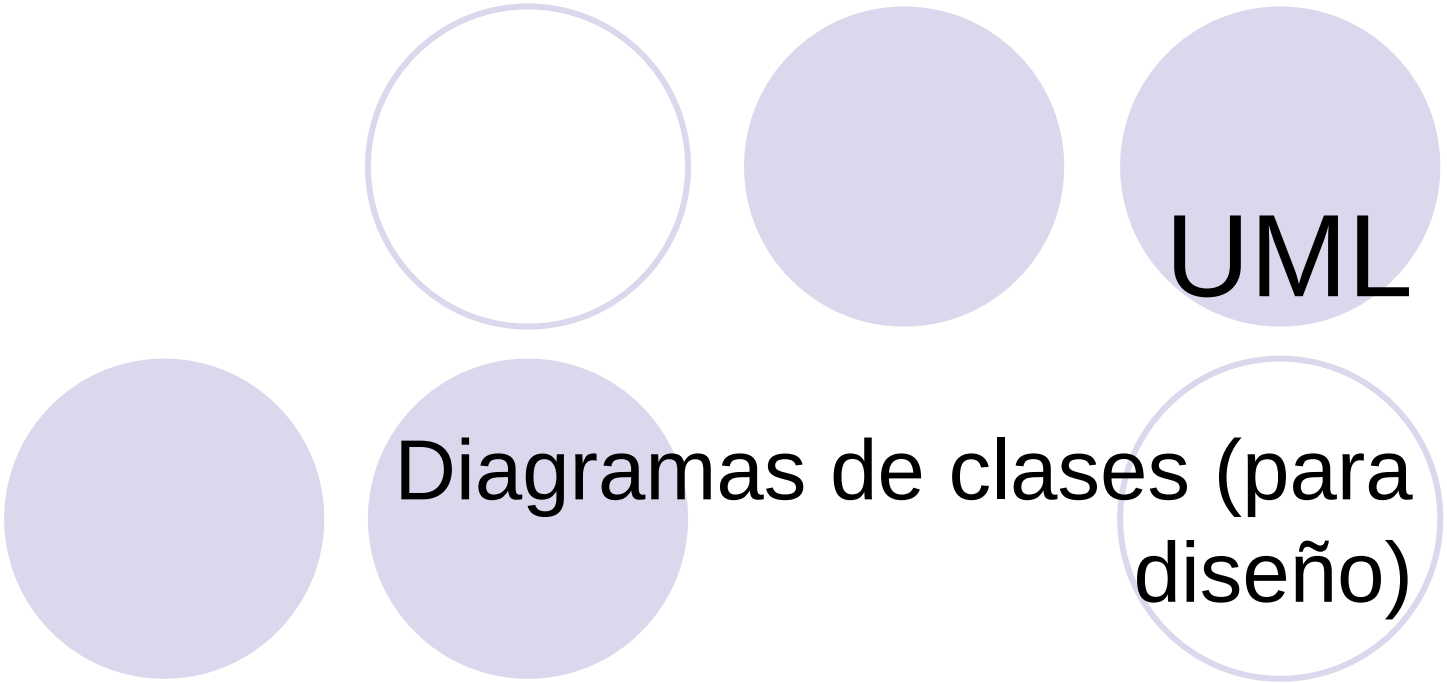


UML

Diagramas de clases (para diseño)

Índice

1. ¿Por qué diagramas de clases de análisis y de diseño?.
2. Diagramas de clases de diseño.

Análisis y diseño

- ¿Por qué diagramas de clases de análisis y diseño?
  - En el análisis queremos contar unas cosas y en el diseño otras.
  - Algunos símbolos de UML se adaptan mejor al análisis y otros al diseño.
  - Al tener una paleta de herramientas más pequeña, dudamos menos.

Análisis y diseño

- ***Usaremos:*** generalización, agregación, composición, clases, paquetes, etc.
- ***No usaremos:*** asociaciones.
- ***Nuevos elementos:*** dependencias, realizaciones, *nesting*.
- Los paquetes ya no son subsistemas sino agrupaciones de clases.
- Usaremos estereotipos para reflejar características del diseño / plataforma de implementación.

Diagramas de clases de diseño

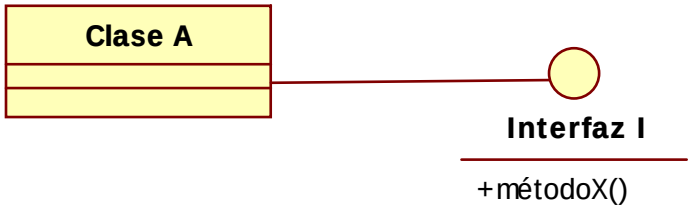
Diagramas de diseño

- Relación de dependencia.
  - Significa que un elemento es estructuralmente dependiente de otro elemento.
  - Esto es, cambios en un elemento afectarán a cambios en el elemento dependiente.


```
class A {
  B b;
 ….
}
```

Diagramas de diseño

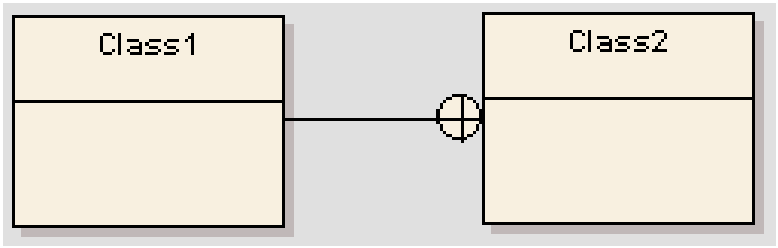
- Relación de implementación.
  - Significa que un elemento implementa a otro.
  - UML no define con precisión el significado de “implementar”.
  - Habitualmente utilizado en interfaces

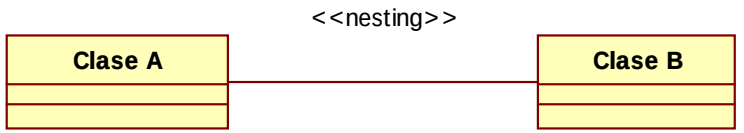



```
class A
  implements I
{
  métodoX() {
        …;
  }
 ….
}
```

Diagramas de diseño

- Relación de *nesting*
  - Aún no una traducción oficial (*¿anidamiento?*).
  - Define un elemento *contenido* dentro de otro.
  - No existe en StarUML (estereotipar).

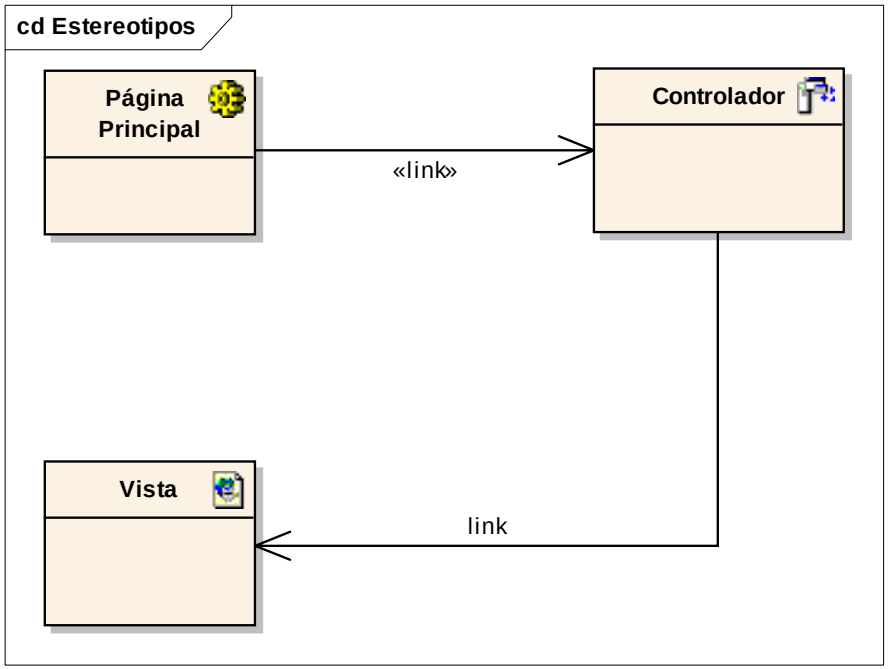





```
class Class1
{
  ....
  class Class2 {
       …;
  }
}
```

Diagramas de diseño

- Estereotipos:
  - Un nuevo tipo de elemento.
  - Añade *algo nuevo*: una semántica o comportamiento, restricciones, métodos atributos.
  - Útil para capturar detalles de la plataforma de implementación.



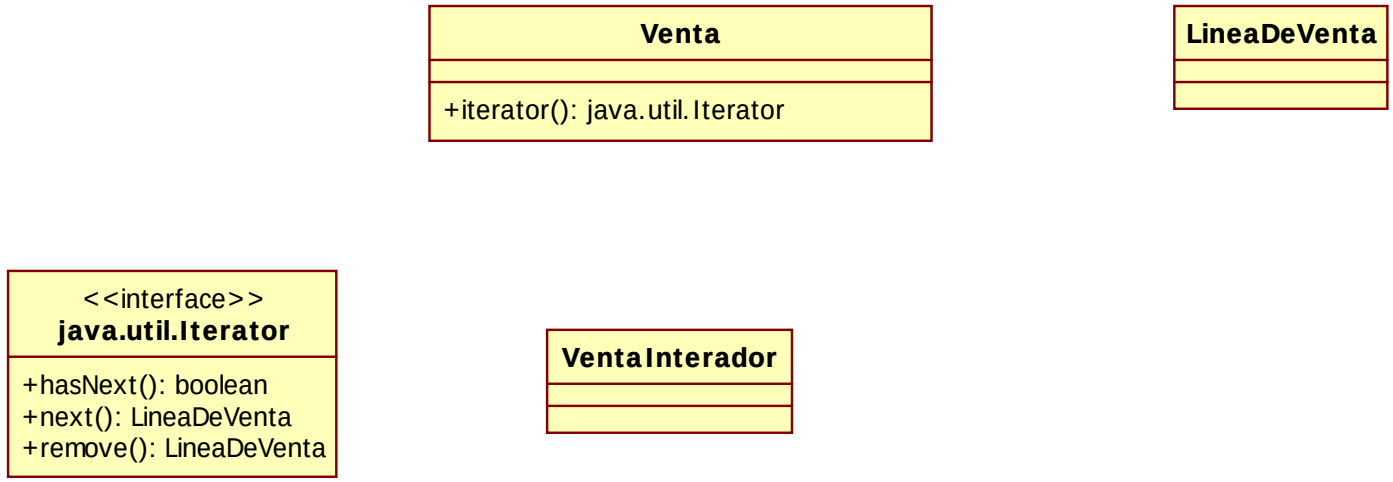
Estereotipos:

- Web page.
- Servlet
- JSP.
- Link

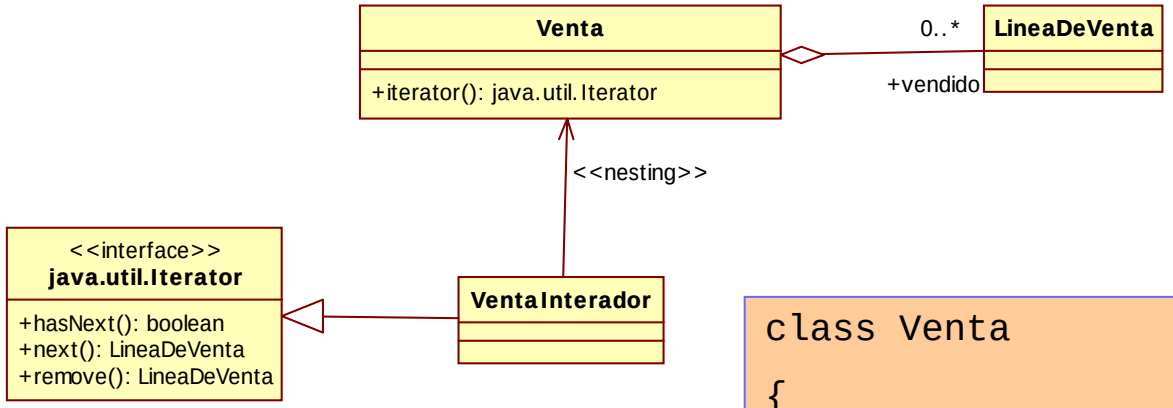
Diagramas de diseño

- Ejercicio:
  - Una clase Venta contiene una lista de objetos LineaDeVenta.
  - Deseamos implementar nuestro propio iterador (en *Java*) para recorrer todas las líneas de venta.
  - Suponemos que las líneas de venta se almacenan en un array.

Diagramas de diseño



Diagramas de diseño



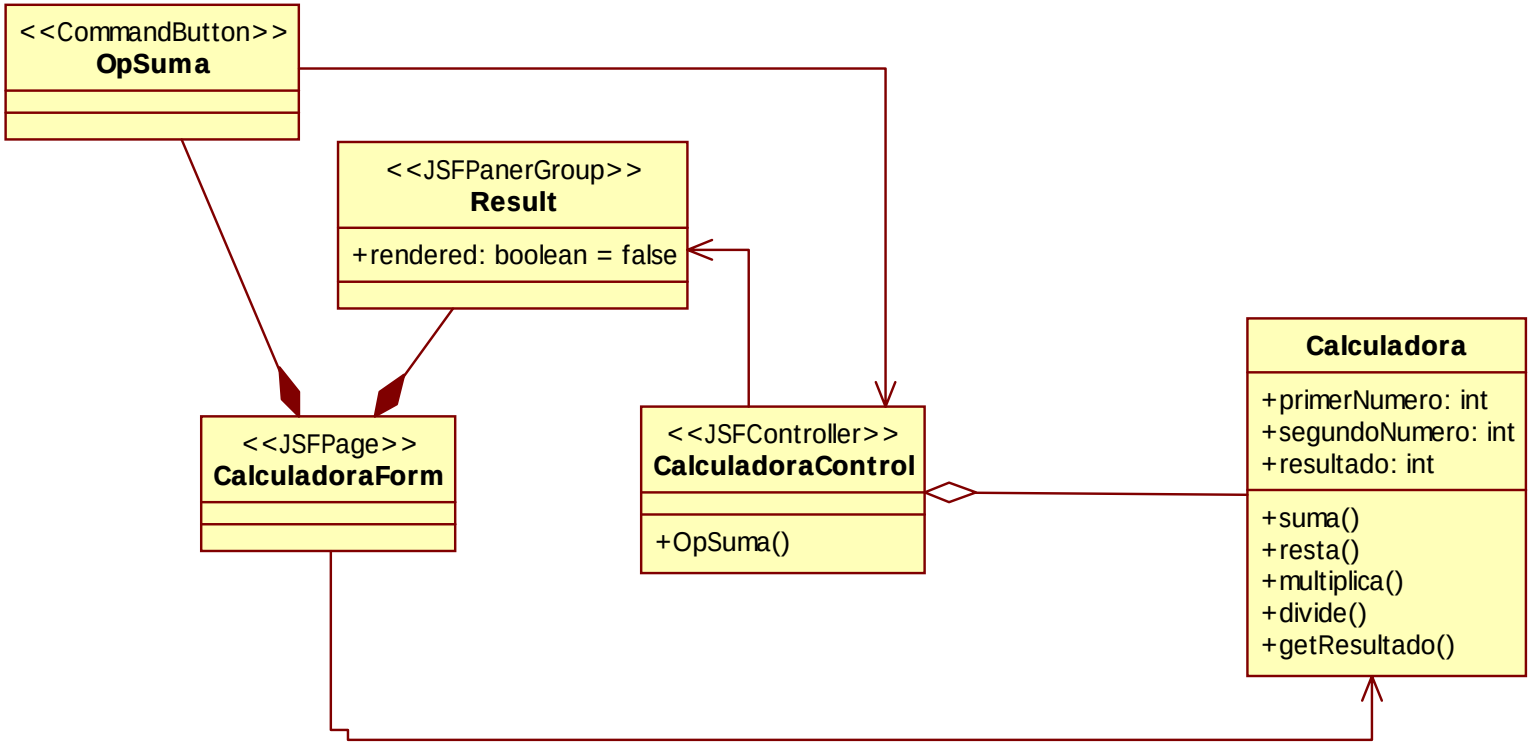

```
class Venta
{
  LineaDeVenta[] vendido;
  Itertor<LineaDeVenta> iterator() {
    return new VentaIterator<LineaDeVenta> ();
  }

  class VentaIterator implements Iterator<E> {
    E next() { ….. }
  }
}
```

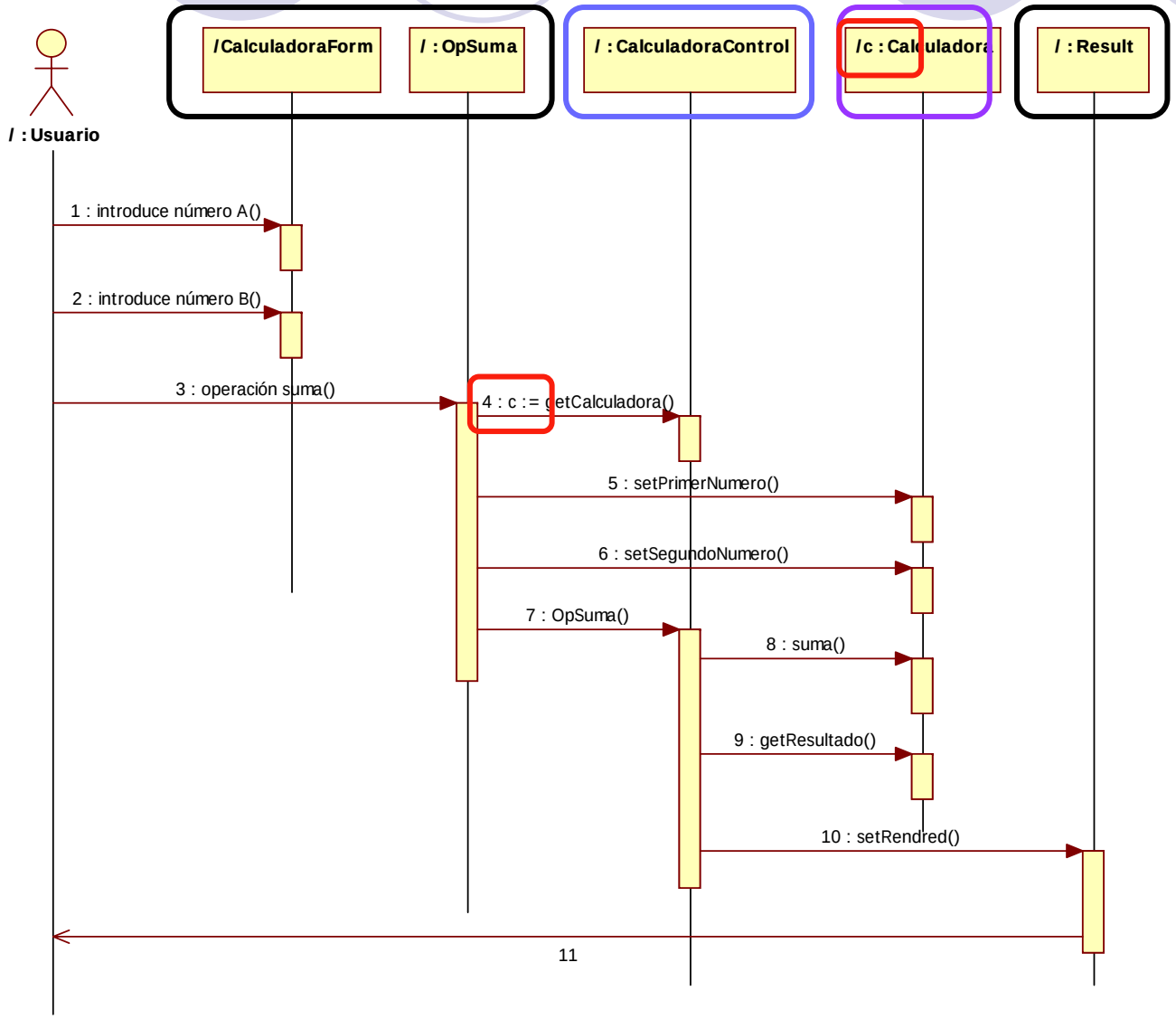
Diagramas de diseño

- Una calculadora en JSF

Modelar la operación suma (con éxito)



Diagramas de diseño

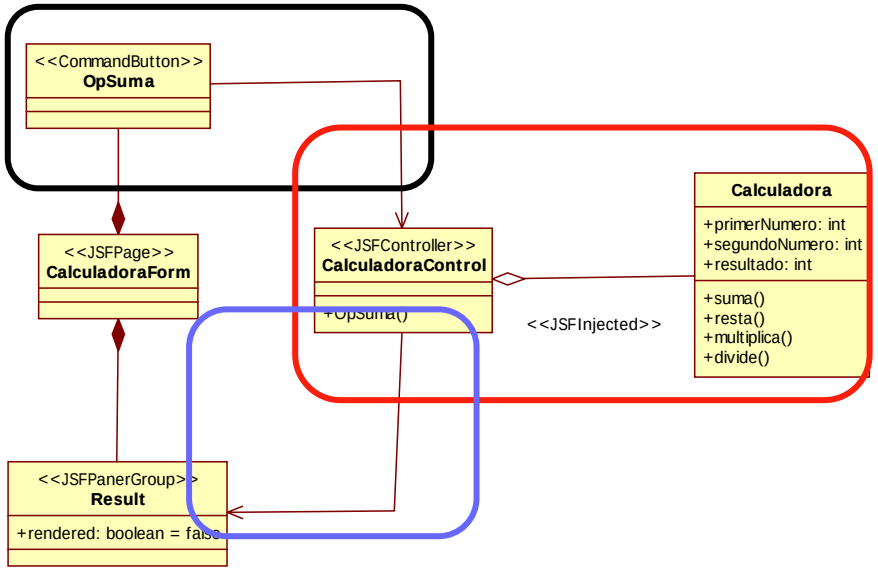


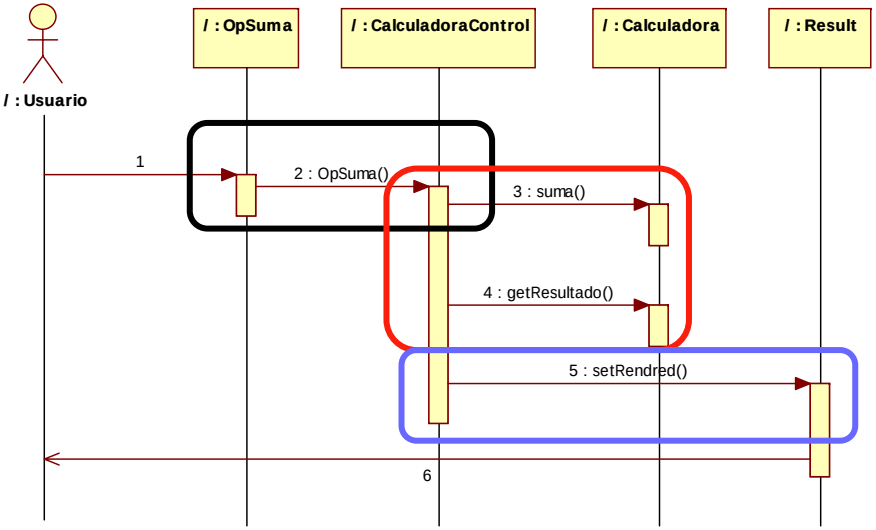
Interfaz de usuario.

Controlador.

Modelo.

Diagramas de diseño



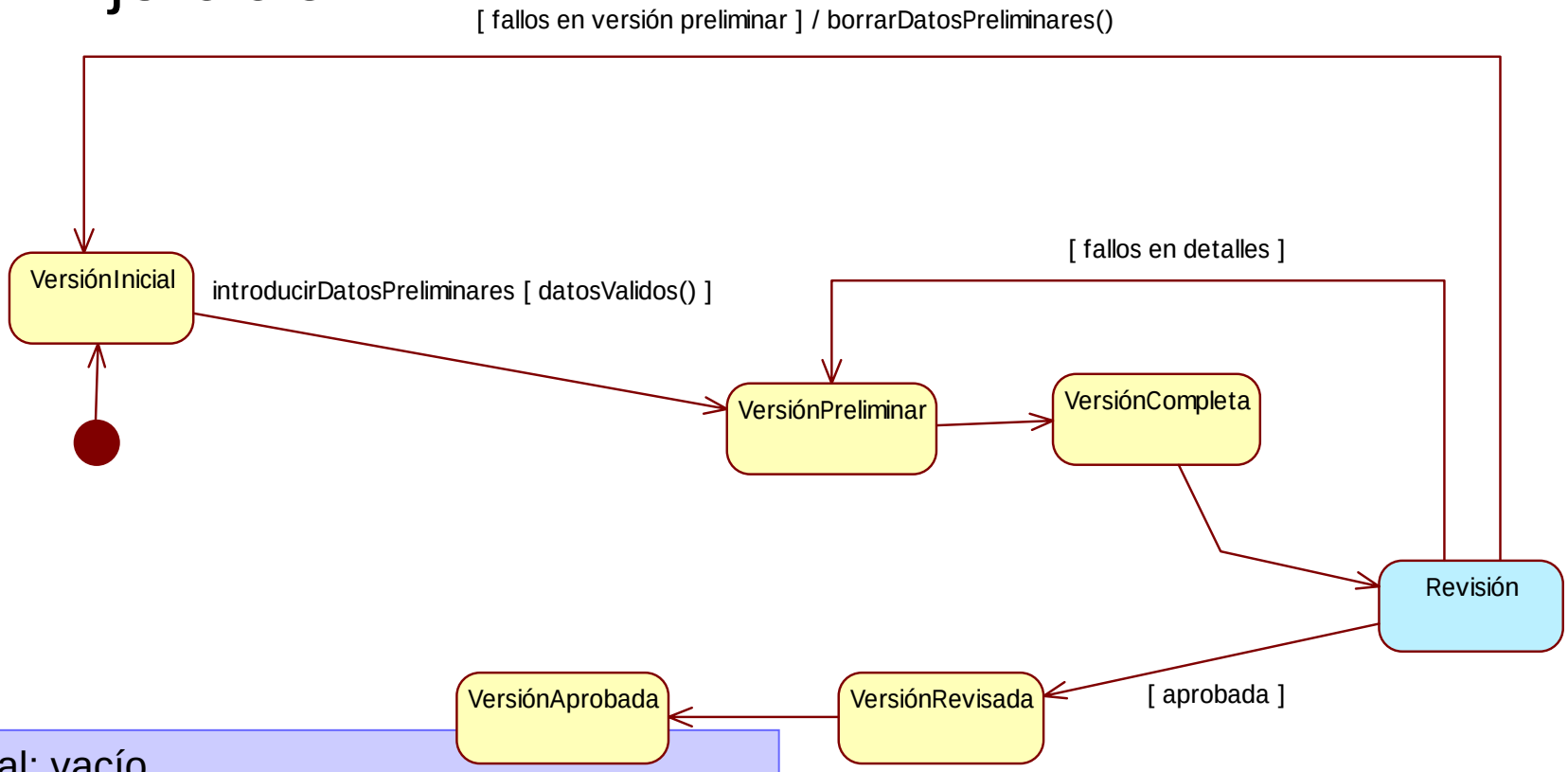


Diagramas de diseño

- Ejercicio: ciclo de vida de un informe.
  - Una primera versión es creada por un auxiliar administrativo.
  - Después pasa al director para completar los detalles.
  - Después pasa a un revisor.
  - Cuando está revisada, se aprueba por el jefe de servicio.

Diagramas de diseño

- Ejercicio



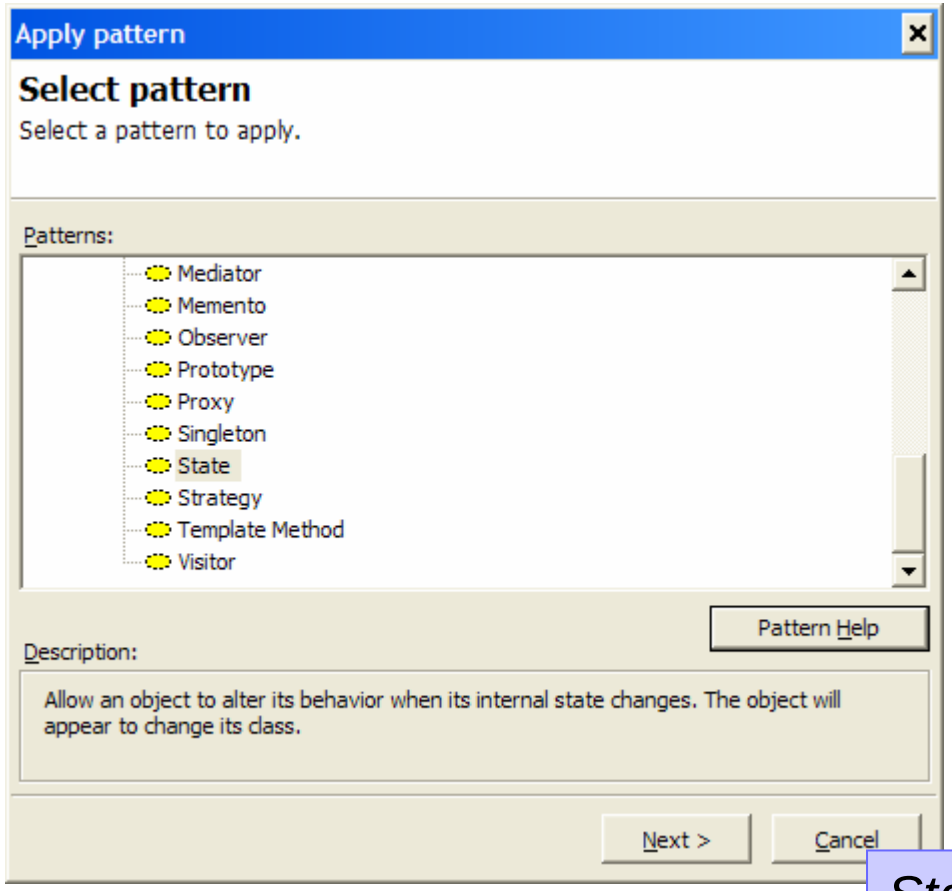
Inicial: vacío.

Preliminar: datos preliminares.

Completo: toda la información.

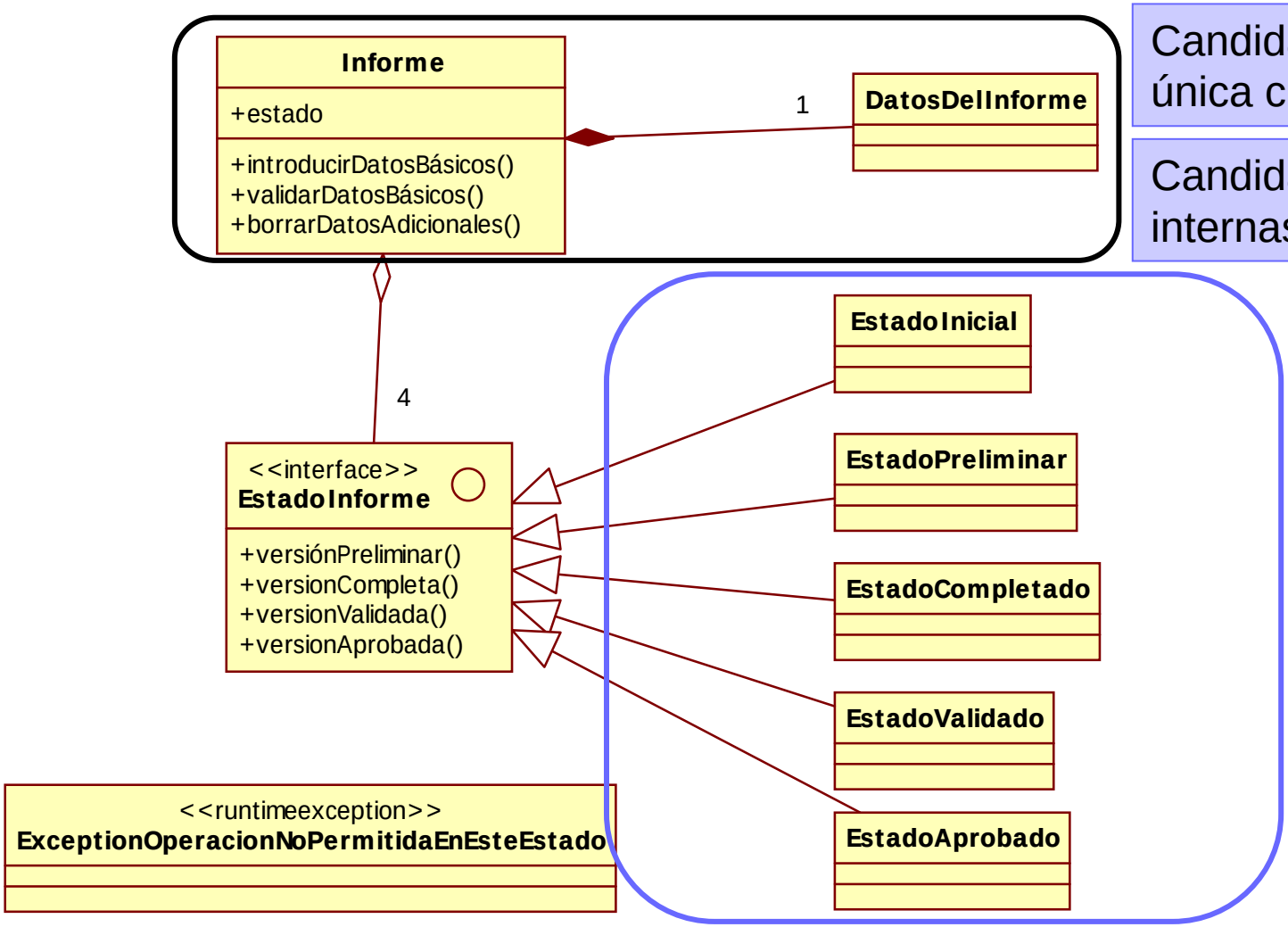
Al retroceder a un estado anterior, se pierde ***toda*** la información almacenada.

Diagramas de diseño



StarUML incluye ayuda sobre los patrones y elementos ya predefinidos.

Diagramas de diseño

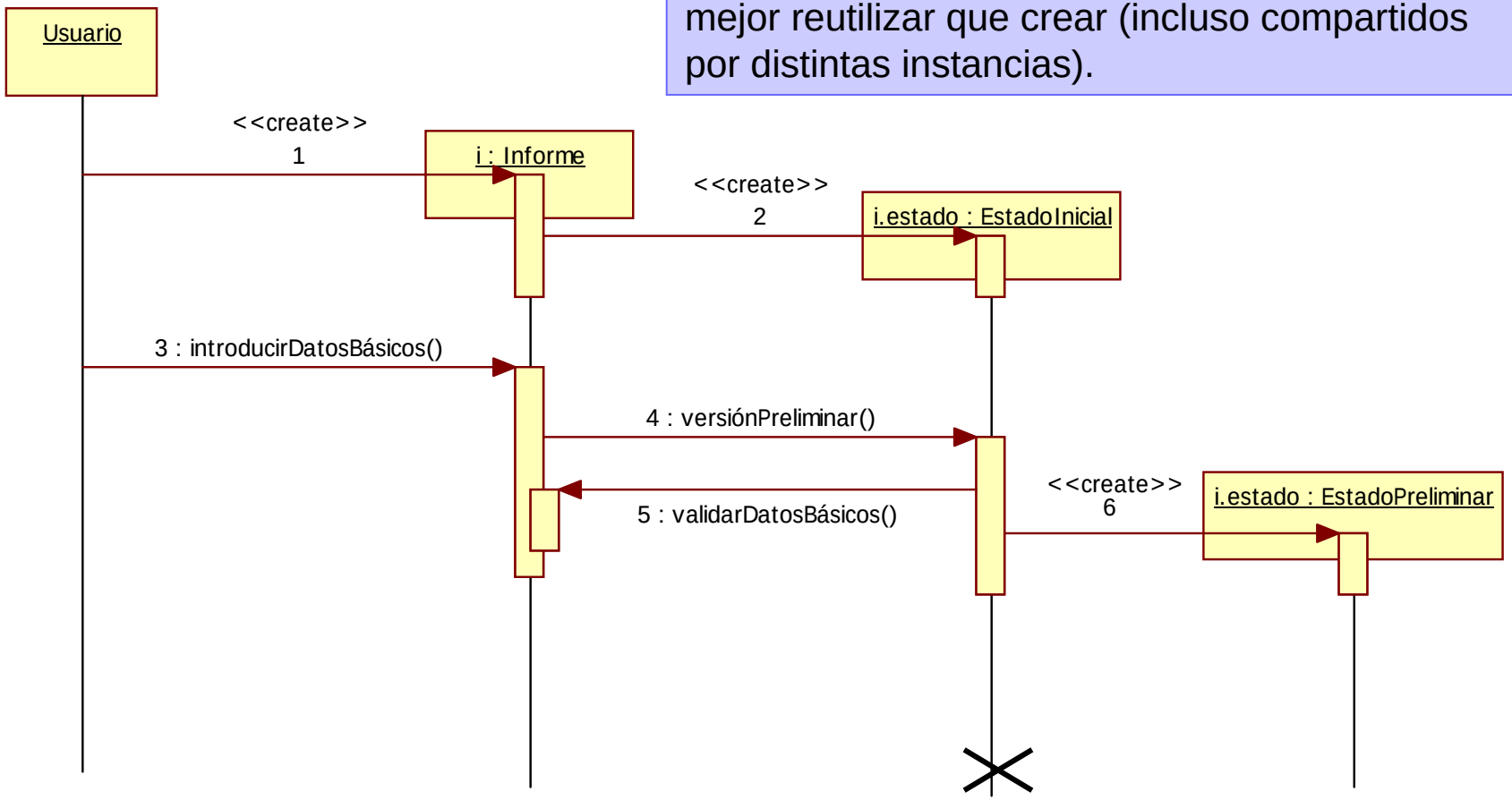


Candidatas a ser una única clase.

Candidatas a ser clases internas.

Diagramas de diseño

Dado que los propios estados son *stateless*, mejor reutilizar que crear (incluso compartidos por distintas instancias).



Diagramas de diseño

¿Cómo podemos asegurarnos de que los usuarios humanos del sistema no hagan lo que no deban?

Mediante sus interfaces. Nunca debería llegarle a un usuario un error provocado por la excepción *ExcepcionOperacionNoPermitidaEnEsteEstado* (por eso es *Runtime*).

Pero los programadores sí deben conocer las secuencias correctas y las secuencias incorrectas.

Para ello tiene la especificación (*máquina de estados*).

Además, probarán que, en ningún momento, ninguna secuencia de acciones del sistema arroja la excepción.

Y, además, probarán que todas las secuencias erróneas lanzan la excepción (en el momento correcto).

Lo cuál es imposible *(¿mutación de secuencias?).*

Diagramas de diseño

- Otros patrones complementarios.
  - En cada estado puedo realizar cualquier operación.
  - Aunque salte un error al intentar cambiar de estado, la operación ya se ha realizado.
  - ¿Cómo podemos invalidar operaciones según el estado?.

Patrón decorador:

- Decorando los métodos del informe según el estado.
- Cada estado tiene su propio decorador los cuáles me redefinen los métodos a evitar.