Metodologías de Programación I

Introducción a Objetos

Análisis y Diseño OO ADOO -Introducción

- Construimos software para resolver problemas. Las personas tienen problemas.
- *Ergo, Construimos software para las personas.*

ADOO –Un buen software

Tres pasos para construir buen software:

- Asegurarnos de que el software que hacemos es lo que el cliente necesita.
- Aplicarlos principios de la Orientación a Objetos
- Esforzarnos por hacer diseños mantenibles y reusables.

ADOO -Requerimientos

- “Creamos” software para las personas
- Necesitamos saber que es lo que las personas quieren que tenga lo que desarrollemos
- Eso se llama ***requerimientos***

ADOO -Algunas cuestiones del DOO

- Flexibilidad
- Encapsulamiento
- Funcionalidad
- Usar soluciones probadas

ADOO -Recordar

- Hacer las cosas simples. Eso siempre funciona. **Menos es más.**
- No c***rear*** problemas para ***resolver*** problemas!

ADOO -Sugerencias

Desarrollo Iterativo

- Repetirlos procesos de desarrollo una y otra vez.
- Iteraciones cortas

Desarrollo Incremental

- Agregar “partes” por vez
- No todo “de una”

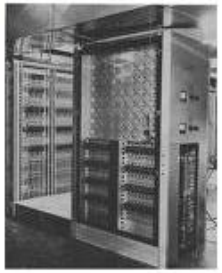
ADOO –Superación

Preguntarnos todo el tiempo:

¿Podemos hacerlo mejor?

ADOO -Cambios

- Los cambios ocurren! Es ley
- Los entornos cambian
- El software y el hardware cambian
- Debemos estar preparados para enfrentarlos





Introducción a Patrones

Observer
Composite
Adaptor
State
Singleton

Patrones.

- Al comenzar un diseño lo ideal sería que un asesor de nuestro equipo se tome el trabajo de extraer la parte común de muchos otros diseños exitosos y se quede con las mejores ideas a nivel diseño para poder aplicarlas en nuestro proyecto.

- Esto nos asegura el éxito de nuestro diseño, siempre y cuando hayamos entendido la idea expresada por el asesor, la apliquemos en el lugar correcto y de manera correcta.

Patrones.

- La idea de los patrones de diseño es cumplir la función de asesor….PERO es responsabilidad nuestra entender correctamente lo que cada uno de ellos propone y aplicarlos de manera correcta.
- Los patrones de diseño no son los únicos patrones que existen, por ejemplo existen también los patrones de arquitectura.
- Patrones que atacan diferentes niveles de abstracción

Patrones.

- Los patrones de diseño son maneras convenientes de *reutilizar código* orientado a objetos entre proyectos y programadores.

- La idea detrás de los patrones de diseño es simple: anotar y catalogar aquellas interacciones comunes entre objetos que se encuentran frecuentemente y son útiles.

Patrones.

- ¿Qué es entonces un patrón?
  - *“Un patrón es una solución a un problema en un determinado contexto”*

Patrones.

- Pero entonces todo es un patrón? NO!
- Los buenos patrones deben tener al menos las siguientes características:
  - Solucionar un problema reiterativo
  - Ser un concepto probado
  - La solución no es obvia
  - Describe participantes y relaciones entre ellos

Patrones.

- Podemos concluir entonces que para poder asegurar que hemos conseguido un patrón, los siguientes puntos deberán ser demostrados:
  - Los patrones indican repetición, si algo no se repite, no es posible que sea un patrón
  - Un patrón se adapta para poder usarlo (adaptabilidad)
  - Un patrón es útil (utilidad)

Patrones.

- **La repetición** es una característica cuantitativa pura. La podemos probar demostrando la cantidad de veces que fue usada (regla de 3).
- **La adaptabilidad** es una característica cualitativa. (como el patrón es exitoso)
- La **utilidad** también es una característica cualitativa. (porque es exitoso y beneficioso)

Patrones. Diferentes definiciones

- Soluciones recurrentes a problemas de diseño que se presentan comúnmente.
- Focalizan en la reutilización de problemas de diseño recurrentes.
- Un patrón trata un problema de diseño recurrente que aparecen en situaciones específicas del diseño y presenta una solución al mismo.

Patrones.

- Describe un problema de diseño recurrente y una solución al mismo de manera que para cada ocurrencia del problema, uno pueda instanciar la solución para ese problema
- Descripción de cosas que los diseñadores conocen
- Abstracciones de alto nivel para la organización de sistemas
- Discusiones y registros de problemas de diseño

Patrones.

- Recuerdos de soluciones para resolver problemas
- Abstracciones de alto nivel para la organización de sistemas
- Discusiones y registros de problemas de diseño
- Conjunto de reglas que describen como realizar ciertas tareas en el desarrollo de software

Patrones.

- Refinemos entonces la definición de patrón: Según Christopher Alexander:

"Un patrón de diseño describe un problema que ocurre una y otra vez en un contexto o medio, y describe el núcleo de la solución a ese problema, tal que pueda ser utilizada infinitamente sin hacer necesariamente lo mismo de la misma manera dos veces"

Que viene “gratis” con los Patrones de Diseño…

- Experiencia.
- Sus nombres forman, colectivamente, un vocabulario que ayuda a los desarrolladores a comunicarse mejor, a hablar en el mismo idioma
- Si la documentación usa patrones, simplifica la lectura y comprensión.
- Si están bien aplicados, hacen los diseños orientados a objetos más flexibles, elegantes y por último reutilizables

Que viene “gratis” con los Patrones de Diseño…

La creatividad:

- Un patrón de diseño no garantiza nada por sí solo. Los patrones no buscan suplantar al humano en el proceso creativo

- Necesitamos seguir usando la creatividad para aplicar correctamente los patterns.

¿Cómo ayudan los Patrones a solucionar problemas de diseño?

1-Encontrando objetos apropiados.

- La parte compleja del diseño OO es descomponer el sistema en objetos debido a todos los factores que entran en juego:
  - granularidad
  - encapsulamiento
  - Dependencia
  - evolución
  - Reusabilidad
- Objetos que no se desprenden de la especificación de requerimientos
- Decidir entre Herencia o Composición

¿Cómo ayudan los Patrones a solucionar problemas de diseño?

2-Determinando el número y tamaño de los objetos.

- Los objetos pueden variar enormemente en tamaño y número. ¿Cómo decidimos como debería ser un objeto?.

¿Cómo ayudan los Patrones a solucionar problemas de diseño?

3-Especificando el protocolo de los objetos:

- Los patterns de diseño ayudan a definir el protocolo de las clases.
- Ayudan a detectar que NO poner en la interfaz
- Ayudan a detectar cuando diferentes clases deben implementar las mismas interfaces

¿Cómo ayudan los Patrones a solucionar problemas de diseño?

4-Diseñar pensando en las interfaces y no en la implementación

- Manipular objetos sólo en términos de sus interfaces.

Ventajas:

- Los clientes se mantienen independientes de las clases de objetos que usan.

- Los clientes sólo conocen las interfaces que estas clases deben implementar.

Como describir un patrón de diseño?

- Formato de un patrón:
  - ***Nombre y clasificación***
  - ***Intención:*** ***explicar en terminos abstractos el objetivo del patrón***
  - ***Nombre alternativo***
  - ***Motivación :contar*** ***un problema concreto, la solución y luego la generalización***

Como describir un patrón de diseño?

- Formato de un patrón:
  - ***Aplicabilidad :*** ***reglas que dicen cuando aplicarlo.***
  - ***Estructura :*** ***esquema de objetos.***
  - ***Participantes***
  - ***Colaboraciones***

Como describir un patrón de diseño?

- Formato de un patrón:
  - ***Consecuencias :*** ***efectos adversos sobre el modelo de OO***
  - ***Implementación :*** ***un ejemplo de la solución.***
  - ***Código de ejemplo*** ***: un ejemplo en pseudocodigo***
  - ***Usos conocidos*** ***: ejemplos de aplicabilidad.***
  - ***Patrones relacionados***

Ventajas de los patrones de Diseño

- Son soluciones concretas:
  - Un catálogo de patrones es un conjunto de recetas de diseño.
  - Cada patrón es independiente del resto.
- Son soluciones técnicas:
  - Dada una situación, los patrones indican cómo resolverla.
- Se aplican en situaciones muy comunes:
  - Proceden de la experiencia y han demostrado su utilidad.

Ventajas de los patrones de Diseño

- Son soluciones simples:
  - Indican cómo resolver un problema utilizando un pequeño número de clases relacionadas de forma determinada.
  - No indican cómo diseñar un sistema completo, sino sólo aspectos puntuales del mismo.
- Facilitan la reutilización del código y del diseño:
  - Los patrones favorecen la reutilización de clases ya existentes y la programación de clases reutilizables.
  - La propia estructura del patrón se reutiliza cada vez que se aplica.

Desventajas de los patrones de Diseño

- Su uso no se refleja claramente en el código:

  A partir de la implementación es difícil determinar qué patrón de diseño se ha utilizado.

  No es posible hacer ingeniería inversa.

- Es difícil reutilizar la implementación del patrón:

  El patrón describe roles genéricos, pero en la implementación aparecen clases y métodos concretos.

Clasificación de los patrones de Diseño

	De Creación	Estructurales	De Comportamiento
Clase	Factory method	Adapter	Template Method
Objeto	Abstract Factory Builder Prototype Singleton Factory	Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento State Strategy Visitor

Patrón Observer

- **Intención:**
  - Define una dependencia uno a muchos entre objetos de modo que, cuando un objeto cambia su estado, todos sus dependientes son notificados y actualizados automáticamente.

Nombre alternativo:

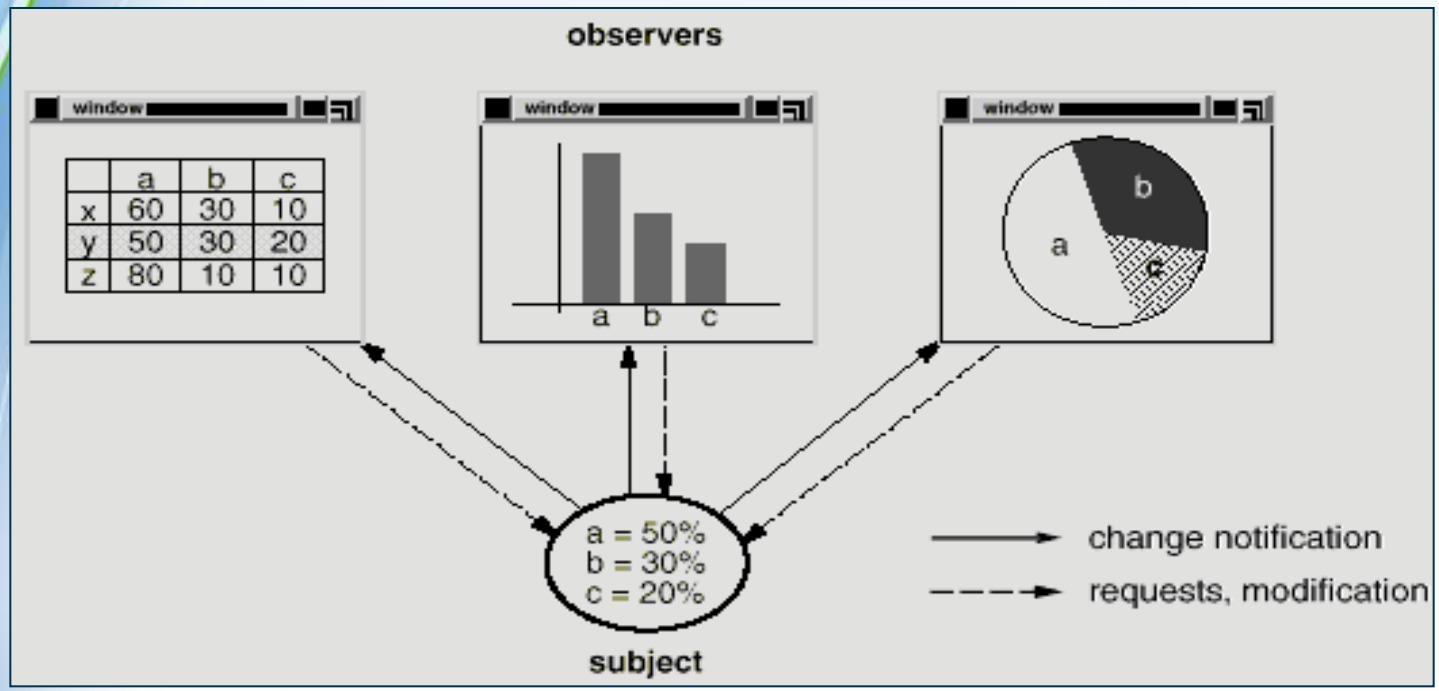
- Dependants, Publish-Subscribe.

Patrón Observer

- Motivación:
  - Un “side-effect” común al particionar el sistema en una colección de clases que cooperan para realizar cierta tarea, es la necesidad de mantener la consistencia entre estos objetos relacionados.
  - Obviamente se quiere lograr consistencia, pero no al costo de atentar contra la flexibilidad y el reuso.

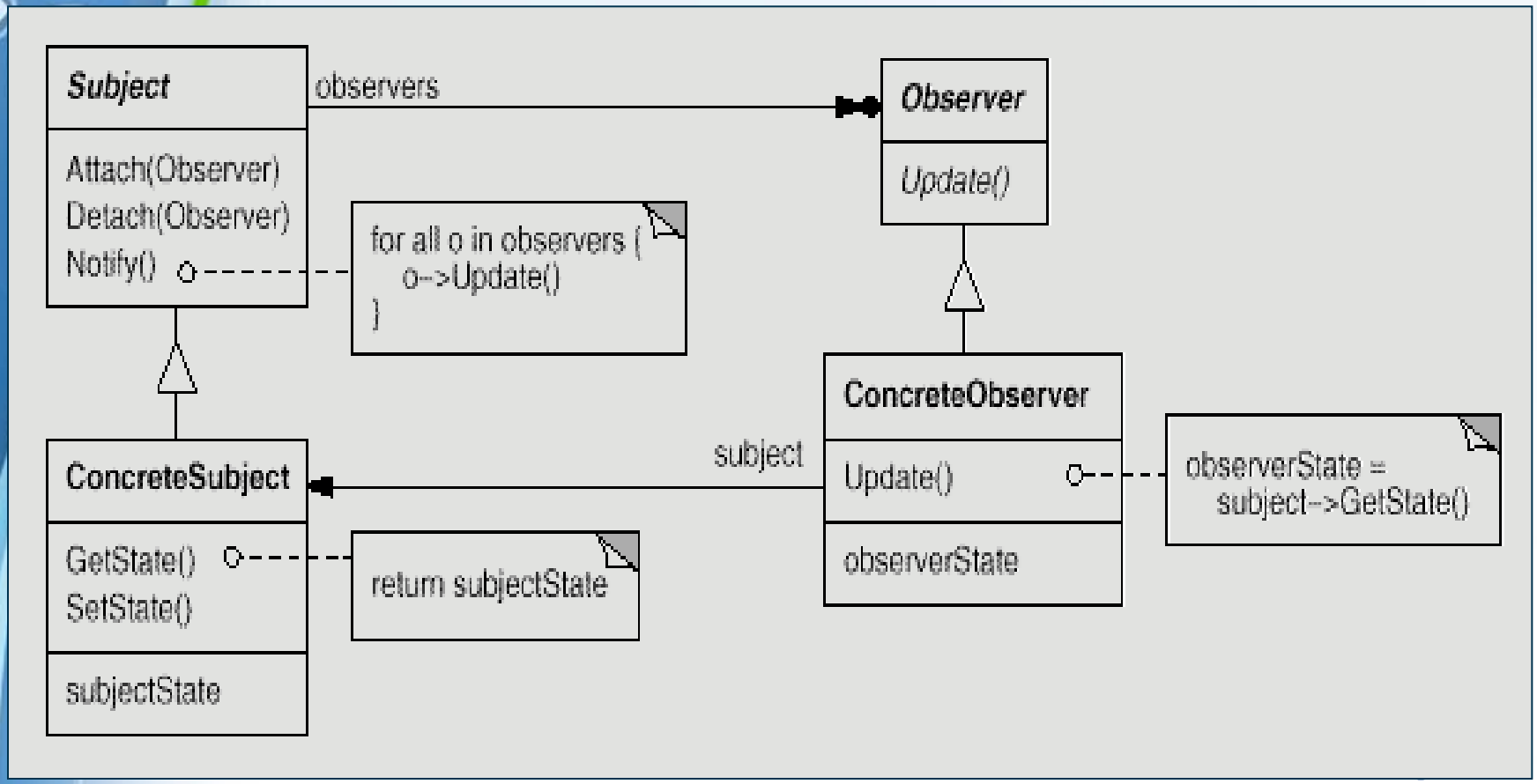
Patrón Observer

- **Motivación**
  - Las aplicaciones con interfaces de usuario son un claro ejemplo de estas situaciones.



Patrón Observer

- Estructura



Patrón Observer

- **Participantes:**
  - Subject:
    - Cualquier cantidad de observadores pueden estar interesados en un subject.
    - provee una interface para agregar y borrar dependientes.
  - Observer:
    - define una interface de actualización para aquellos objetos que deberían ser notificados de cambios.

Patrón Observer

- **Participantes**
  - ConcreteSubject:
    - lleva cuenta del “interés” de los ConcreteObserver.
    - manda una notificación de cambio cuando altera su estado.
  - ConcreteObserver:
    - mantiene una referencia a un objeto ConcreteSubject.
    - almacena cierto estado que debería ser consistente con el objeto observado.
    - implementa la interface de actualización para mantenerse consistente con su “modelo”.

Patrón Observer

- **Colaboraciones:**
  - ConcreteSubject notifica a sus dependientes cada vez que ocurre un cambio.
  - Luego de ser informado de la ocurrencia de un cambio, un ConcreteObserver puede llegar a preguntar al subject sobre su estado.

Patrón Observer

- **Consecuencias:**
  - Bajo acoplamiento entre subjects y observers → puedo hacerlos interactuar aunque pertenezcan a diferentes layers de un sistema (model - view)
  - Se pueden reusar los subjects sin necesidad de reusar los observers
  - Puede agregar observers en run-time sin necesidad de modificar ni el subject ni los otros observers (+)
  - Depende la implementación, puede haber actualizaciones no esperadas. (-)

Patrón Observer

- **Aplicabilidad:**
  - Usar este patrón en cualquier de estas situaciones:
    - Cuando una abstracción tiene dos aspectos, uno dependiente del otro.
    - Cuando el cambio en un objeto requiere la actualización de otros, sin saber cuantos son.
    - Cuando un objeto debería ser capaz de notificar a otros objetos sobre su cambio, sin necesidad de conocer las características de esos objetos

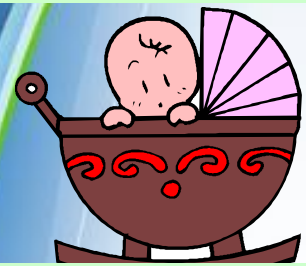
Patrón Observer

- **Patrones relacionados:**
  - Mediator: puede utilizarse para encapsular la lógica de actualización.
  - Singleton: puede utilizarse para manejar los sujetos.

Observer, Ej de un caso real

Interfaces y Observable

```
// Interfaces
public interface Observer {
    public void update();
}

public interface Observable {
    public void addObserver();
    public void removeObserver();
    public void notify();
}
```



```
// Estados posibles
        - Dormir
        - Llorar
        - Warning en pañal
        - ...
```

```
// Clase Sobrino
public class Sobrino implements Observable {
    private Vector _victimas;
    private int _estado;

    public void addObserver(Observer victima) {
        _victimas.add(victima);
    }

    public void removeObserver(Observer victima) {
        _victimas.remove(victima);

    public void notify() {
        ListIterator li=_victimas.listIterator();
        while (li.hasNext())
            li.next().update(this);
    }

    public int getState() { return _estado; }

    public void setState(int estado) {
       _estado=estado;
       this.notify();    }
}
```

Observers

```
// Clase Madre
public class Madre extends Santidad implements Observer
{
    private Vector _laAlegriaDeLaCasa;
    ....
    public void laFamiliaCrece (Observable otro) {
       _laAlegriaDeLaCasa.add(otro);
       otro.addObserver(this);
    }

    public void update(Observable terminator) {
       int estado=terminator.getState();
       switch (estado) {
              case DORMIR:
                     ...
       }
    }
    ....
}

// Clase Padre (muy similar)
   public void update(Observable terminator) {
       int estado=terminator.getState();
       switch (estado) {
              case WARNING_PAÑAL: noop;...
       }
    }
```

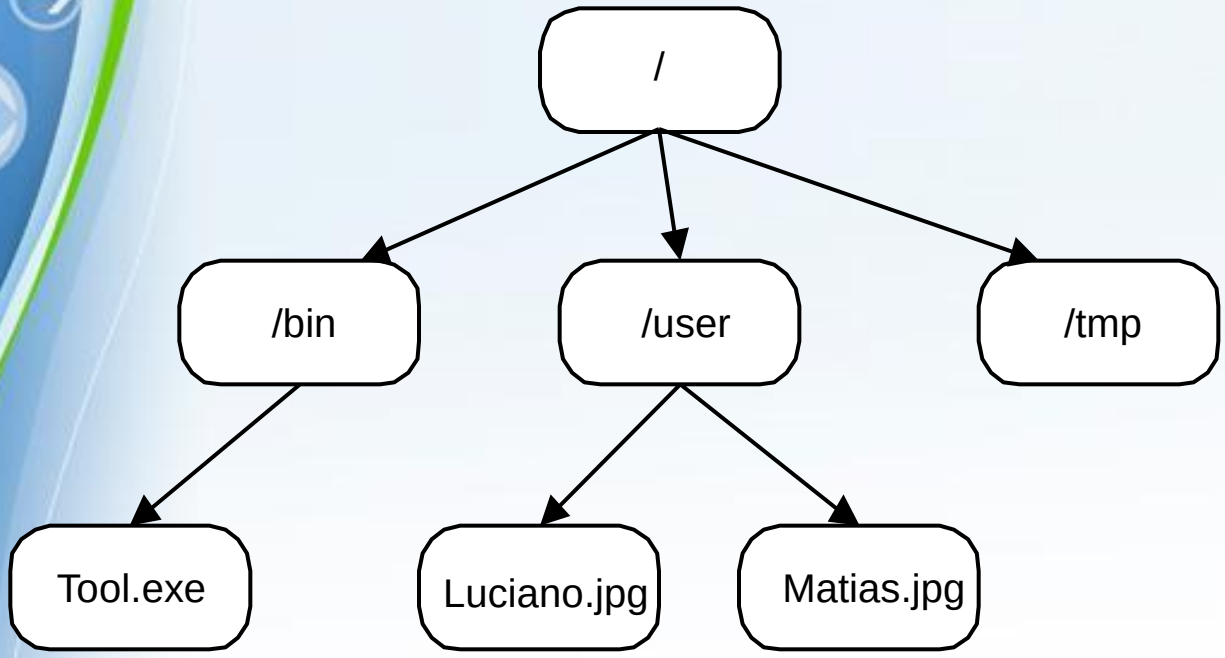
Observers

```
// Clase Tio
public class Tio implements Observer
{
    private Vector _sobrinos;
    ....
    public void unaVisita (Vector laFamilia) {
       ListIterator li=laFamilia.listIterator();
       Object o;
       while(li.hasNext())
           if ((o=li.next()) instanceof Sobrino) {
              o.addObserver(this);
              _sobrinos.add(o);
            }
    }

    public void hastaOtra () {
       // Recorrer vector
            _sobrinos.removeObserver(this);
       _sobrinos.removeAll();
    }

    public void update(Observable terminator) {
       int estado=terminator.getState();
       switch (estado) {
              case GATEAR: // Alerta general
    }                 ...
}
```

Patrón composite. Ejemplo

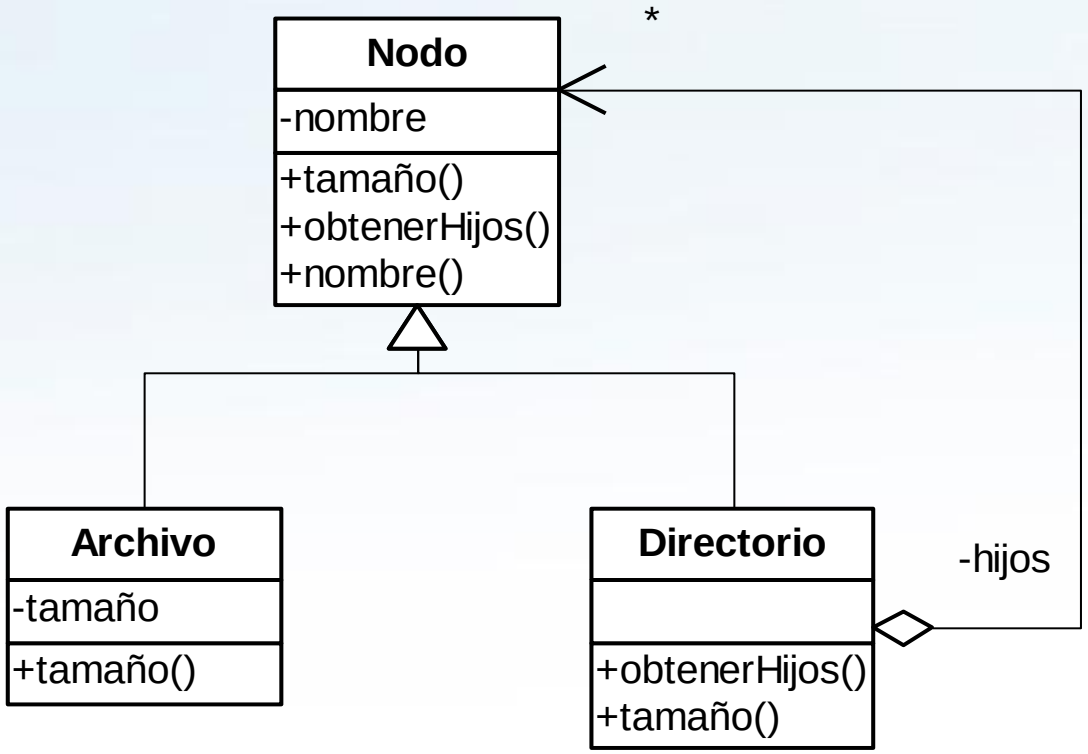
- Supongamos que tenemos que implementar un file system simplificado
- Cada componente puede tener varios atributos de interés en común como
  - Tamaño (Kb)
  - Nombre
  - Protección o permisos
- Cada atributo puede tener operaciones para accederlos y modificarlos

Patrón Composite Ejemplo



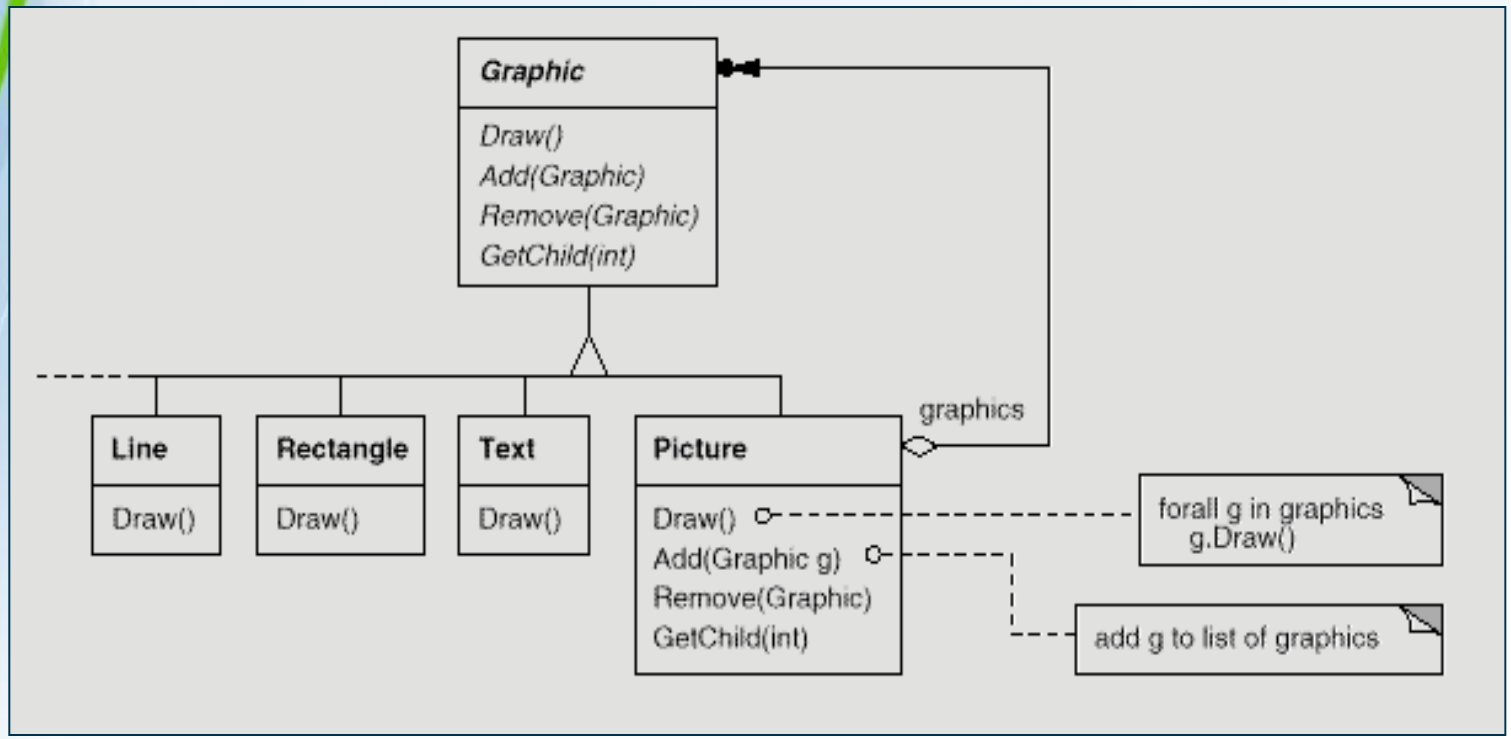
Patrón Composite

- ¿Estructura?
- ¿Operaciones?



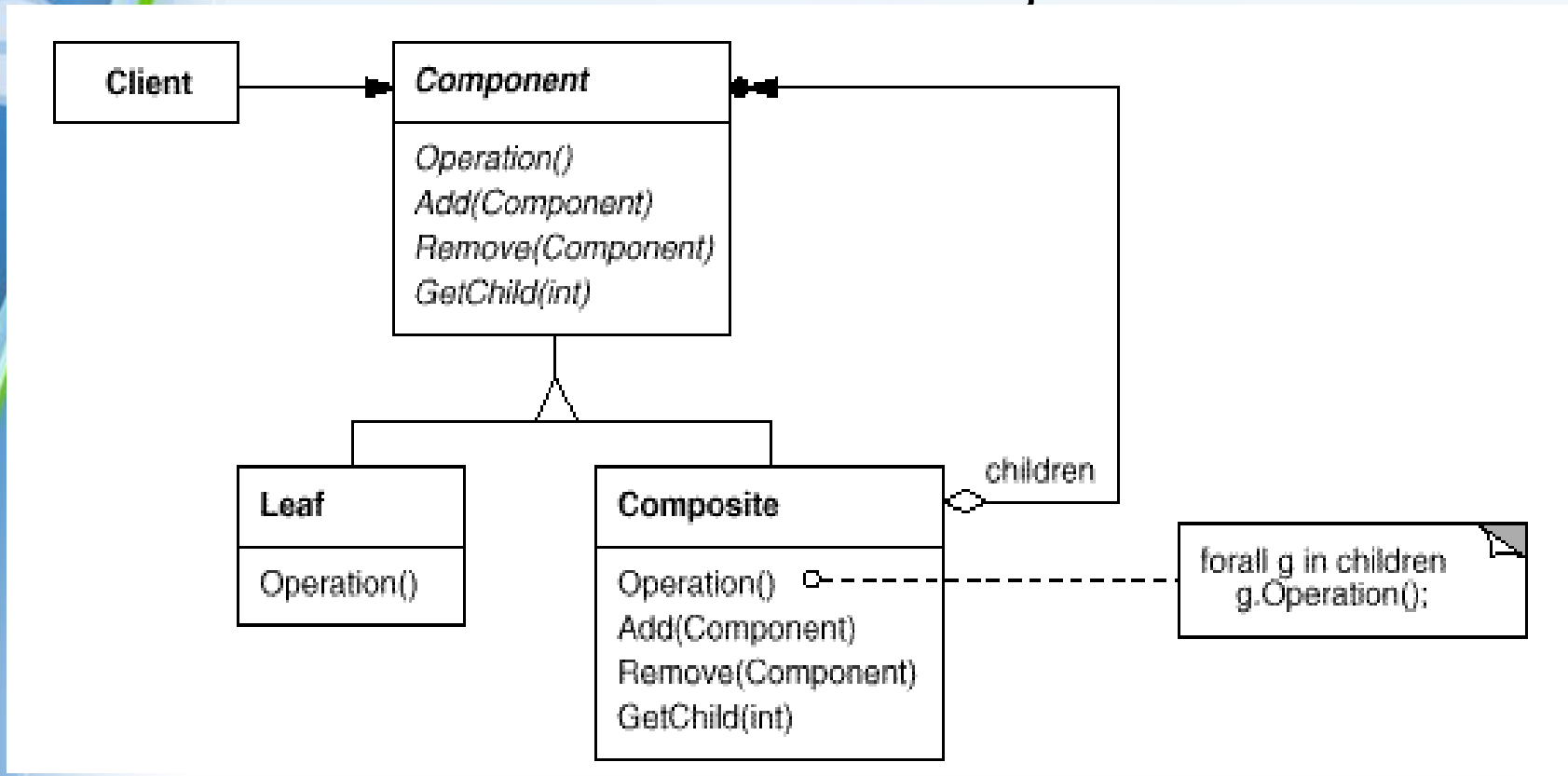
Patrón Composite

- Motivación:



Patrón Composite

- Permite a los “clientes” tratar uniformemente tanto a objetos individuales



Patrón Composite

- Implementación:
  - La clase Component debe tener TODAS las operaciones que comparten las hojas y los composite.
  - Para ganar transparencia (pero perdiendo seguridad) en el cliente, se podrían definir operaciones en component que no son de leaf (en Java se deberían manejar con excepciones)

Patrón Composite

- Consecuencias:
  - El cliente no necesita diferenciar el tratamiento de objetos simples y objetos composites.(+)
  - Hace a los clientes más simples. (+)
  - El cliente no se toca al agregar nuevas componentes.(+)

Ejemplo

- Supongamos un archivo de configuración, que tiene información utilizada por varias clases del sistema.
- Como implementamos este esquema de lectura de esta información de configuración?

Ejemplo : Solución 1 (impensable)

- Hay un método en cada clase que necesite datos de configuración, que se encarga de la lectura cada vez que necesite un dato de configuración.

- Desventajas:
  - Se accede a disco cada vez que se necesita un dato de configuración.
  - El cambio del nombre del archivo implica cambiar N clases

Ejemplo: Solución 2 (un poco mejor)

- Centralicemos el acceso en una sola clase (Config)
- Los clientes instanciarán esta clase y se la guardarán para que cada vez que necesiten un dato de configuración se lo puedan pedir a esta instancia.
- En el constructor de la clase Config estará el codigo de deserialización del archivo en objetos. Se lee el archivo y se carga en memoria para no tener que volver a acceder a disco.

Ejemplo: Solución 2 (un poco mejor)

- **Desventajas:**
  - Por cada clase que necesite datos de configuración, existirá una instancia de Config con el mismo estado.
  - Se lee el archivo N veces.

Ejemplo: Solución 3

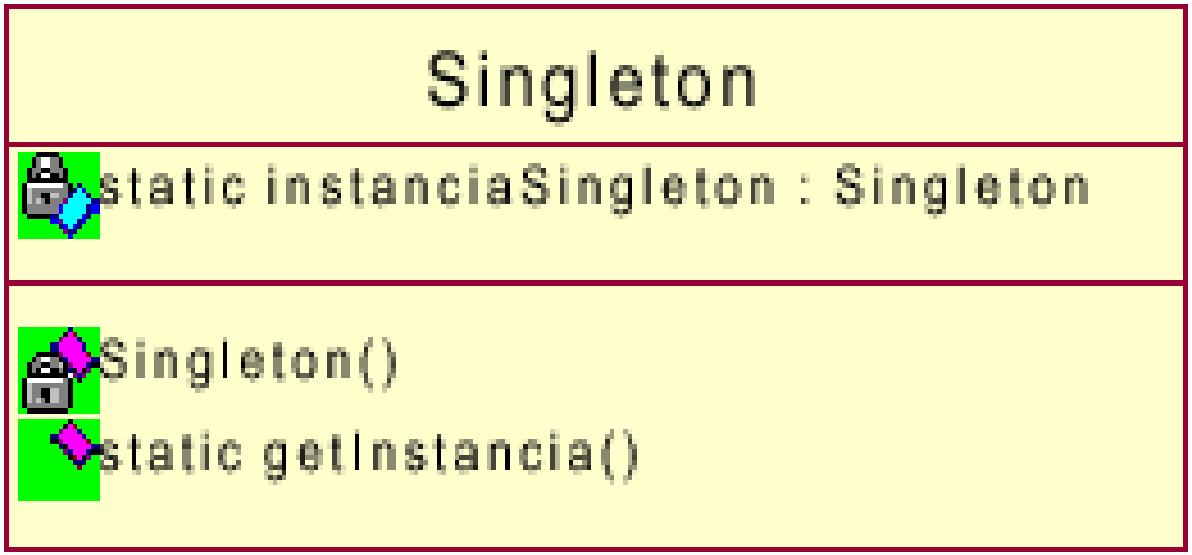
- Que existe una sola instancia de Config y todos usen la misma.
- Así conseguimos que el archivo se lea una sola vez.
- Como nos aseguramos que solo habrá una instancia?
  - Debemos controlar el intento de instanciación de estos objetos.
  - Que pasa con los Constructores?

Singletón

- **Objetivo**
  - Garantiza que solamente se crea una instancia de la clase y provee un punto de acceso global a él. Todos los objetos que utilizan una instancia de esa clase usan la misma instancia.

Singletón

- **Estructura**



Singletón

- Consecuencias
  - Es raro pensar en una subclase de una clase singleton.
  - Si se querría no es simple lograrlo
    - Para lograr el objetivo del Singleton, no deje ningún constructor público ni por defecto.
    - Puedo hacer una subclase si no tengo un constructor público?
    - Tengo que hacer un constructor público (puede violarse el Singleton)
    - Si además se quiere que sea Singleton querré redefinir el método de clase getInstance. Java no lo permite

Singletón

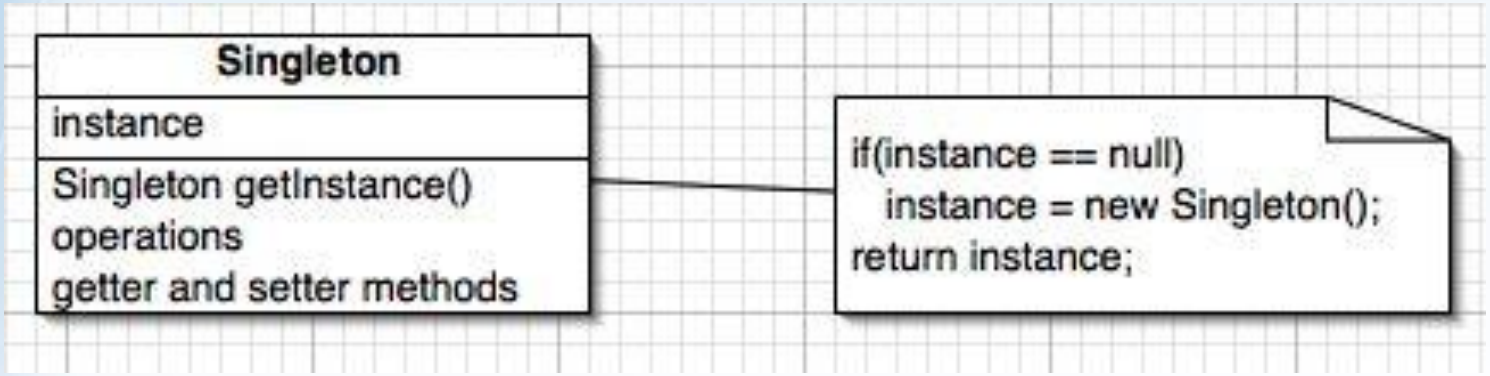
Usos en la API Java:

Clase java.lang.Runtime

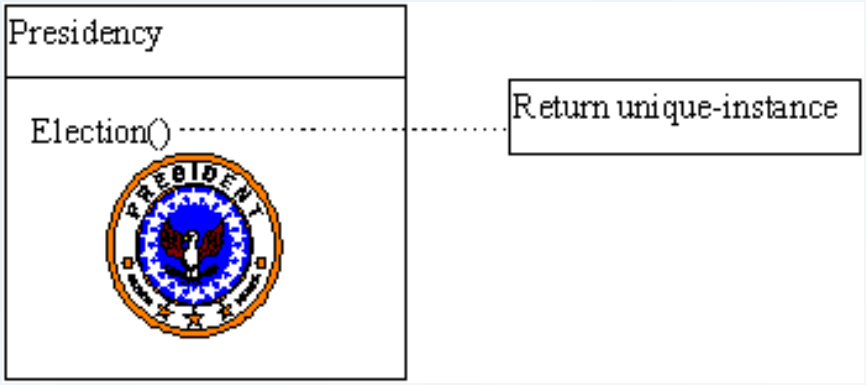
Ejemplos:

- contadores para asignar identificadores.
- controladores de colas de impresión.
- clases que representen tipos univaluados.

Singleton


Singleton
instance
Singleton getInstance()
operations
getter and setter methods
if(instance == null)
instance = new Singleton();
return instance;

Singleton



Ejemplos donde se desee tener una única instancia y un punto global de acceso:

- DAO :data access object., genero una única instancia para acceder a la base de datos.
- Login de usuario.
- Puerto de comunicación.
- Etc, Se usa mucho aunque no lo percibamos.

Ejemplo

Supongamos que tenemos un editor que permite embeber lineas y puntos de diferentes ambientes de diseño (Visio, Rational, Poseidon, etc.)

Sigamos agregando elementos al escenario

- Se pueden dar las siguientes situaciones
  - Hay un objeto seleccionado
  - Hay varios objetos seleccionados
  - No hay selección
  - Se esta moviendo un elemento gráfico
  - …
- En base a esto, las operaciones modeladas anteriormente tendrán un efecto distinto
  - Si no hay un objeto seleccionado, el borrado no tiene efecto
  - Si hay muchos objetos seleccionados, la operación borrar los elimina

Ejemplo

- ¿Solución?
  - Jerarquía de elementos seleccionados, no seleccionados, en movimiento, draggeando (aaahhh.)
    - PuntoSeleccionado
    - PuntoEnMovimiento
    - PuntoNoSeleccionado
    - PuntoDraggeando
    - LineaSeleccionada
    - LineaEnMovimiento
    - LineaNoSeleccionada
    - LineaDraggeando
  - Cuando un punto pasa de seleccionado a no seleccionado?

Ejemplo

- Poner una variable que me indica que valor tiene, y cada vez que realizo una operación hago el super-case correspondiente.
- Delegar? Plantear un esquema de estados?

State

- Objetivo:
  - Encapsula los estados de un objeto como objetos separados
  - Permite a un objeto cambiar su comportamiento cuando su estado interno cambia. El objeto parecerá haber cambiado de clase.

State

Se utiliza cuando se desea tener una clase que puede cambiar su estado. La clase incluye a otra que determina su estado y el mismo puede cambiar.

La clase envía los mensajes a su estado y de acuerdo al mismo la funcionalidad varía. Nótese que la responsabilidad sigue siendo de la clase original y no del estado.

State sugiere cambiar entre las clases internas de una manera tal que el objeto simule cambiar su estado.

Este patrón es utilizado para evitar el if-then-else o switch.

State

Es un patrón de comportamiento para objetos.

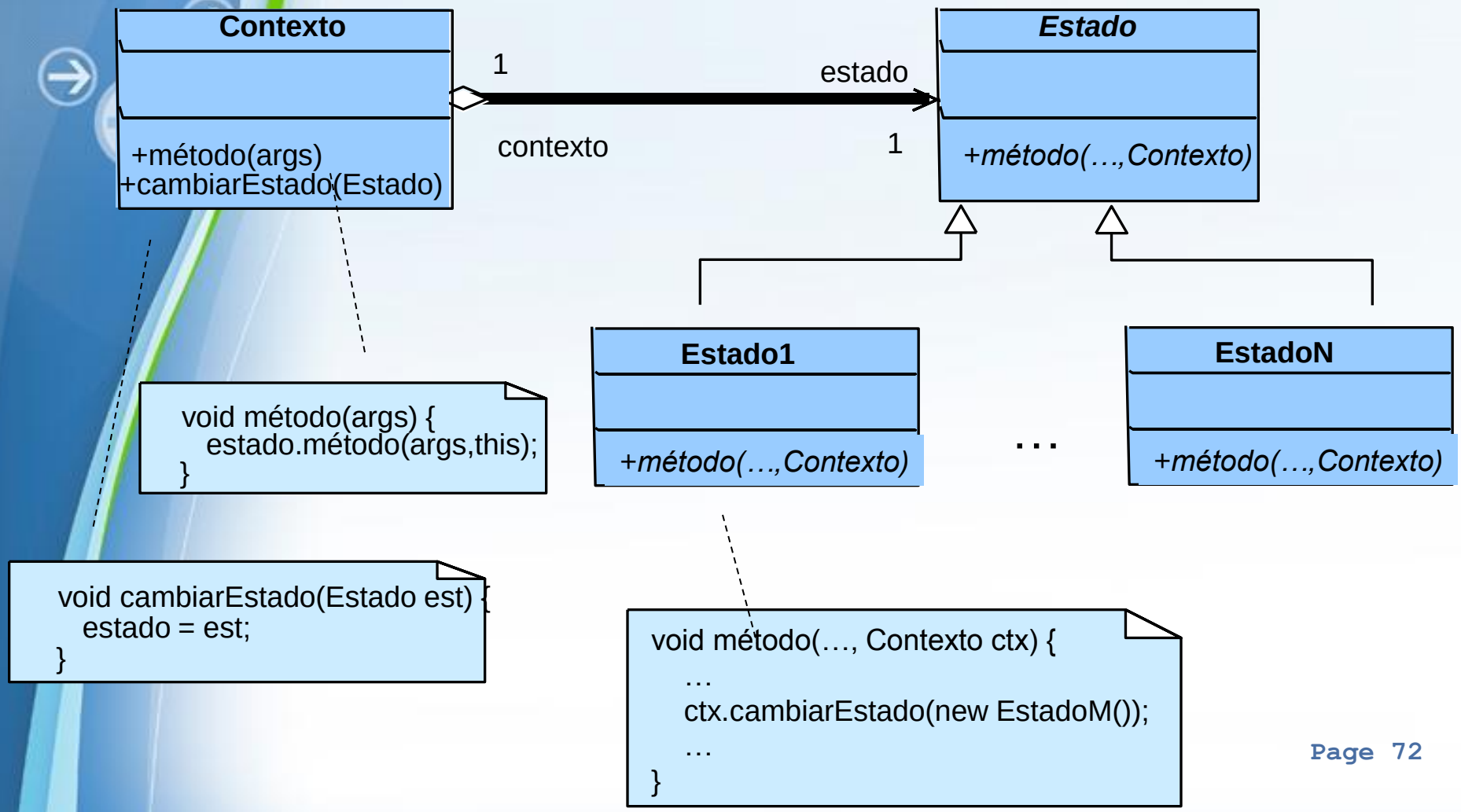
Se debe usar si el comportamiento de un objeto

- depende de su estado, y
- se ve modificado en tiempo de ejecución.

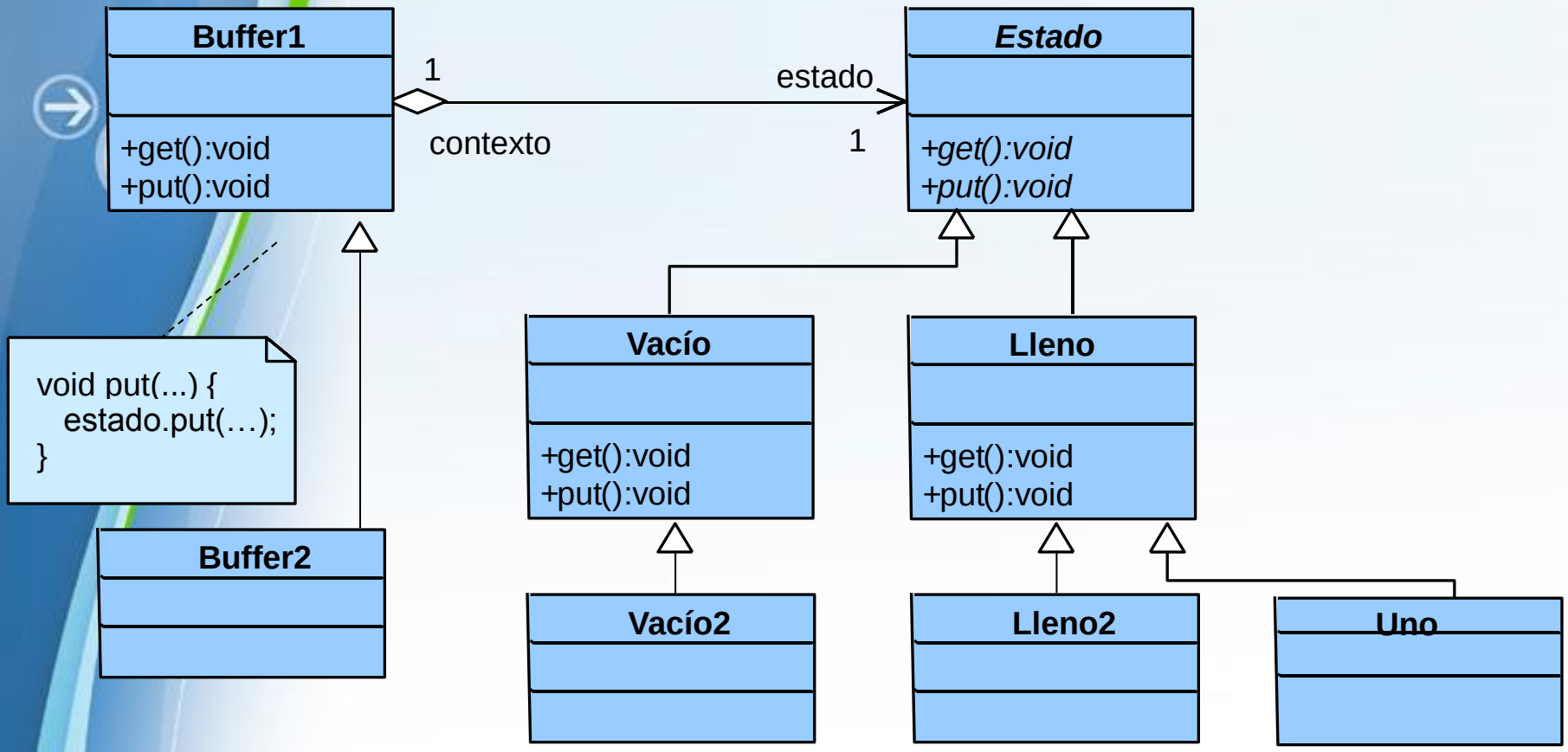
Permite que un objeto cambie su comportamiento

- cuando cambia su estado interno,
- tal y como si el objeto cambiase de clase.

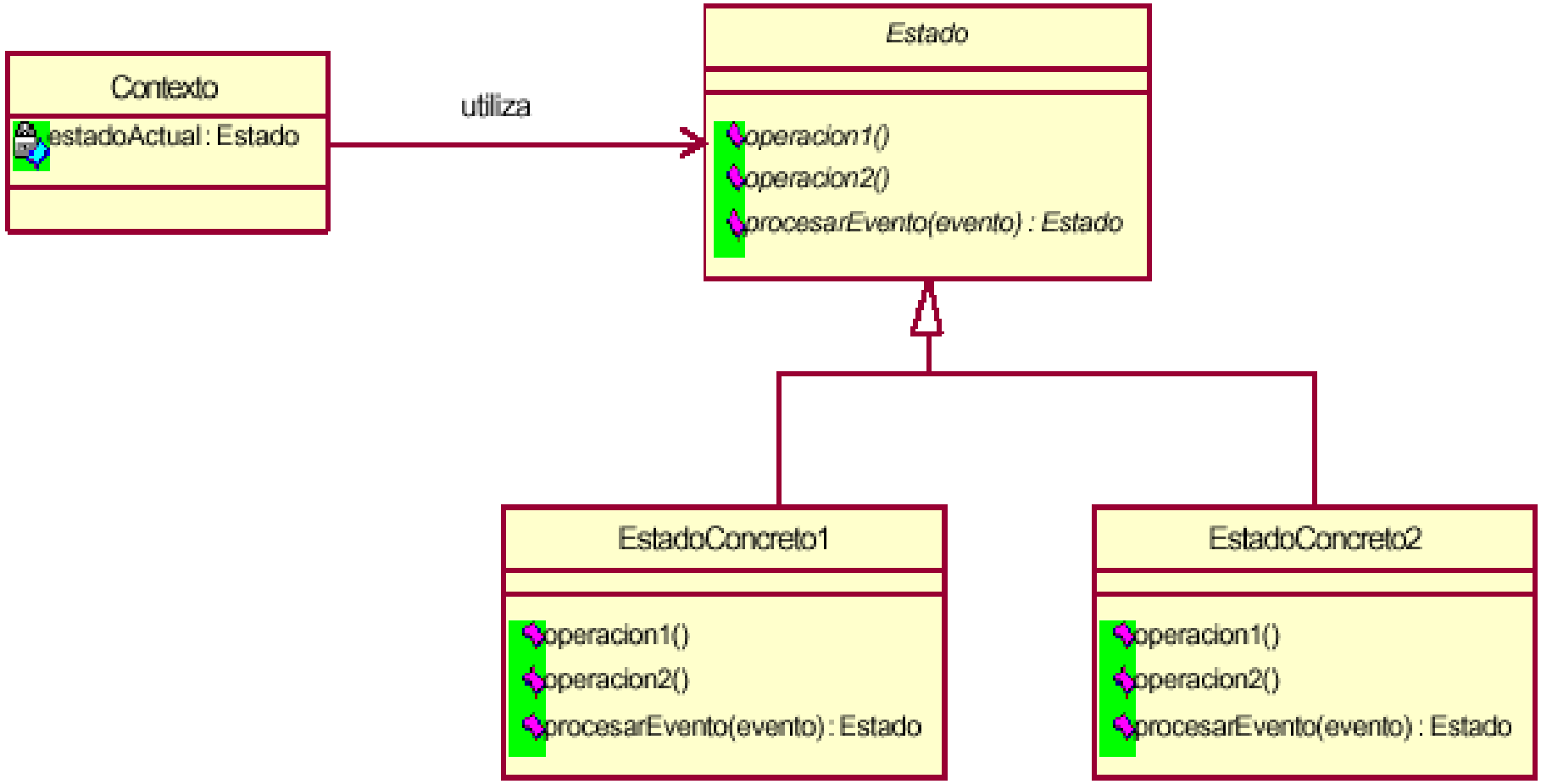
State : solución



State : ejemplo de aplicación

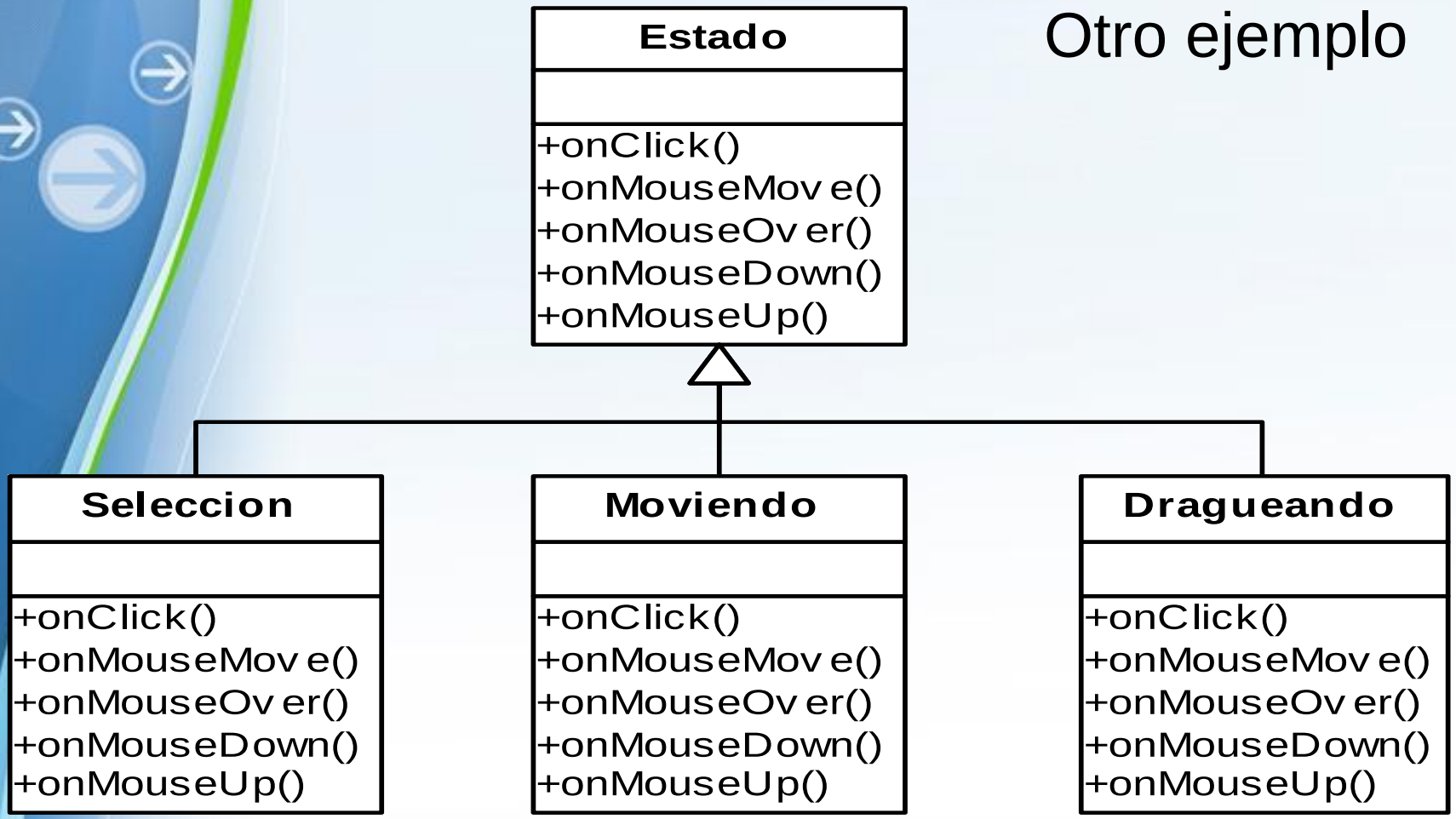


State - Estructura



State

Otro ejemplo



State

Otro ejemplo, Maquina expendedora