MVC - Modelo Vista Controlador

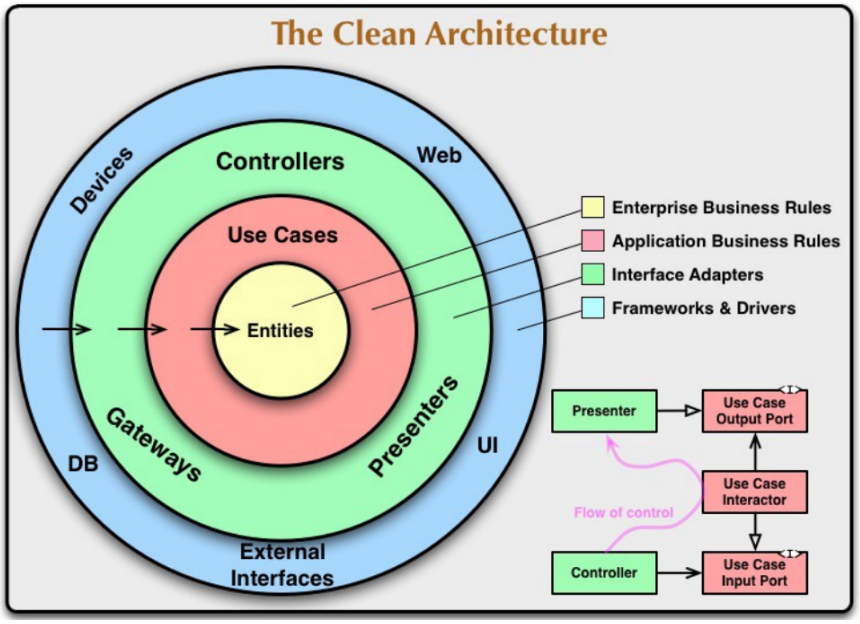
Mecanismo de despacho:

Entidades de dominio.

Casos de uso

Acciones

Interfaces



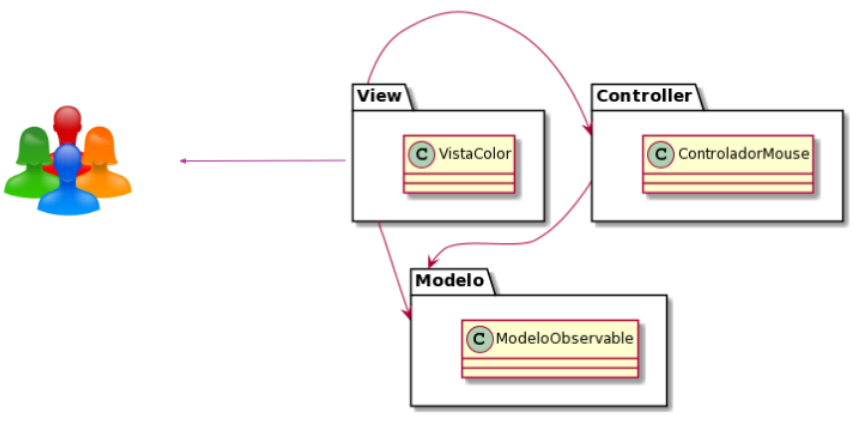
- **¿Que?**
  - Disponibilizar software

- **¿Como?**
  - Procrastinar decisiones no esenciales
  - Cohesion
  - Escritura de pruebas
  - Diseño (Paradigma)

- Patron de arquitectura diseñado para la construcción de interfaces de usuarios
- La principal motivación es la separación de responsabilidades

Tres tipos de objetos:

1. *Modelo:*
   a. Lógica y entidades de negocio o dominio de nuestra aplicación
2. *Vista:*
   a. Formas en que los objetos del modelo se muestran y representan al usuario
3. *Controlador:*
   a. Definen como la interfaz de usuario reacciona a las acciones del usuario

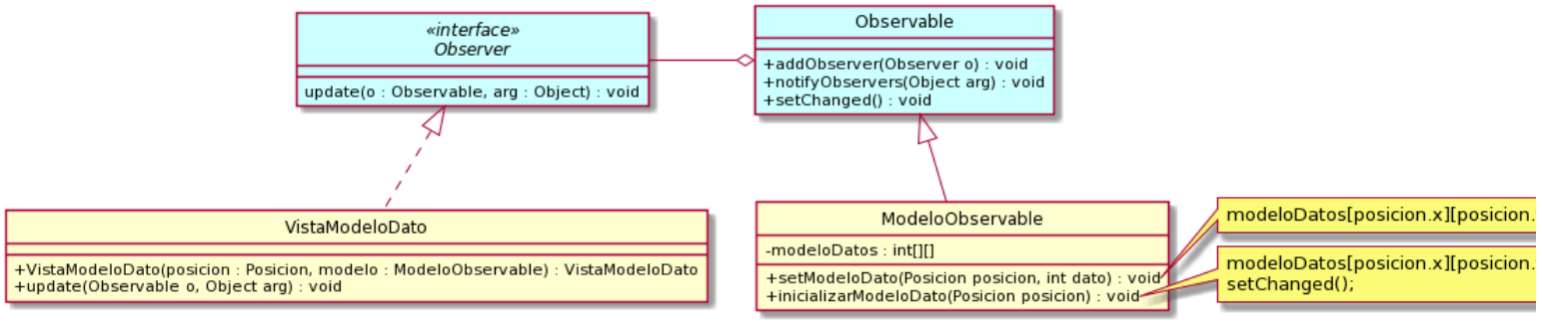


Intención:

- Definir una dependencia uno-a-muchos entre objetos de manera que cuando un objeto cambie su estado, todas sus dependencias sean notificadas y actualizadas automáticamente

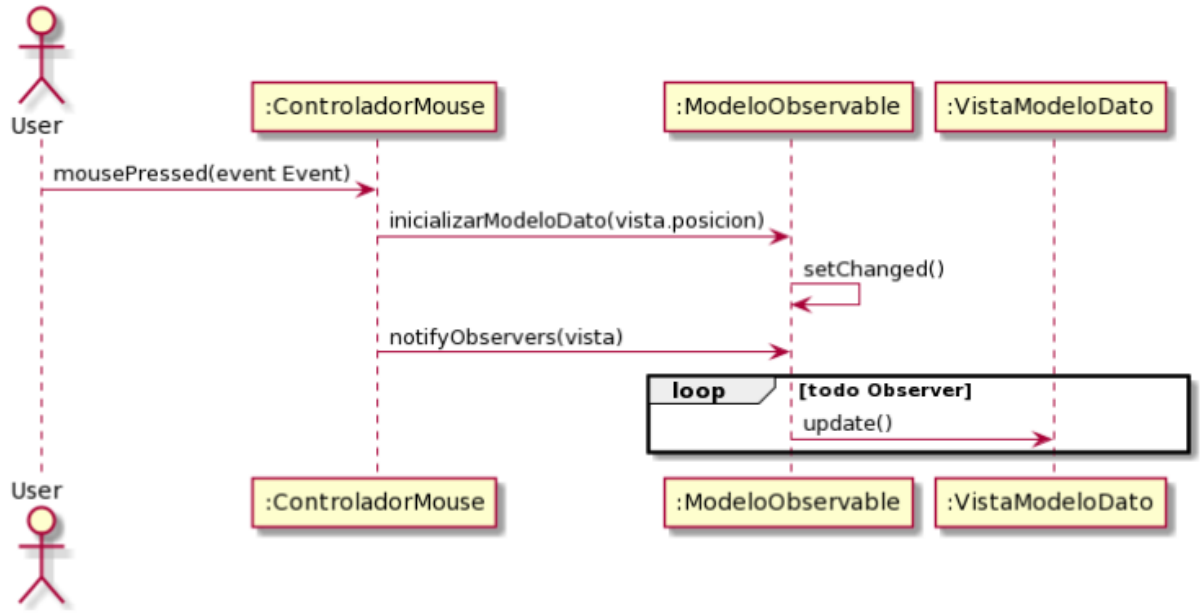
Motivación:

- Mantener la consistencia entre objetos que dependen entre si, reduciendo el acoplamiento y preservando la reusabilidad



Básicamente:

- Permite a un objeto ser “observado” (sujeto) por un conjunto de otros objetos (observador)
- Cuando un sujeto sufre un cambio en su estado, “notifica” a los observadores a través de un método especial



Persistencia

Persistencia significa trascender en el tiempo y/o espacio, en la informática hace referencia a la característica que puede tener un dato o estado que le permite sobrevivir al proceso que lo creo.

Persistencia en POO

Un ambiente orientado a objetos debe permitir que los objetos persistan, para mantener su vida mas allá de la vida de la aplicación.

Este concepto nos permite que un objeto pueda ser usado en diferentes momentos a lo largo del tiempo, por el mismo programa o por otros, así como en diferentes instalaciones de hardware en el mismo momento.

Objeto persistente:

- Es aquel que conserva su estado en medio de almacenamiento permanente, pudiendo ser reconstruido por el mismo proceso que lo generó u otro, de modo tal que al reconstruirlo se encuentre en el mismo estado que se lo guardó.
- Al objeto persistente lo llamamos *efímero*

Tipos de persistencia:

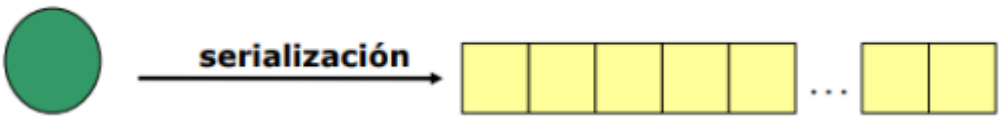
- *Nativa:*
    - Provista por la plataforma, ejemplo Java, Smalltalk
- *No Nativa:*
    - A través de una biblioteca externa.
    - Programación a mano.

El ideal en cuanto a persistencia sería que cada clase tenga un método para guardar objetos y otro para recuperarlos, consistentes entre sí. Sin embargo, hay una contradicción entre este ideal y el principio de separación de intereses.

Lo más habitual es que los lenguajes orientados a objetos no soporten la auténtica persistencia, sino una variante que exige guardar y recuperar objetos en forma activa.

Serialización

Proceso que consiste en convertir la representación de un objeto en un *stream (flujo o secuencia) de bytes*



Reconstruir un objeto a partir de un stream de bytes se denomina deserialización

Formatos de serialización

- *Formato propietario:*
  - Más eficiente en términos de uso de almacenamiento y tiempo de traducción
  - Solo sirve para comunicar aplicaciones basadas en la misma plataforma

- Uso de lenguajes estándar:
  - ejemplo JSON
  - Es el más popular lenguaje de intercambio de información
  - Al ser formato de texto ocupa más espacio y su estructura no propietaria precisa hacer transformaciones en términos de computo.

Obs: Java posee serialización automática en formato propietario, para trabajar con JSON hay bibliotecas

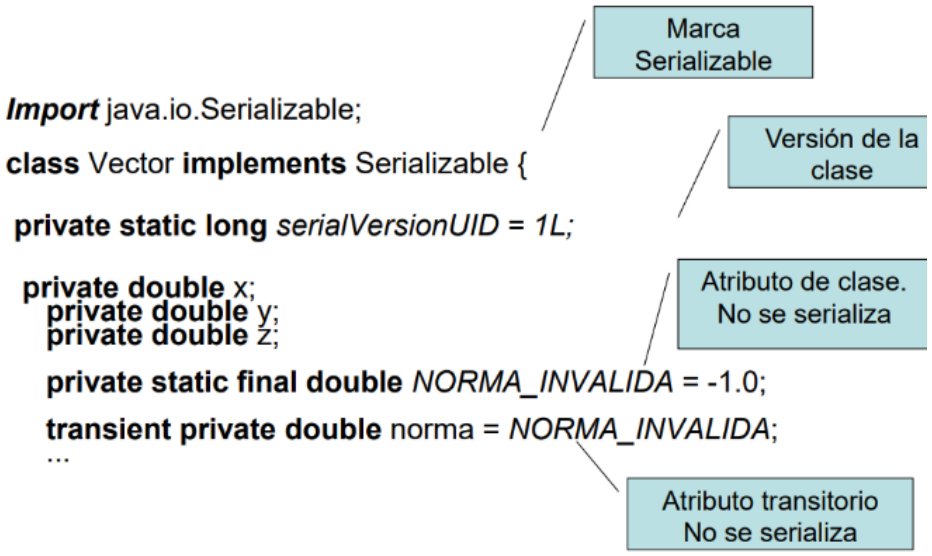
Persistencia y serialización:

- Para persistir debo primero serializar.
- Serializar no implica necesariamente persistir.
- Otras acciones luego de serializar:
  - Enviar por red.
  - Mantener en memoria
  - Enviar a una impresora, etc.

Para define una clase serializada, debe implementarse la interfaz Serializable.

- No tiene métodos
- Sirve para avisarle a la maquina virtual que la clase pued serializarse (maker interface)
- Todas las subclases de una clase serializable son serializables tambien
- Por defecto, se serializan todos los atributos de un objeto que no posean el modificador *transient (transitorio)*
- Si un objeto serializable tiene referencias a otro objeto serializable, tambien lo serializa
- Si algún objeto del “árbol de objetos” a serializar no es serializable se lanza la excepción *NonSerializableException*
- Los objetos serializables deben tener un constructor sin parámetros para poder ser deserializados correctamente

Serialización en Java



Persistencia en Java

- Paquete java.io
  - De objeto a archivo

```
UnaClase objeto = new UnaClase();
OutputStream fos = new FileOutputStream("objeto.dat");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(objeto);
```

  - De archivo a objeto

```
InputStream fis = new FileInputStream("objeto.dat");
ObjectInputStream ois = new ObjectInputStream(fis);
Clase objeto = (Clase) ois.writeObject(objeto);
```

Persistencia Nativa:

- *Ventajas*:
  - Es nativa del lenguaje (casi no hay que programar)
  - Resuelve referencias circulares
- *Desventajas:*
  - No es portable a otros lenguajes de manera sencilla
  - No es óptima en cuanto a tamaño (tiene overhead alto)
  - La información en el archivo es binario
  - No es extensible ni reparable

Serialización en Smalltalk

- Objeto a Archivo

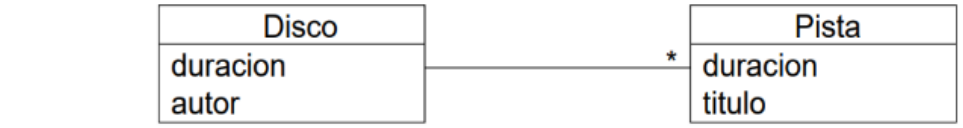
```
objeto := MiClase new.
rr := ReferenceStream fileNamed: 'objeto.bin'.
rr nextPut: objeto; close.
```

- Archivo a Objeto

```
rr := ReferenceStream fileNamed: 'objeto.bin'.
objeto := rr next.
rr close.
```

Ej: Serialización no nativa

- Persistencia en Texto (XML)



instancias

```
Disco disco = new Disco(50, “Queen”);
disco.addPista(new Pista(4,”the show must go on”);
disco.addPista(new Pista(3, “inuendo”);
```

Stream de texto

```
<disco duracion=“50” autor=“Queen”>
        <pista duracion=“4” titulo=“the show must go on” />
        <pista duracion=“3” titulo=“inuendo” />
</disco>
```

Persistencia no nativa (XML):

- *Responsabilidad:*
  - Cada clase conoce como serializarse
- *Cada clase serializable tendrá:*
  - Un método serializar que devolverá un nodo XML
  - Un constructor sobrecargado que recibirá un nodo XML

Persistencia no nativa(XML)

```
public class Disco{
   private int duracion;
   private string autor;
   private List pistas = new ArrayList();

   public Disco(XMLNode nodo){
         duracion=Integer.ParseInt)nodo.getAttribute("duracion"));
         autor=nodo.getAttribute("autor");
         List nodosPistas = nodo.getChildrens("pista");
         for(XMLNode nodoPista : nodosPistas){
                  pistas.add(new Pista(nodoPista);
         }
   }

   public XMLNode serializar(){
         XMLNode nodo = CrearNodo();
         nodo.setAttribute("autor",this.autor);
         nodo.setAttribute("duracion",this.duracion);
         for(Pista pista : this.pistas){
                  nodo.addChildren(pista.serializar());
         }
   }

}
```

Deserialización

Serialización

La clase pista es similar

- **Queda en manos del programados:**
  - *Cuestiones de diseño*
    - Responsabilidad:
      - Cada clase sabe como persistir
      - Existe un gestor externo que sabe como persistir las clases
    - Formato:
      - Binario, texto pano, texto jerárquico (XML)
    - Identidad de los objetos:
      - Referencias circulares, duplicaciones de objetos, etc
  - *Mayor versatilidad*
  - *Mayor complejidad*

Concurrencia

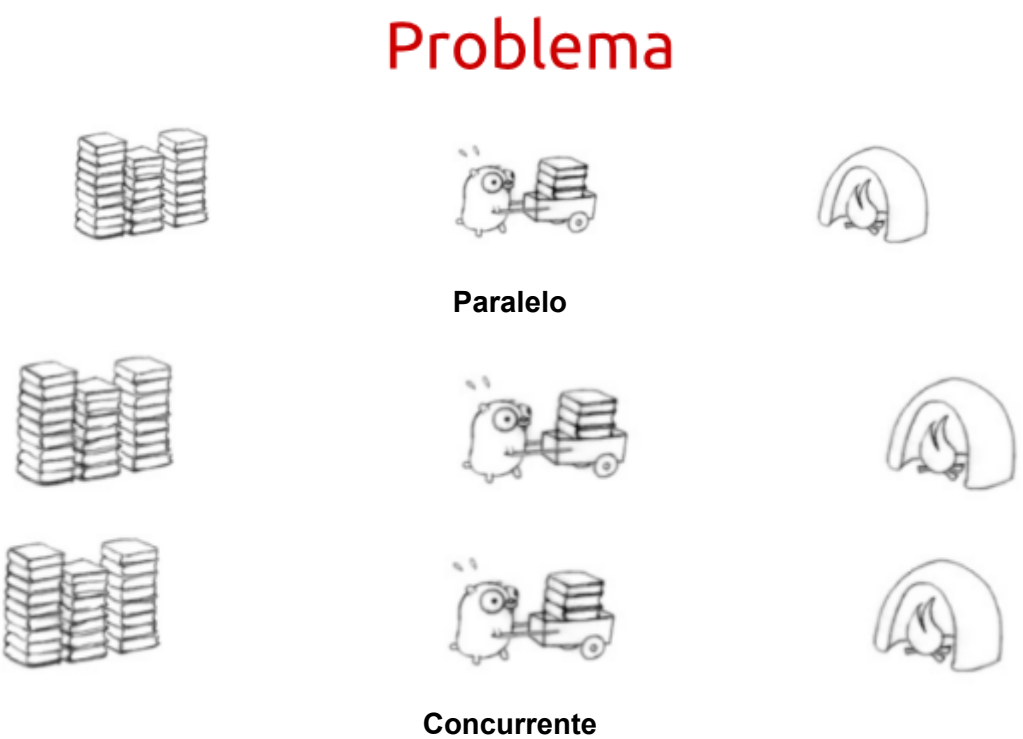
¿Por qué concurrencia?

- Performance
- Tiempo de respuesta al usuario
- tiempo de ejecución de una aplicación
- “Mundo paralelo”
- Aprovechar el hardware

“Concurrencia es tratar de lidiar con muchas cosas a la vez, mientras que paralelismo es hacer muchas cosas a la vez” - Rob Pike

Conceptos:

- Concurrencia es la composición de la ejecución de “cosas” independientes. Es sobre la estructura de los problemas, descomponer tareas en tareas más pequeñas.
- Paralelismo es ejecutar tareas en simultáneo
- Concurrencia **no implica** paralelismo



¿Qué es un thread?
Secuencia independiente de instrucciones ejecutándose dentro de un programa.

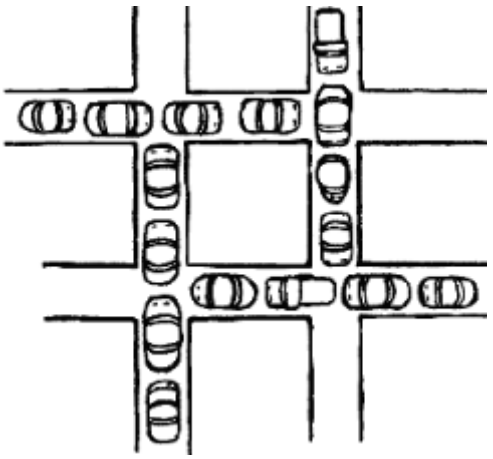
- Race condition:
  - Se da cuando varios threads pueden acceder a recursos compartidos (código). El resultado del programa depende de cómo se intercalan los threads.
- Critical section
  - Sección de código que necesita ser ejecutada en forma atómica por un solo hilo a la vez

Locks:
- Se basa en el uso de una variable de exclusión mutua (mutex).

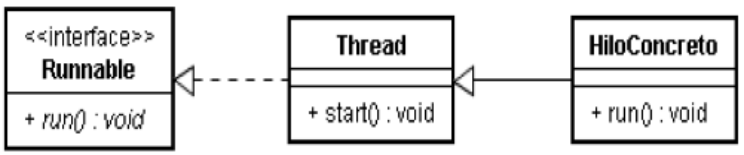
Monitores:
- Objetos thread-safe, sus métodos están sincronizados (mutex).

Conditional variable:
- Mecanismo de bloque con una señalización

Semáforos:
- Aparece cuando entre dos o más threads uno obtiene un recurso y no lo libera generando un bloqueo.



1. **Heredar de Thread**
   (Template Method)

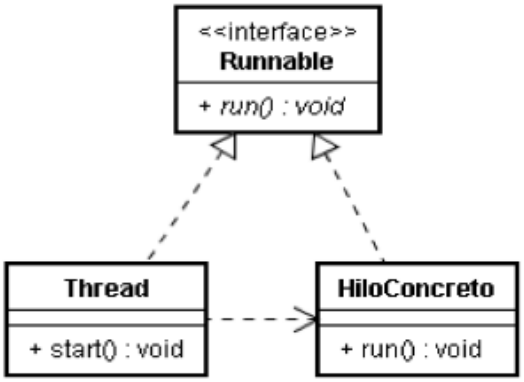



```
public class HiloConcreto extends Thread
{
  @Override
  public void run() {
    // código del hilo
  }
}
```

```
....
HiloConcreto unHilo = HiloConcreto();
unHilo.start();
...
```

2. Implementar Runnable

Más flexible



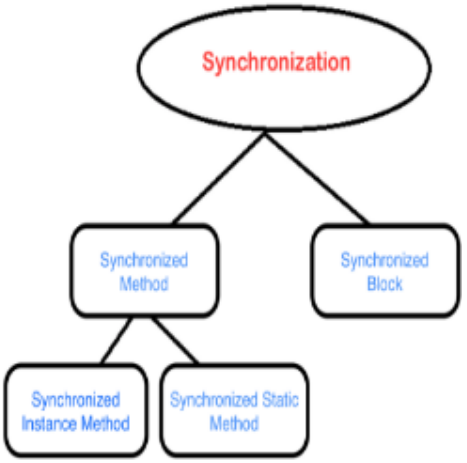

```
public class HiloConcreto implements Runnable {
  @Override
  public void run() {
    // código del hilo
  }
}
```

```
...
new Thread(new HiloConcreto()).start();
```

```
public synchronized void metodo() {
   //...
}

public void metodo() {
   synchronized(this) {
      //...
   }
}

public static synchronized void
metodo() {
   //...
}
```



Más sobre la API

- **t.interrupt()** -> Interrumpe el thread.
- **t.join()** -> Espero a que el thread termine.
- **Thread.sleep(mls)** -> Duerme el thread por mls.

General

- **objeto.wait()** -> El thread queda en espera.
- **objeto.notify()** -> Despierto a uno de los thread que esperan.
- **objeto.notifyAll()** -> Despierto a todos los que esperan.

Condition Variables

Consideraciones:

- Debuggear es complicado[1]
- Agregar **println()** puede alterar el resultado
- Demasiada **sincronización** perdemos la ventaja de usar threads
- Poca **sincronización** genera errores difíciles de detectar

[1] No doy más, pero desaprobar no es opción.
Caspian 2C2023