

TEMA 3. PARADIGMA ORIENTADO A EVENTOS

- PROGRAMACIÓN SECUENCIAL:

- **FLUJO DE INFORMACIÓN:** Formado por la secuencia de sentencias que componen el programa

- **ESTADO DEL PROGRAMA:** Definido por el punto del programa en que se encuentra la ejecución (VALOR DE LOS DATOS)

- PROGRAMACIÓN ORIENTADA A EVENTOS: Guiada por sucesos que ocurren y que, según cual suceda, ejecutan un código u otro

* **NO EXISTE UN ÚNICO FLUJO DE EXECUCIÓN (Ejemplo: las GUI)**

- ESTADO EN UN POE: Más difícil de determinar que en un programa secuencial

→ Habitualmente programas concurrentes (MULTITILO)

→ Distintos módulos que se ejecutan a distintas velocidades

→ Los eventos pueden suceder en cualquier momento y suelen depender de la interacción del usuario

→ Los módulos se ejecutan simultáneamente

↳ Algunos nunca se ejecutarán

↳ Otros pueden tener varias instancias en ejecución

→ Los módulos pueden compartir info (DATOS)

- EVENTOS: Ocurrencias observables

↳ Un observador percibe que sucede
↳ Sucede en algún momento

- TIPOS:

1. Externos: Acciones del USUARIO

2. Internos: Vencimiento temporizador ó datos en líneas de comunic.

- RESPUESTA A EVENTOS:

→ Si el observador se interesa por un evento, puede responder a este de alguna forma (Se dice que maneja o controla el evento)

- FUENTES DE EVENTOS:

→ Algunos eventos se dan porque los provoca algún agente (FUENTE DEL EVENTO)

↳ CUANDO DE ALGÚN EVENTO NO NOS INTERESA SU ORIGEN, SE DICE QUE EL EVENTO 'SE DISPARA', AUNQUE POR SI MISMOS NO SON AGENTES CAPACES DE ACTUAR

DISPARA O Lanza EL EVENTO ←

- **SISTEMA**: Conjunto de agentes sujetos a un conjunto definido de comportamiento e interacciones

+ **COMPORTAMIENTO EN CUALQUIER MOMENTO DEPENDE DEL ESTADO**

+ **TIPOS DE INTERACCIÓN BASADOS EN EVENTOS**:

- Petición - Respuesta: Se encuentra entre 2 agentes



- Paso de mensaje: También entre 2 ejemplos, pero en este caso el agente que recibe el MENSAJE no está obligado a responder

- Publicación - Subscripción: Implica múltiples agentes, los Bi a Bm se suscriben a un servicio de mensajes indicando que desean recibir los de cierto tipo.

los agentes Ai a An publican varios tipos de mensajes para el servicio.

Si Ai publica un mensaje al que Bj está interesado, este recibe una copia



- **ESTADO DEL SISTEMA**: Descripción completa del sistema en un momento dado

+ **SE MODIFICA A TRAVÉS DE LOS AGENTES DEL SISTEMA QUE INTERACTUAN POR EVENTOS**

- **SISTEMA BASADO EN EVENTOS**: las interacciones entre agentes se rigen por eventos

- **SISTEMAS DISCRETOS Y EVENTOS**:

→ Un sistema es discreto si cada estado del sistema se puede definir por una cantidad finita de memoria y si, en cualquier periodo finito de tiempo, el sistema sufre un nº finito de cambios

+ **PARA CADA ESTADO, SIEMPRE HAY UNO PROXIMO**

- El PARADIGMA DE POE :

- Es una forma de pensar a cerca de los problemas y sus soluciones
- Proporciona abstracciones ⇒ **MODELO DE EVENTO**

- MODELO DE EVENTO: El concepto de evento es el núcleo del POE
- EXISTEN 3 TIPOS DE OBJETOS COMPUTACIONALES ASOCIADOS A CADA EVENTO:

→ Fuentes de eventos:

- * Es el creador del evento
- * Activa un evento cuando crea un **OBJETO DE EVENTO** y lo prepara para los **MANEJADORES**

→ Objetos de eventos:

- * Encapsula los datos críticos asociados con el evento

→ Manejadores de eventos (o controladores):

- * Los controladores de eventos responden a estos llevando a cabo las acciones especificadas por el programador

- EVENTOS VS INVOCACIONES DE MÉTODO: Aunque la idea de provocar la ejecución de cierto código es parecida, existen diferencias notables.

1. Fuente del evento y manejador mucho más débilmente acoplados que los objetos en una relación estandar de LLAMADOR Y LLAMADO

2. Controladores registrados con las fuentes de eventos en tiempo de ejecución.

↳ ES POSIBLE CONECTAR DIFERENTES CONTROLADORES CON UNA MISMA FUENTE EN DIFERENTES MOMENTOS

3. Puede haber 0, 1 o varios controladores para un evento

Estos puntos varían según la implementación del lenguaje que utilizamos

↳ MULTIDIFUSIÓN: Util cuando hay varias vistas que dependen de los mismos datos

4. Los controladores de eventos no devuelven ninguna información al origen del evento (son de tipo void)

5. Multiples fuentes de eventos pueden disparar eventos al mismo controlador

6. En la mayoría de lenguajes, el origen del evento no se bloquea esperando a que los manejadores se completen.

↳ Dispara el evento, luego continua funcionando: fuente y controlador trabajan de forma asíncrona

7. Puede haber un retraso entre el momento en que se desata el evento y cuando cada controlador lo procesa

- DISEÑO DE POEs:

- Respuestas ante eventos:

→ PROGRAMAS DE ALGORITMO DE RESPUESTA UNICO:

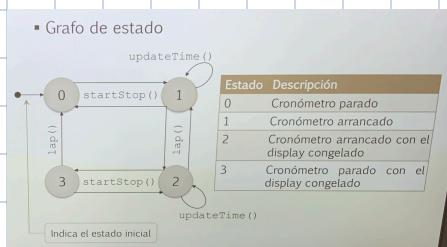
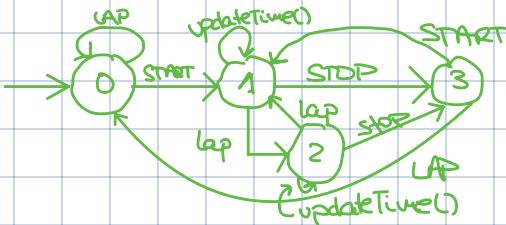
- Respuesta ante un evento → UN UNICO ALGORITMO
- Puede ser con o sin memoria. Si no la tiene la respuesta es siempre la misma, si la tiene puede variar

→ PROGRAMAS DE RESPUESTA SEGÚN ESTADO: la respuesta depende de los eventos que se hayan producido previamente.

- Se pueden tener algoritmos de respuesta diferentes según estado
- Se diferencian según el mecanismo para diferenciar estados:
 - GRAFOS
 - Máquinas de pila
- Para completar la respuesta ante un evento se requiere indicar para cada uno:
 - Respuesta de cada estado
 - Acción inicial

EJEMPLO: Cronómetro de 2 botones

¿Este estaría mal?
¿Por qué?



■ Grafo de estado en formato tabular

Eventos	0	1	2	3
startStop()	1	0	3	2
lap()	ANP	2	1	0

- Eventos que lanzan la excepción ANP (Acción No Permitida): lap()
- El estado de control está dado por la variable que toma los valores 0, 1, 2 y 3

■ Estado de datos

- Un entero t que contabiliza las décimas de segundo transcurridas cuando el cronómetro está arrancado. Valor inicial: 0

■ Tabla de acciones

Estados	startStop()	lap()	updateTime()
0			
1			$t = t + 1$ $display(t)$
2		$display(t)$	$t = t + 1$
3			$t = 0$ $display(t)$

- INFRAESTRUCTURA DE EVENTOS: Infraestructura de servicios de eventos que incluyen los lenguajes y las bibliotecas que admiten POE para facilitar el desarrollo y la ejecución de programas basados en eventos (BUCLE DE EVENTOS)

- REQUISITOS DE LA INFRAESTRUCTURA:

↳ DEL MANEJO DE EVENTOS: En algún momento después del evento, el sistema logra un estado consistente con los requisitos de comportamiento definidos para el evento

↳ DE TIEMPO: Se suelen establecer plazos para completar el procesamiento de un evento (PRIMORDIAL EN SISTEMAS DE TIEMPO REAL)

↳ MANEJO CONCURRENTE DE EVENTOS: los eventos se procesan en lapsos de tiempo superpuestos (DE FORMA CONCURRENTE)



- + En sistemas de tiempo real es común encontrar varios procesadores dedicados a manejar eventos
- + Los PC de sobremesa modernos tienen varios núcleos y el software de forma automática solo usa uno.
lenguajes modernos usan hilos para implementar concurrencia

- El **DISTRIBUIDOR DE EVENTOS**: Responsable de invocar a los controladores cuando se dispara un evento. Existen 2 enfoques de envío de eventos:
 - ↳ Envío directo (**PUSH**): Fuente del evento responsable de activar al distribuidor (o EMPUJA).
Podría ser que la fuente llame directamente al manejador

Lo De extracción (PULL**)**: Distribuidor consulta periódicamente o sondea las fuentes para eventos. Cuando encuentra un evento, llama a los controladores pertinentes

- **COLA DE EVENTOS**: En el envío directo, si el manejador no retorna lo suficientemente rápido antes de que llegue otro evento, pueden darse resultados impredecibles. Para ello se implementa esta ED.
 - + En la infraestructura, el tamaño de la cola y el tiempo de ejecución de los manejadores son críticos

- CONCURRENCIA : HILOS

- Uno de los requisitos de una Infraestructura de Eventos es la **CONCURRENCIA** (los eventos se procesan en lapsos de tiempo superpuestos)
- Un hilo es un mecanismo que permite implementar concurrencia en un programa.
 - * ES UNA UNIDAD DE COMPUTO MAS PEQUEÑA QUE EL S.O. PUEDE PROGRAMAR PARA EJECUTAR
- Los programas multihilo tienen varias ventajas sobre los que solo usan el hilo principal
 - + PROGRAMAS SOLO 1 HILO = PROCESAMIENTO SECUENCIAL
- Mejora la eficiencia del programa
 - + CON MULTIHILLO SE PUEDEN APROVECHAR LOS CORES DE LA CPU
 - + ACCESO A RECURSOS COMPARTIDOS

- HILOS EN JAVA:

- Objetos que ejecutan, en concurrencia con el resto del programa, el código predefinido en el método **run()**
- Hay 2 formas de crear los hilos:
 - * Heredar la clase Thread
 - * Implementando interfaz Runnable
- Son procesos ligeros → COMPARTEN MEMORIA (Objetos static)

- Comienzo de ejecución: start() arranca la ejecución concurrente del método run()
- Terminación:
 - + Retorno del método run()
 - + Lanzamiento de excepción no capturada
- Herencia de la clase Thread: Documentación PÁGINAS 49 - 53 DIAPÓS.
 - * Permite instanciar objetos que se ejecutan en un hilo por separado
 - * La clase en la que se instancian los objetos derivan de Thread y no pueden heredar de otra clase
- Implementación Runnable:
 - * Cualquier clase que implemente esto, puede ejecutarse como un hilo separado
 - * Permite heredar de otras clases, implementar varias interfaces, ...
- ESTADOS DE UN HILO:
 - enum Threads.State:
 - * NEW: Creado pero no arrancado
 - * RUNNABLE: Se está ejecutando en la JVM, aunque en un determinado momento no tenga el procesador
 - * BLOCKED: bloqueado a la espera de un semáforo, entrar en una zona synchronized, o continuar tras un wait
 - * WAITING: esperando indefinidamente a que otro hilo realice alguna acción
 - Se invocó wait() y se espera un notify()
 - Se invocó join() y se espera por el fin de otro hilo
 - * TIMED_WAITING: esperando a que otro hilo realice alguna acción, pero con un temporizador
 - Se invocó sleep(), wait() o join() con un tiempo
 - * TERMINATED: termina su ejecución

- PROBLEMAS INHERENTES DE LA CONCURRENCIA:

- EXCLUSIÓN MUTUA: Cuando 2 hilos acceden a recursos compartidos y los modifican, los resultados son impredecibles si no se garantiza el acceso exclusivo a estos
SOLO BLOQUEA EN ESCRITURA, EN LECTURA PUEDEN ACCEDER VARIOS HILOS A LA VEZ

■ Exclusión mutua con semáforos

```
public class EjemploShare implements Runnable {  
    private int n = 0;  
    private Semaphore mutex = new Semaphore(1, true);  
    . . .  
    public void run() {  
        for (int i = 0; i < 1000; i++) {  
            mutex.acquire(); // P(mutex)  
            n++;  
            n--;  
            mutex.release(); // V(mutex)  
        }  
    }  
}
```

- **SÍNCRONISMO:** Aunque los hilos se ejecuten concurrentemente (de forma asíncrona), puede necesitarse que un hilo tenga que esperar por otros
 - CORRECCIÓN DE PROGRAMAS:
 - Un programa es correcto si cumple las especificaciones
 - Satisfacción de propiedades inherentes a la programación concurrente
 - **PROPIEDADES DE SEGURIDAD:**
 - * Exclusión mutua
 - * Sincronización
 - * Interbloqueo (DEADLOCK): Garantizar que no se da la situación en la que todos los hilos estén esperando por un evento que nunca llegará
 - **PROPIEDAD DE VIVIENDA**
 - * Interbloqueo activo (LIVELOCK): Evitar situaciones en las que se ejecutan instrucciones sin lograr ningún progreso (BUENES INFINITOS POR EJEMPLO)
 - * Inanición (starvation): El sistema en su conjunto progresa, pero hay hilos que no lo hacen nunca (DIFÍCIL DE DETECTAR)

- CÓMO GARANTIZAR LA SEGURIDAD?

1. SEMAFOROS: Medio primitivo para garantizar la seguridad

→ TIPO SEMAPHORE:

- + Método `adquiere()` (P)
 - + Método `release()` (V)
 - + Heredados de `Object` los métodos `wait()`, `notify()` y `notifyAll()`

2. INSTRUCCIÓN synchronized:

- Facilita las REGIONES CRÍTICAS y MONITORES en JAVA
 - + Región crítica:
`synchronized (objeto) { SECCIÓN CRÍTICA }`
 - + Monitor: cada método que lo requiera está sincronizado
`public synchronized TIPO NOMBRE (...) {...}`

// VER CODIGO DE LA SECCION DEL WORDLE, CRONOMETRO Y PRODUCTOR-CONSUMIDOR