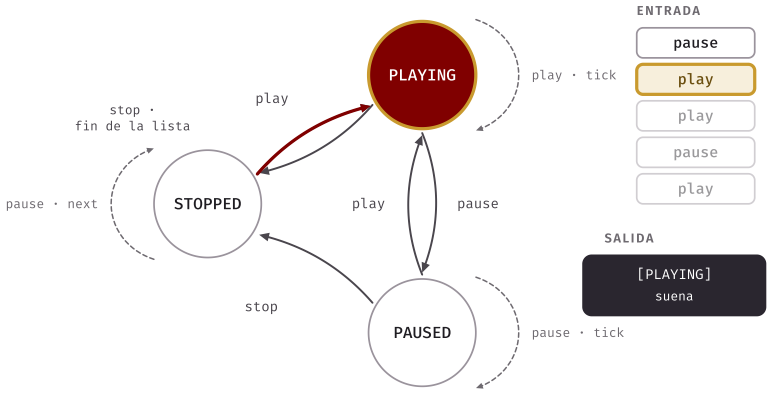
La máquina, recorriendo la entrada



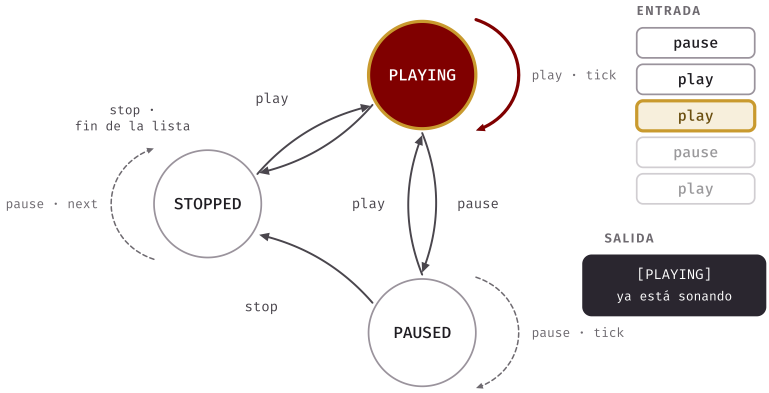
La máquina, recorriendo la entrada



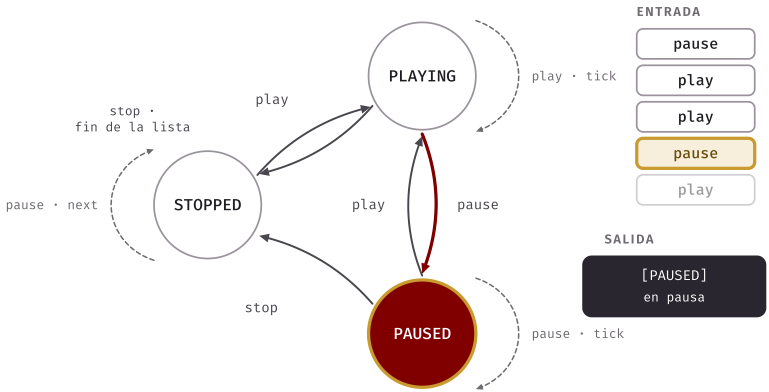
La máquina, recorriendo la entrada



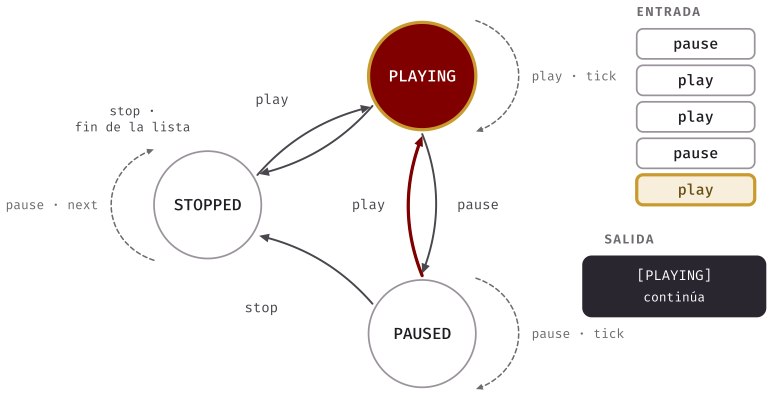
La máquina, recorriendo la entrada



La máquina, recorriendo la entrada



La máquina, recorriendo la entrada



el mismo evento, play, hace tres cosas distintas según el estado

En Java: un `enum` y un `switch` por evento

Player.java

```java
public void play() {
  String message = switch (state) {
    case STOPPED -> {
      index = 0;
      elapsed = 0;
      yield "suena";
    }
    case PAUSED -> "continúa";
    case PLAYING -> "ya está sonando";
  };
  change(State.PLAYING, message);
}
```

- `change` actualiza el estado y avisa a los observers con un `PlayerEvent`.

En Java: un `enum` y un `switch` por evento

Player.java

```java
public void play() {
  String message = switch (state) {
    case STOPPED -> {
      index = 0;
      elapsed = 0;
      yield "suena";
    }
    case PAUSED -> "continúa";
    case PLAYING -> "ya está sonando";
  };
  change(State.PLAYING, message);
}
```

- `change` actualiza el estado y avisa a los observers con un `PlayerEvent`.
- El reproductor es un subject, como el termómetro.

En Java: un `enum` y un `switch` por evento

Player.java

```java
public void play() {
  String message = switch (state) {
    case STOPPED -> {
      index = 0;
      elapsed = 0;
      yield "suena";
    }
    case PAUSED -> "continúa";
    case PLAYING -> "ya está sonando";
  };
  change(State.PLAYING, message);
}
```

- `change` actualiza el estado y avisa a los observers con un `PlayerEvent`.
- El reproductor es un subject, como el termómetro.
- El cambio de canción puede venir del usuario (`next`) o del tiempo (`tick`): el modelo no distingue.

Error común: el `switch` que no se revisa

Player.java · pause

```
1  public void pause() {
2    switch (state) {
3      case PLAYING -> change(PAUSED, ..)
         ;
4      case PAUSED, STOPPED -> change(..)
         ;
5    }
6  }
```

Error común: el `switch` que no se revisa

Player.java · pause

```java
public void pause() {
  switch (state) {
    case PLAYING -> change(PAUSED, ..)
     ;
    case PAUSED, STOPPED -> change(..)
     ;
  }
}
```

Es una **sentencia**; `play` usa una **expresión**.
¿Qué pasa si se agrega `BUFFERING` a `State`?

Error común: el `switch` que no se revisa

Player.java · pause

```java
public void pause() {
  switch (state) {
    case PLAYING -> change(PAUSED, ..)
     ;
    case PAUSED, STOPPED -> change(..)
     ;
  }
}
```

Es una **sentencia**; `play` usa una **expresión**.
¿Qué pasa si se agrega `BUFFERING` a `State`?

Error común

Una sentencia `switch` sobre un `enum` puede dejar casos sin cubrir (compatibilidad con el código anterior a Java 21). Solo la expresión se revisa.

Con `BUFFERING` en `State`

bash

```
Player.java:36: error: the switch expression does not cover all possible input
    values
        String message = switch (state) {
                         ^
1 error
```

Con `BUFFERING` en `State`

```bash
Player.java:36: error: the switch expression does not cover all possible input
    values
        String message = switch (state) {
                         ^
1 error
```

- Solo falla `play`. `pause` compila, y en `BUFFERING` no hace nada en silencio.

Con `BUFFERING` en `State`

```
bash

Player.java:36: error: the switch expression does not cover all possible input
    values
        String message = switch (state) {
                         ^
1 error
```

- Solo falla `play`. `pause` compila, y en `BUFFERING` no hace nada en silencio.
- En una máquina de estados: escribir el `switch` como expresión, o con un `default` que lance una excepción.

SECCIÓN 03

Modelo e interfaz

03

Modelo, vista y controlador

PlayerApp.java · build, resumido

```
1  player.addListener(new ConsoleView(out));
2  var loop = new EventLoop(out);
3  loop.on("play", args -> player.play());
4  loop.on("pause", args -> player.pause());
5  loop.on("next", args -> player.next());
6  loop.on("stop", args -> player.stop());
7  loop.on("tick", args -> ...);
8  loop.on("salir", args -> loop.stop());
```

- **Modelo**: `Player`. Estado y reglas; no sabe que hay una consola.

Modelo, vista y controlador

PlayerApp.java · build, resumido

```
player.addListener(new ConsoleView(out));
var loop = new EventLoop(out);
loop.on("play", args -> player.play());
loop.on("pause", args -> player.pause());
loop.on("next", args -> player.next());
loop.on("stop", args -> player.stop());
loop.on("tick", args -> ...);
loop.on("salir", args -> loop.stop());
```

- **Modelo**: `Player`. Estado y reglas; no sabe que hay una consola.
- **Vista**: `ConsoleView`. Observa al modelo.

Modelo, vista y controlador

PlayerApp.java · build, resumido

```java
player.addListener(new ConsoleView(out));
var loop = new EventLoop(out);
loop.on("play", args -> player.play());
loop.on("pause", args -> player.pause());
loop.on("next", args -> player.next());
loop.on("stop", args -> player.stop());
loop.on("tick", args -> ...);
loop.on("salir", args -> loop.stop());
```

- **Modelo**: `Player`. Estado y reglas; no sabe que hay una consola.
- **Vista**: `ConsoleView`. Observa al modelo.
- **Controlador**: el event loop y sus handlers. Eventos → llamadas al modelo.

Modelo, vista y controlador

PlayerApp.java · build, resumido

```java
player.addListener(new ConsoleView(out));
var loop = new EventLoop(out);
loop.on("play", args -> player.play());
loop.on("pause", args -> player.pause());
loop.on("next", args -> player.next());
loop.on("stop", args -> player.stop());
loop.on("tick", args -> ...);
loop.on("salir", args -> loop.stop());
```

- **Modelo**: `Player`. Estado y reglas; no sabe que hay una consola.
- **Vista**: `ConsoleView`. Observa al modelo.
- **Controlador**: el event loop y sus handlers. Eventos → llamadas al modelo.
- `PlayerApp`: el único que conoce todas las piezas.

Una sesión con el reproductor

bash · java -cp build PlayerApp < player-input.txt

```
[STOPPED] no hay nada que pausar: Contigo Perú
[PLAYING] suena: Contigo Perú
[PLAYING] ya está sonando: Contigo Perú
[PAUSED] en pausa: Contigo Perú
[PLAYING] continúa: Contigo Perú
[PLAYING] siguiente: Cariñito
[PLAYING] siguiente: Valicha
comando desconocido: rewind; comandos: [play,
    pause, next, stop, tick, salir]
error: uso: tick <segundos>
[STOPPED] fin de la lista: Valicha
```

Entrada: `pause`, `play`, `play`, `tick 200`, `pause`, `tick 100`, `play`, `tick 60`, `next`, `rewind`, `tick`, `next`, `salir`.

Una sesión con el reproductor

bash · java -cp build PlayerApp < player-input.txt

```
[STOPPED] no hay nada que pausar: Contigo Perú
[PLAYING] suena: Contigo Perú
[PLAYING] ya está sonando: Contigo Perú
[PAUSED] en pausa: Contigo Perú
[PLAYING] continúa: Contigo Perú
[PLAYING] siguiente: Cariñito
[PLAYING] siguiente: Valicha
comando desconocido: rewind; comandos: [play,
    pause, next, stop, tick, salir]
error: uso: tick <segundos>
[STOPPED] fin de la lista: Valicha
```

Entrada: `pause`, `play`, `play`, `tick 200`, `pause`, `tick 100`, `play`, `tick 60`, `next`, `rewind`, `tick`, `next`, `salir`.

- `tick 200`: la canción dura 254 s; no pasa nada visible.

Una sesión con el reproductor

bash · java -cp build PlayerApp < player-input.txt

```
[STOPPED] no hay nada que pausar: Contigo Perú
[PLAYING] suena: Contigo Perú
[PLAYING] ya está sonando: Contigo Perú
[PAUSED] en pausa: Contigo Perú
[PLAYING] continúa: Contigo Perú
[PLAYING] siguiente: Cariñito
[PLAYING] siguiente: Valicha
comando desconocido: rewind; comandos: [play,
    pause, next, stop, tick, salir]
error: uso: tick <segundos>
[STOPPED] fin de la lista: Valicha
```

Entrada: `pause`, `play`, `play`, `tick 200`, `pause`, `tick 100`, `play`, `tick 60`, `next`, `rewind`, `tick`, `next`, `salir`.

- `tick 200`: la canción dura 254 s; no pasa nada visible.
- `tick 100` en pausa: se ignora.

Una sesión con el reproductor

bash · java -cp build PlayerApp < player-input.txt

```
[STOPPED] no hay nada que pausar: Contigo Perú
[PLAYING] suena: Contigo Perú
[PLAYING] ya está sonando: Contigo Perú
[PAUSED] en pausa: Contigo Perú
[PLAYING] continúa: Contigo Perú
[PLAYING] siguiente: Cariñito
[PLAYING] siguiente: Valicha
comando desconocido: rewind; comandos: [play,
    pause, next, stop, tick, salir]
error: uso: tick <segundos>
[STOPPED] fin de la lista: Valicha
```

Entrada: `pause`, `play`, `play`, `tick 200`, `pause`, `tick 100`, `play`, `tick 60`, `next`, `rewind`, `tick`, `next`, `salir`.

- `tick 200`: la canción dura 254 s; no pasa nada visible.
- `tick 100` en pausa: se ignora.
- $200 + 60 = 260$: la canción termina sola.

Error común: lógica en el handler

Error común

Que la lambda de `play` decida si empezar o continuar, o que el `case "vender"` de un menú calcule el total y descuente el stock. No se prueba sin simular la interfaz, y hay que reescribirlo cuando la interfaz cambia.

Error común: lógica en el handler

Error común

Que la lambda de `play` decida si empezar o continuar, o que el `case "vender"` de un menú calcule el total y descuente el stock. No se prueba sin simular la interfaz, y hay que reescribirlo cuando la interfaz cambia.

El handler traduce un evento en una llamada al modelo. Las decisiones están en el modelo.

Probar sin interfaz

PlayerTest.java · fragmento

```
1  player = new Player(List.of(a, b));
2  player.addListener(events::add);   // un spy
3  @Test
4  void playMeansDifferentThingsInEachState() {
5      player.play();
6      assertEquals("suena", lastMessage());
7      player.play();
8      assertEquals("ya está sonando",
9              lastMessage());
10 }
```

- El spy: una lista y una referencia a método.

Probar sin interfaz

PlayerTest.java · fragmento

```
1  player = new Player(List.of(a, b));
2  player.addListener(events::add);   // un spy
3  @Test
4  void playMeansDifferentThingsInEachState() {
5      player.play();
6      assertEquals("suena", lastMessage());
7      player.play();
8      assertEquals("ya está sonando",
9              lastMessage());
10 }
```

- El spy: una lista y una referencia a método.
- `EventLoopTest`: el event loop con handlers de prueba, sin reproductor.

Probar sin interfaz

PlayerTest.java · fragmento

```java
player = new Player(List.of(a, b));
player.addListener(events::add);   // un spy
@Test
void playMeansDifferentThingsInEachState() {
    player.play();
    assertEquals("suena", lastMessage());
    player.play();
    assertEquals("ya está sonando",
            lastMessage());
}
```

- El spy: una lista y una referencia a método.
- `EventLoopTest`: el event loop con handlers de prueba, sin reproductor.
- `PlayerAppTest`: toda la aplicación, con texto de entrada y de salida.

SECCIÓN 04

Interfaces gráficas (lectura)

04

Lectura: el mismo reproductor, con una ventana

SwingPlayer.java · fragmento, solo lectura

```
SwingUtilities.invokeLater(() -> {
  JLabel status = new JLabel("detenido");
  JButton play = new JButton("Play");
  play.addActionListener(e -> player.play());
  player.addListener(ev -> status.setText(
      ev.message() + ": " + ev.song().title()));
  JPanel panel = new JPanel();
  panel.add(play);
  panel.add(status);
  // ... un JFrame que muestra el panel
});
```

- Swing viene con el JDK.

Lectura: el mismo reproductor, con una ventana

SwingPlayer.java · fragmento, solo lectura

```java
SwingUtilities.invokeLater(() -> {
  JLabel status = new JLabel("detenido");
  JButton play = new JButton("Play");
  play.addActionListener(e -> player.play());
  player.addListener(ev -> status.setText(
      ev.message() + ": " + ev.song().title()));
  JPanel panel = new JPanel();
  panel.add(play);
  panel.add(status);
  // ... un JFrame que muestra el panel
});
```

- Swing viene con el JDK.
- `JButton`: un subject. `addActionListener`: suscribirse.

Lectura: el mismo reproductor, con una ventana

SwingPlayer.java · fragmento, solo lectura

```java
SwingUtilities.invokeLater(() -> {
  JLabel status = new JLabel("detenido");
  JButton play = new JButton("Play");
  play.addActionListener(e -> player.play());
  player.addListener(ev -> status.setText(
      ev.message() + ": " + ev.song().title()));
  JPanel panel = new JPanel();
  panel.add(play);
  panel.add(status);
  // ... un JFrame que muestra el panel
});
```

- Swing viene con el JDK.
- `JButton`: un subject. `addActionListener`: suscribirse.
- `Player` es la misma clase, con las mismas pruebas.

Lectura: el mismo reproductor, con una ventana

SwingPlayer.java · fragmento, solo lectura

```java
SwingUtilities.invokeLater(() -> {
  JLabel status = new JLabel("detenido");
  JButton play = new JButton("Play");
  play.addActionListener(e -> player.play());
  player.addListener(ev -> status.setText(
      ev.message() + ": " + ev.song().title()));
  JPanel panel = new JPanel();
  panel.add(play);
  panel.add(status);
  // ... un JFrame que muestra el panel
});
```

- Swing viene con el JDK.
- `JButton`: un subject. `addActionListener`: suscribirse.
- `Player` es la misma clase, con las mismas pruebas.
- `invokeLater`: el código corre en el hilo del event loop de Swing.

Lectura: una ventana necesita una pantalla

bash · en el contenedor del curso

```
Exception in thread "AWT-EventQueue-0" java.awt.HeadlessException:
No X11 DISPLAY variable was set,
or no headful library support was found,
...
```

- El contenedor no tiene pantalla: el código compila pero no corre.

Lectura: una ventana necesita una pantalla

bash · en el contenedor del curso

```
Exception in thread "AWT-EventQueue-0" java.awt.HeadlessException:
No X11 DISPLAY variable was set,
or no headful library support was found,
...
```

- El contenedor no tiene pantalla: el código compila pero no corre.
- Para probarlo: copiarlo con las clases del reproductor a una computadora con JDK y pantalla.

Lectura: una ventana necesita una pantalla

bash · en el contenedor del curso

```
Exception in thread "AWT-EventQueue-0" java.awt.HeadlessException:
No X11 DISPLAY variable was set,
or no headful library support was found,
...
```

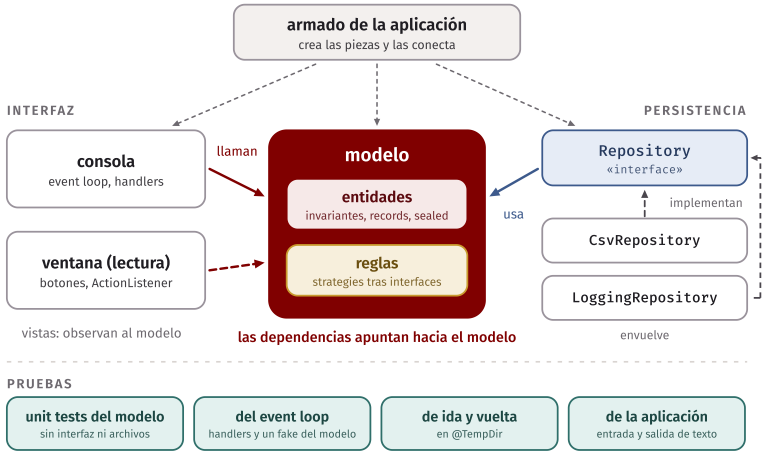
- El contenedor no tiene pantalla: el código compila pero no corre.
- Para probarlo: copiarlo con las clases del reproductor a una computadora con JDK y pantalla.
- Lectura: Eck, edición Swing, cap. 6 (6.1 a 6.5) y 13.4.1 (MVC en Swing). JavaFX se instala aparte.

SECCIÓN 05

Cierre

05

Una aplicación por capas



Cada capa, con lo que vimos en el curso

Capa	Qué hay ahí	Semanas
Modelo	Entidades con invariantes, valores inmutables	2–8: encapsulamiento, `record`, herencia, `sealed`
Reglas	Precios, descuentos, órdenes: strategies	7, 12, 14: interfaces, lambdas, Strategy
Operaciones	Ventas, reportes, consultas	9–12: excepciones, genéricos, colecciones, streams
Persistencia	Repositorio tras una interfaz; CSV	15
Interfaz	Event loop, handlers, vistas	15, 16
Armado	Una clase que conecta por constructores	13: inversión de dependencias
Pruebas	Por capa, de ida y vuelta, de la aplicación	desde la 3

Cada capa, con lo que vimos en el curso

Capa	Qué hay ahí	Semanas
Modelo	Entidades con invariantes, valores inmutables	2–8: encapsulamiento, `record`, herencia, `sealed`
Reglas	Precios, descuentos, órdenes: strategies	7, 12, 14: interfaces, lambdas, Strategy
Operaciones	Ventas, reportes, consultas	9–12: excepciones, genéricos, colecciones, streams
Persistencia	Repositorio tras una interfaz; CSV	15
Interfaz	Event loop, handlers, vistas	15, 16
Armado	Una clase que conecta por constructores	13: inversión de dependencias
Pruebas	Por capa, de ida y vuelta, de la aplicación	desde la 3

Para P5: ¿en qué clase está cada fila? ¿Alguna clase está en dos filas?

Las preguntas del principio

- **¿Qué es un objeto?** Estado + comportamiento + identidad. Los demás envían mensajes.

Las preguntas del principio

- **¿Qué es un objeto?** Estado + comportamiento + identidad. Los demás envían mensajes.
- **¿Quién protege cada invariante?** La clase que tiene el dato: campos privados, validación en el constructor. Un record, incluso al deserializarse.

Las preguntas del principio

- **¿Qué es un objeto?** Estado + comportamiento + identidad. Los demás envían mensajes.
- **¿Quién protege cada invariante?** La clase que tiene el dato: campos privados, validación en el constructor. Un record, incluso al deserializarse.
- **¿Polimorfismo o tipo sealed?** Tipos nuevos frecuentes: polimorfismo. Tipos fijos y operaciones nuevas: `sealed` con `switch`.

Las preguntas del principio

- **¿Qué es un objeto?** Estado + comportamiento + identidad. Los demás envían mensajes.
- **¿Quién protege cada invariante?** La clase que tiene el dato: campos privados, validación en el constructor. Un record, incluso al deserializarse.
- **¿Polimorfismo o tipo sealed?** Tipos nuevos frecuentes: polimorfismo. Tipos fijos y operaciones nuevas: `sealed` con `switch`.
- **¿Qué depende de qué?** Lo que cambia seguido depende de lo estable, por constructores e interfaces.

Las preguntas del principio

- **¿Qué es un objeto?** Estado + comportamiento + identidad. Los demás envían mensajes.
- **¿Quién protege cada invariante?** La clase que tiene el dato: campos privados, validación en el constructor. Un record, incluso al deserializarse.
- **¿Polimorfismo o tipo sealed?** Tipos nuevos frecuentes: polimorfismo. Tipos fijos y operaciones nuevas: `sealed` con `switch`.
- **¿Qué depende de qué?** Lo que cambia seguido depende de lo estable, por constructores e interfaces.
- **¿Cómo sabemos que funciona?** Porque hay pruebas, y porque con ellas podemos cambiar el diseño sin miedo.

Lo que no vimos

- **Concurrencia**: varios hilos a la vez (Eck, cap. 12).

Lo que no vimos

- **Concurrencia**: varios hilos a la vez (Eck, cap. 12).
- **Interfaces gráficas** a fondo (Eck, caps. 6 y 13).

Lo que no vimos

- **Concurrencia**: varios hilos a la vez (Eck, cap. 12).
- **Interfaces gráficas** a fondo (Eck, caps. 6 y 13).
- **Bases de datos**: consultas, concurrencia, transacciones.

Lo que no vimos

- **Concurrencia**: varios hilos a la vez (Eck, cap. 12).
- **Interfaces gráficas** a fondo (Eck, caps. 6 y 13).
- **Bases de datos**: consultas, concurrencia, transacciones.
- **Maven, Gradle**: build y dependencias, en lugar de nuestro `Makefile`.

Lo que no vimos

- **Concurrencia**: varios hilos a la vez (Eck, cap. 12).
- **Interfaces gráficas** a fondo (Eck, caps. 6 y 13).
- **Bases de datos**: consultas, concurrencia, transacciones.
- **Maven, Gradle**: build y dependencias, en lugar de nuestro `Makefile`.
- **Más patrones**: el libro de 1994 tiene 23; vimos Strategy, Observer, Factory Method, Composite, Decorator y, sin nombrarlo, Template Method en `importAll`.

Repaso para el examen final (semanas 9 a 16)

9	¿Checked o unchecked? ¿Dónde se captura? ¿Qué garantiza `try`-with-resources?
10	¿Qué es el type erasure? ¿`? extends T` o `? super T`?
11	¿Qué colección para cada problema? ¿Qué contrato une `equals`, `hashCode` y `compareTo`?
12	¿Qué interfaz funcional corresponde a una lambda? ¿Qué hace cada paso de un stream?
13	¿Qué principio viola este código? ¿Cuál es el siguiente paso pequeño de refactoring?
14	¿Qué problema resuelve cada patrón? ¿Cuándo sería un exceso?
15	¿Dónde va el código que lee archivos? ¿Por qué no deserializar datos de fuera?
16	¿Qué hace este evento en este estado? ¿Qué cambia si cambia la interfaz?

SECCIÓN 06

Laboratorio

06

Laboratorio: presentación de proyectos

1. Cada equipo presenta P5 en **10 minutos**: una demostración por línea de comandos, el diagrama de clases final y una decisión de diseño que cambiaría.

Laboratorio: presentación de proyectos

1. Cada equipo presenta P5 en **10 minutos**: una demostración por línea de comandos, el diagrama de clases final y una decisión de diseño que cambiaría.
2. Preguntas sobre el código: dónde está un invariante, qué prueba lo cubre, cómo se agregaría un tipo de producto nuevo.

Laboratorio: presentación de proyectos

1. Cada equipo presenta P5 en **10 minutos**: una demostración por línea de comandos, el diagrama de clases final y una decisión de diseño que cambiaría.
2. Preguntas sobre el código: dónde está un invariante, qué prueba lo cubre, cómo se agregaría un tipo de producto nuevo.
3. Revisión cruzada: leer el `README.md` y las pruebas de otro equipo; anotar un code smell y una fortaleza.

Laboratorio: presentación de proyectos

1. Cada equipo presenta P5 en **10 minutos**: una demostración por línea de comandos, el diagrama de clases final y una decisión de diseño que cambiaría.
2. Preguntas sobre el código: dónde está un invariante, qué prueba lo cubre, cómo se agregaría un tipo de producto nuevo.
3. Revisión cruzada: leer el `README.md` y las pruebas de otro equipo; anotar un code smell y una fortaleza.

`code/week-16`: `make test` corre el reproductor con `player-input.txt` y 12 pruebas.

Resumen

- Orientado a eventos: el orden lo deciden los eventos; el programa decide cómo reaccionar.
- Un **event loop** busca el **handler** de cada evento y lo ejecuta. Registrar un handler es registrarse como **observer**.
- Un modelo que depende de su estado es una **máquina de estados**: `enum` y una **expresión** `switch`, que el compilador revisa.
- **MVC**: el modelo no sabe de la interfaz; la vista observa; el controlador traduce eventos. La lógica va en el modelo, que se prueba sin interfaz.
- (Lectura) En Swing, un `JButton` es un subject; el modelo es el mismo.
- Una aplicación completa: capas con las dependencias hacia el modelo, y pruebas en cada una.

Para el examen final

- Repasar los apuntes y los ejercicios de las semanas 9 a 16, y las preguntas de repaso de cada guía.
- Volver a correr `make test` en el código de cada semana y cambiar algo: ¿qué prueba falla?
- **P5** se entrega esta semana y se presenta en el laboratorio.

Pregunta para pensar

Si mañana la bodega necesitara una interfaz gráfica, ¿qué clases de su proyecto cambiarían y cuáles no?