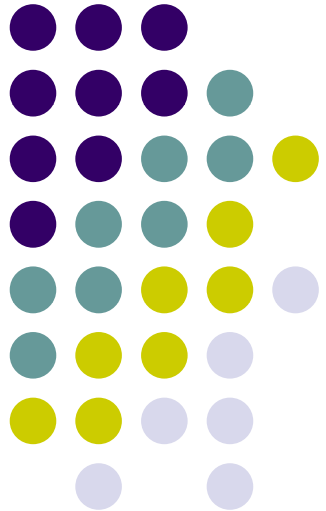
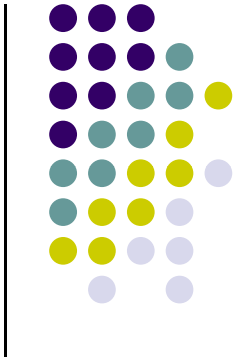
Metodologías de Diseño y Programación

Diseño

Patrones de Diseño

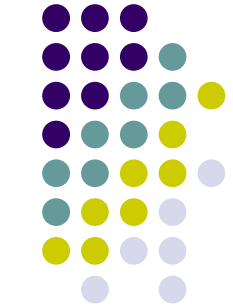


Contenido



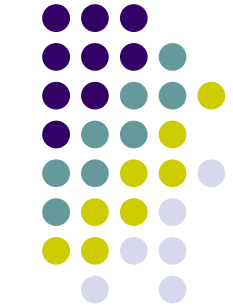
- Introducción
- Patrones de Diseño
- Singleton
- State
- Observer
- Strategy

Introducción



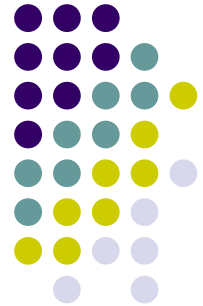
- Los diseñadores expertos afirman que es casi imposible lograr un diseño flexible y reusable en el primer intento
- Los expertos logran buenos diseños mientras que los principiantes se ven abrumados por todas las opciones disponibles
- Se requiere de mucho tiempo para que los principiantes aprendan de qué se trata un buen diseño
- Evidentemente los expertos tienen un conocimiento que los principiantes no

Introducción (2)



- Algo que los expertos saben es que no deben resolver un mismo problema de una forma diferente cada vez
- En cambio, reutilizan soluciones que les dieron buenos resultados en el pasado
- Cuando encuentran una buena solución a un problema la usan una y otra vez cada vez que el mismo problema se les presenta
- Esa experiencia es lo que los convierte en expertos

Patrones

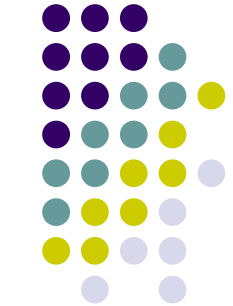


patrón *sust* **1** modo usual según el cual algo ocurre, se desarrolla o es hecho **2** cosa o forma que representa un ejemplo a copiar

Patrones (2)

- Al desarrollar un sistema orientado a objetos un diseñador enfrenta una serie de problemas a resolver
- Muchos de ellos aparecerán nuevamente en los siguientes proyectos independientemente del dominio de cada uno de ellos
- Se detectan entonces problemas de naturaleza muy similar entre sí
  - Que aparecen recurrentemente en el diseño de aplicaciones de diversos tipos (patrones)

Patrones (3)



- Los diseñadores reutilizan diseños exitosos basando nuevos diseños en experiencias anteriores
- Un diseñador que esté familiarizado con esos patrones puede aplicarlo directamente a problemas de diseño sin tener que redescubrirlos

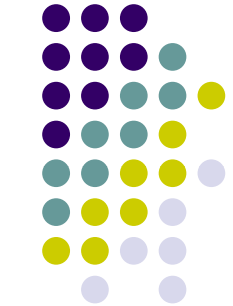
Patrones (4)

- Si fuera posible recordar los detalles de un problema atacado en el pasado y la forma en que este fue solucionado, se podría reusar esa experiencia en lugar de redescubrirla
- El propósito de los Patrones de Diseño es registrar esa experiencia para que otros la aprovechen
  - Ejemplo: Fábricas, Façades, ...

Patrones de Diseño

- Un Patrón de Diseño sistemáticamente da un nombre, motiva y explica un diseño general que se aplica a un problema de diseño recurrente en sistemas orientados a objetos
- Describe el problema, la solución, cuándo aplicar la solución y sus consecuencias

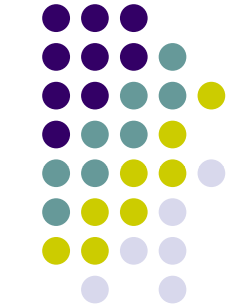
Patrones de Diseño (2)



- Además provee guías para su implementación y ejemplos
- La solución es un arreglo general de objetos y clases que solucionan el problema
- La solución concreta es adaptada e implementada a partir de ello para resolver el problema en un contexto particular

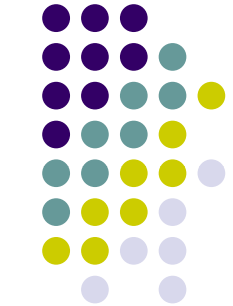
Patrones de Diseño

Colaboración Abstracta

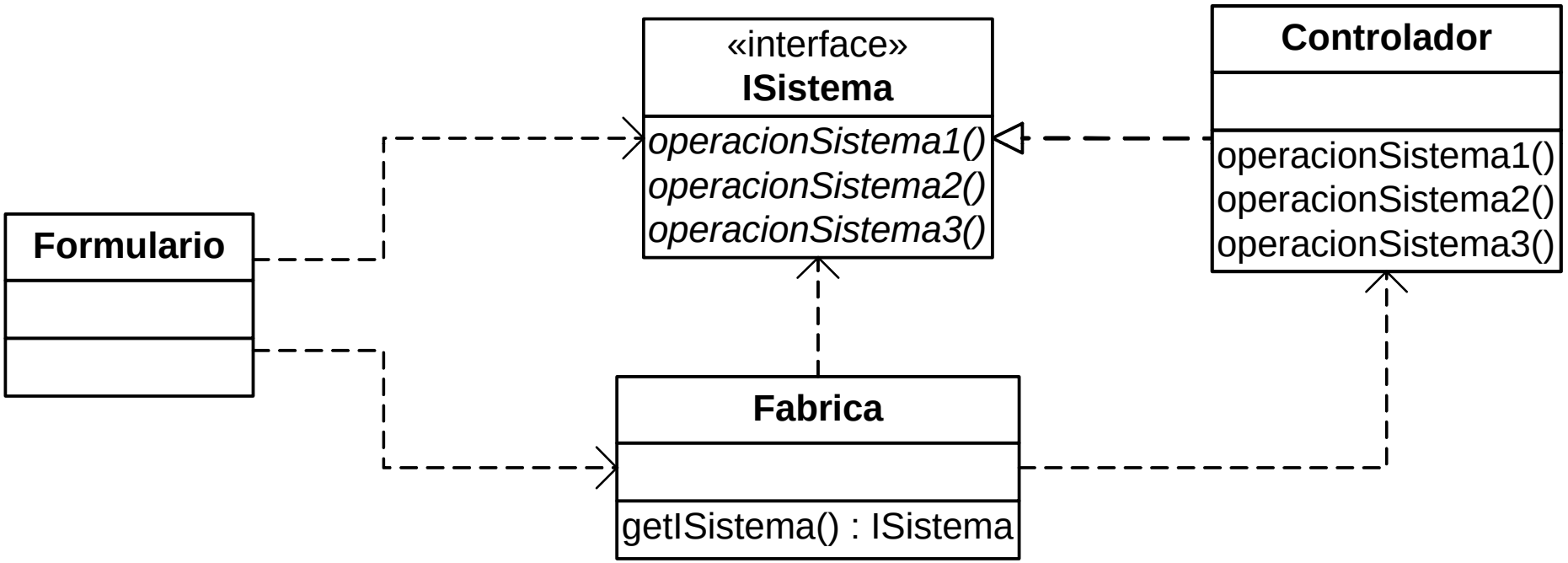


- La solución propuesta por un Patrón de Diseño es una colaboración abstracta (o paramétrica) que resuelve un problema tipo
- Dicha colaboración es por lo tanto interpretada como una “plantilla de colaboración”
- Si el problema concreto a resolver es compatible con el problema tipo se genera una colaboración concreta a partir de la plantilla
- Dicha colaboración será prácticamente idéntica a la plantilla y resolverá el problema concreto

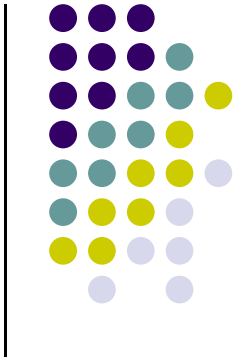
Patrones de Diseño
Ejemplo



- El mecanismo de Fábricas utilizado para comunicar el Formulario con el Controlador no es aplicable exclusivamente a este problema concreto



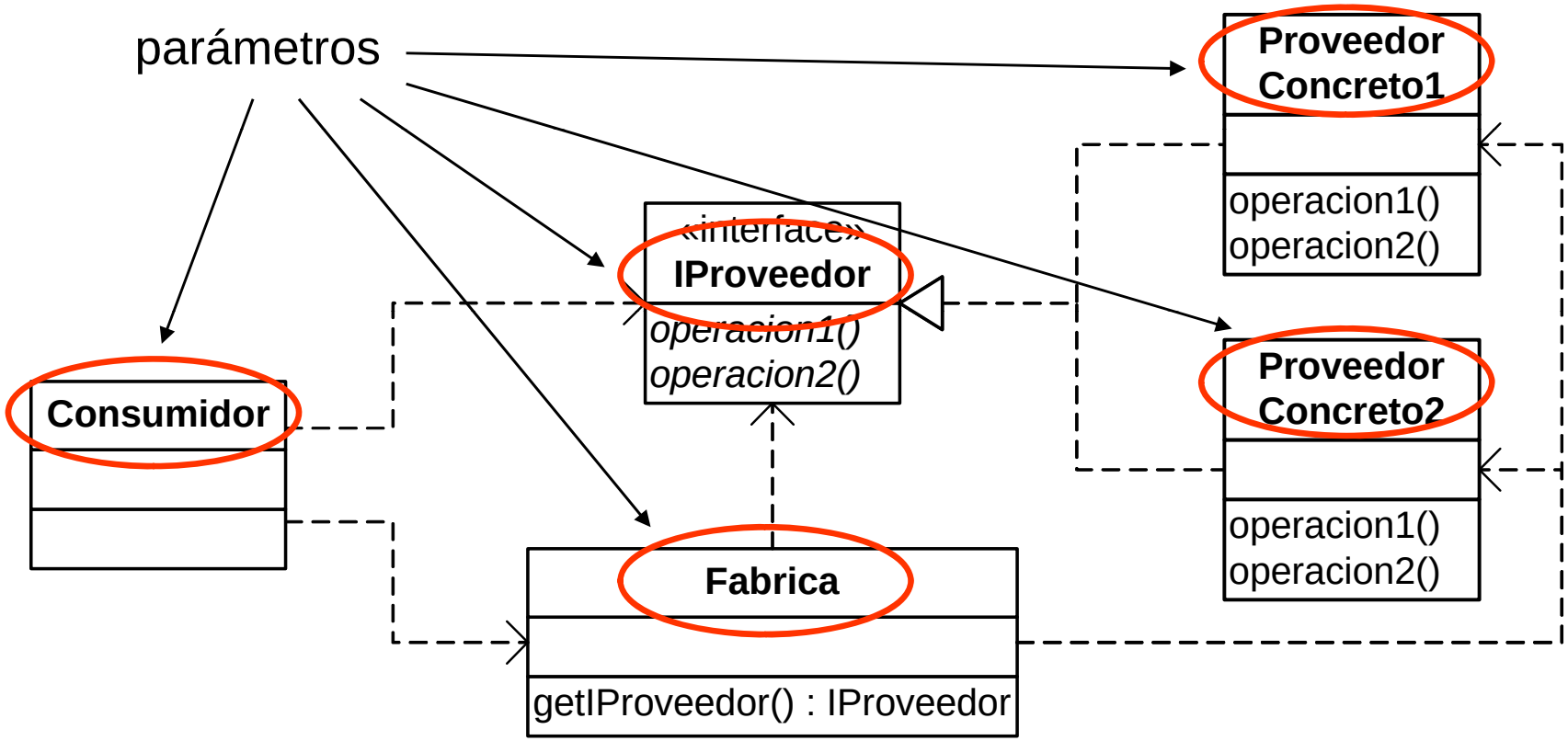
Ejemplo (2)



- Tampoco es aplicable exclusivamente para "comunicar un elemento de la Capa de Presentación con un Controlador"
- El problema tipo que el mecanismo de Fábricas permite resolver es
  - "Permitir visibilidad desde un consumidor hacia proveedores concretos sin que el consumidor quede acoplado directamente a éstos"

Patrones de Diseño
Ejemplo (3)

- La estructura de una colaboración que solucione el problema tipo puede ser:

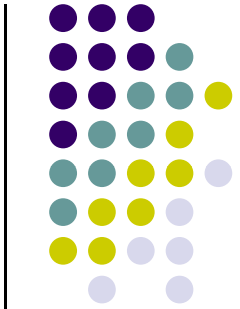


Ejemplo (4)

- Para el problema concreto determinamos las siguientes correspondencias entre clases del diseño y parámetros de la colaboración

Clase	Parámetro
Formulario	Consumidor
Fabrica	Fabrica
ISistema	IProveedor
Controlador	ProveedorConcreto1

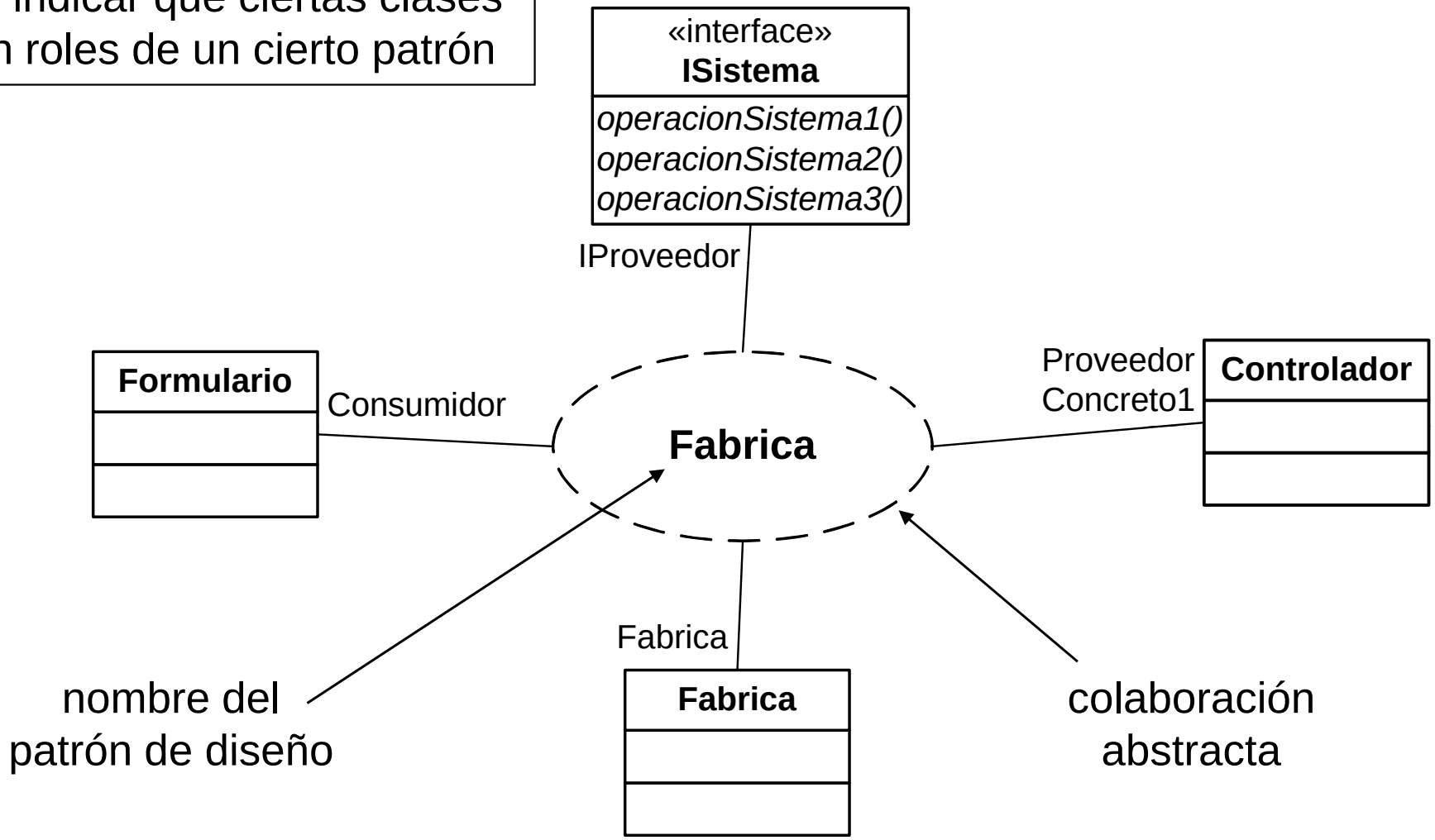
Ejemplo (5)



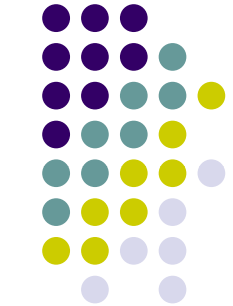
- Para explicitar la correspondencia definida es posible embeber la estructura propuesta por el patrón en el diseño de la solución (reemplazando los nombres de parámetros por los de las clases correspondientes)
- Esto conduce a un resultado como el mostrado originalmente
- En general la estructura resultante es idéntica a la del patrón por lo que aumenta la complejidad del diseño final

Patrones de Diseño
Ejemplo (6)

Alternativa: indicar que ciertas clases desempeñan roles de un cierto patrón

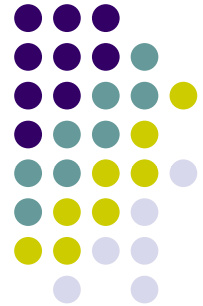


Descripción



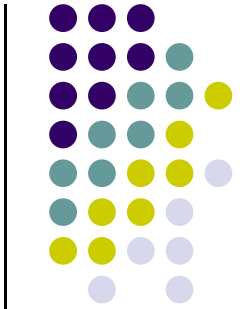
- La descripción de un Patrón de Diseño está organizada de la siguiente manera
  - **Nombre:**
    - Se utiliza para referenciar a un problema, sus soluciones y consecuencias en una o dos palabras
    - Asignarle un nombre a un patrón incrementa el vocabulario de diseño permitiendo diseñar en un nivel mayor de abstracción
  - **Problema:** describe cuándo utilizar el patrón, explica el problema tipo y su contexto

Descripción (2)



- Descripción (cont.)
  - **Estructura:** diagrama de clases que ilustra la estructura de la colaboración abstracta que soluciona el problema tipo
  - **Participantes:** descripción de las clases que forman parte de la estructura y sus responsabilidades
  - **Interacciones:** diagramas de interacción que ilustran el funcionamiento de la colaboración abstracta

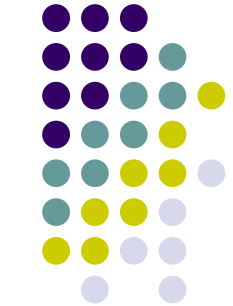
Descripción (3)



- Descripción (cont.)
  - **Consecuencias:** Comentarios, discusiones, sugerencias y advertencias que permitan entender e implementar el patrón
- Nota
  - La estructura e interacciones conforman la colaboración abstracta que soluciona el problema tipo

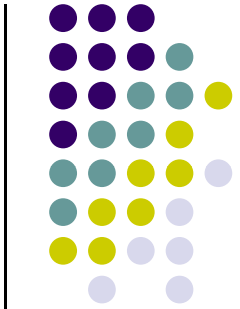
Patrones de Diseño

Sugerencia



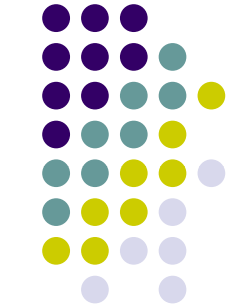
- Comprender el detalle de la colaboración propuesta por un patrón es fundamental para su aplicación
- Tan importante como esto es comprender
  - El problema tipo
  - El contexto de aplicabilidad de un patrón
- Es de muy poca utilidad conocer la solución a un problema desconocido o que no sabemos reconocer

Sugerencia (2)



- Conocer las consecuencias de la aplicación de un patrón es también fundamental
- Aplicar la solución abstracta propuesta a un problema concreto puede no ser siempre adecuado
- Esto depende de las características particulares del problema concreto
  - Por ejemplo: En ocasiones la cantidad de clases puede quedar demasiado grande

Singleton

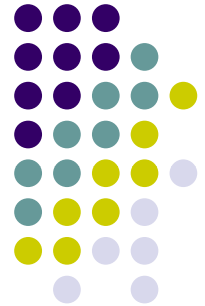


- Problema Tipo:

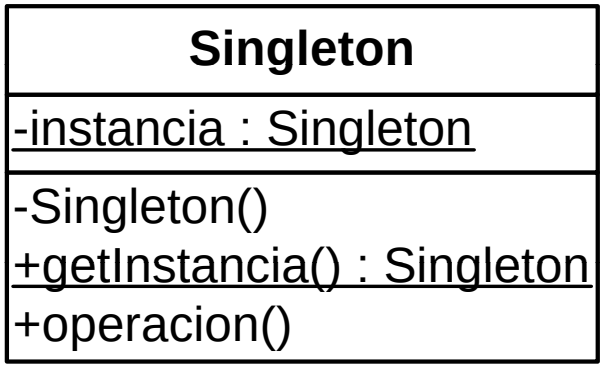
> "Asegurar que una clase tenga una sola instancia y proveer un acceso global a ella"

- Aplicabilidad
  - Debe existir una única instancia de una clase
  - Dicha instancia debe ser accesible por cualquier cliente de la clase

Singleton (2)



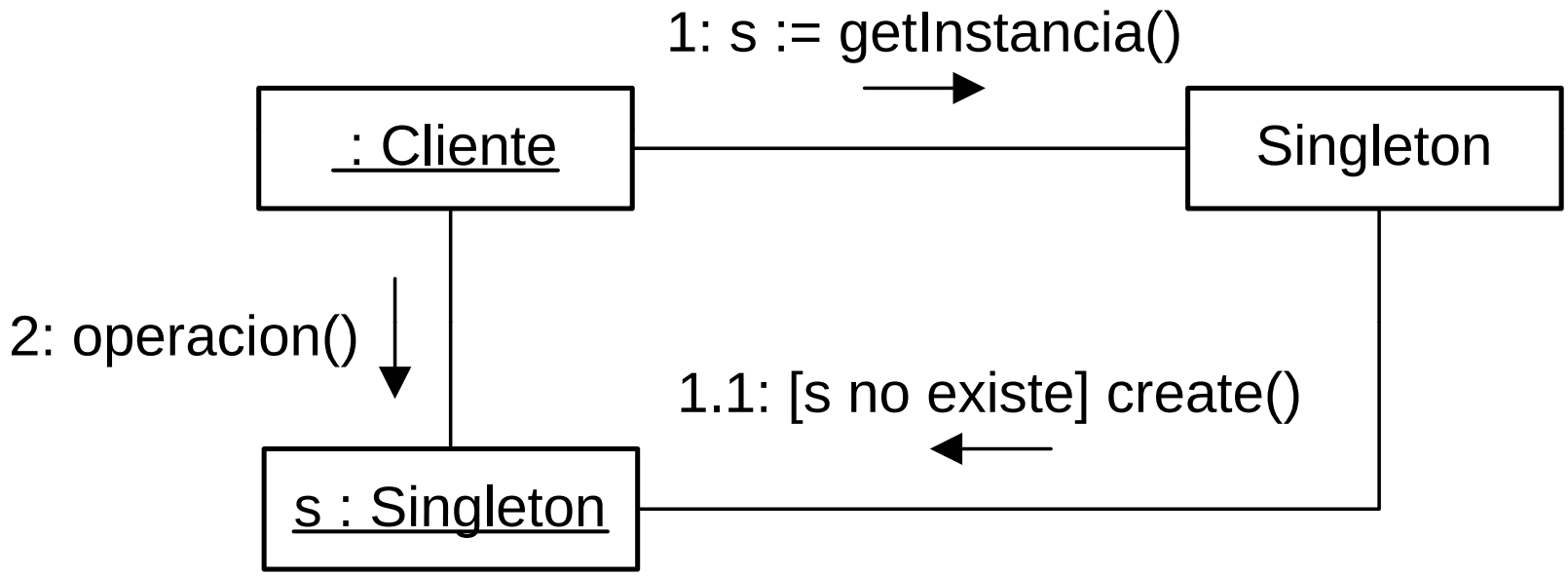
- Estructura



- Participantes
  - **Singleton:** provee una operación de clase (`getInstancia()`) que permite acceder a la única instancia

Singleton (3)

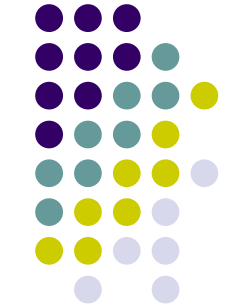
- Interacciones



Singleton (4)

- Consecuencias
  - Se provee acceso controlado a una única instancia
  - Se permiten variantes en las que se varíe el número máximo de instancias
  - Un cliente puede liberar la memoria de la instancia
  - Derivar una clase Singleton resulta complejo
- Aplicaciones
  - Las fábricas suelen ser singletons
  - Algunos controladores también

Singleton (5)



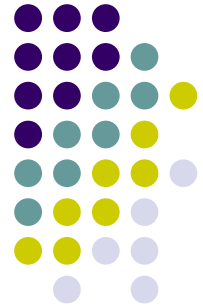
```
class Singleton {
   static private Singleton * instancia = NULL;

   private Singleton() { ... }

   static public Singleton getInstancia() {
      if (instancia == null)
         instancia = new Singleton();
      return instancia;
   }

   public void operación() { ... }
}
```

Singleton (6)



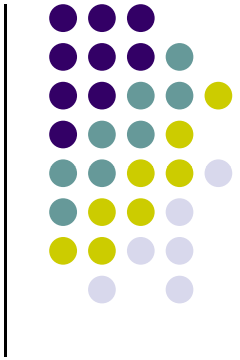
- Ejemplo de uso

```
void main() {
   Singleton * s;

   s = Singleton::getInstancia();

   s->operacion();
}
```

State



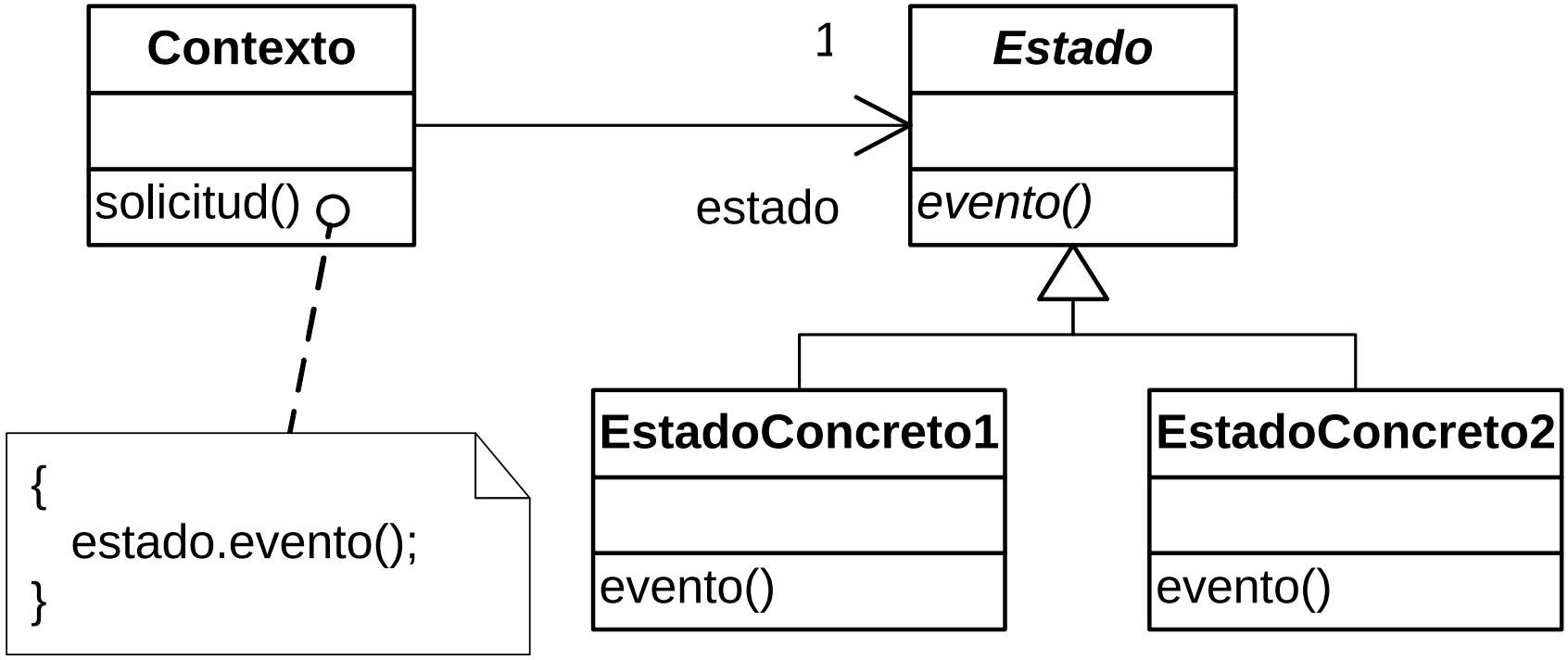
- Problema Tipo:

> "Permitir que un objeto varíe su comportamiento cuando su estado interno cambie. El objeto parecerá haber cambiado de clase"

- Aplicabilidad
  - El comportamiento de un objeto depende de su estado y debe cambiar su comportamiento en tiempo de ejecución dependiendo de ese estado
  - Las operaciones de un objeto tienen fragmentos condicionales dependientes de su estado

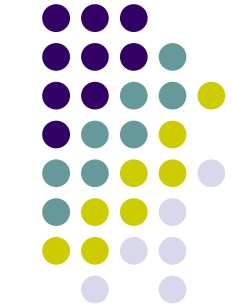
State (2)

- Estructura



No se incluyen las dependencias

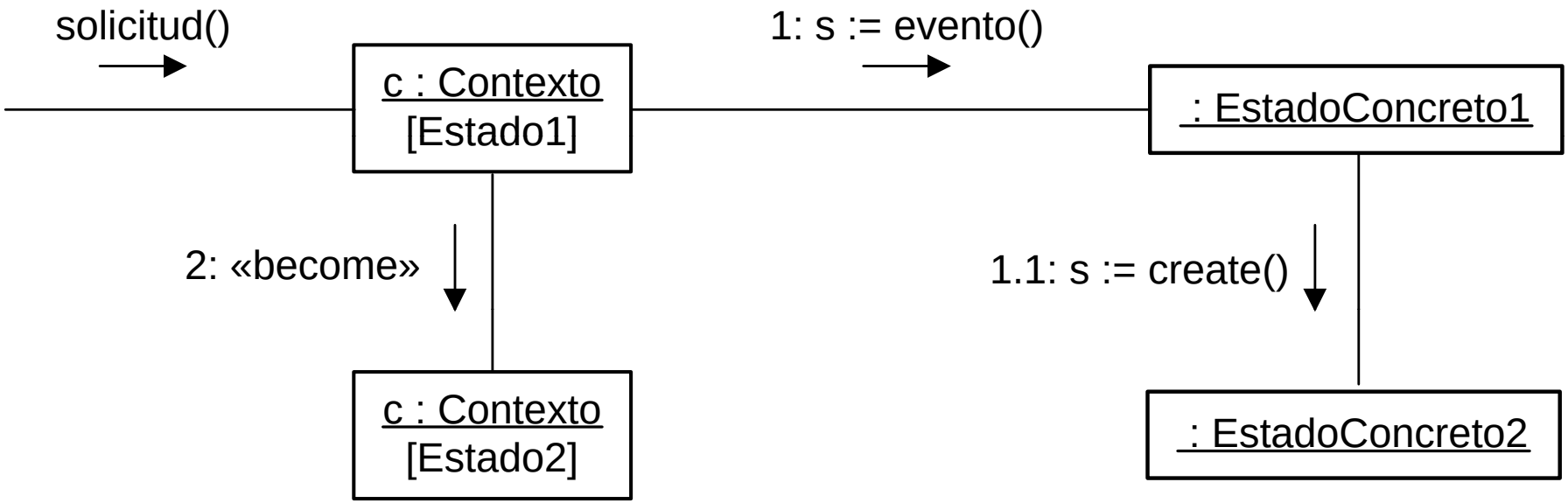
State (3)



- Participantes
  - **Contexto:**
    - Es la clase de objetos cuyo comportamiento varía al cambiar el estado interno
    - Mantiene una referencia a un estado concreto
    - Delega el comportamiento variable al estado concreto actual
  - **Estado:** generaliza los diferentes estados concretos del Contexto
  - **EstadoConcreto:** cada una de estas clases implementa un comportamiento particular del Contexto que sea dependiente del estado

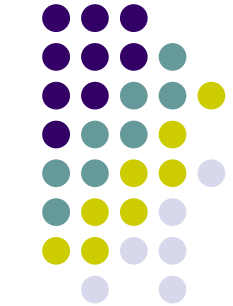
State (4)

- Interacciones



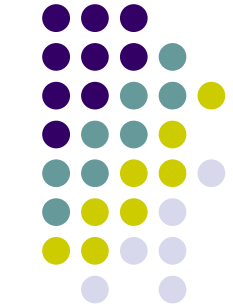
El Contexto, estando en Estado1, al recibir una “solicitud” cambia a Estado2

State (5)



- Consecuencias
  - Sin Garbage Collector es necesario asignar a alguien la responsabilidad de eliminar luego de una transición la instancia que representa el estado anterior
  - Los estados concretos pueden tener estado propio
    - Si no lo tienen pueden ser diseñados como singletons
  - En casos en que el Contexto tiene muchos estados la cantidad de clases (a causa de los estados concretos) puede ser muy grande

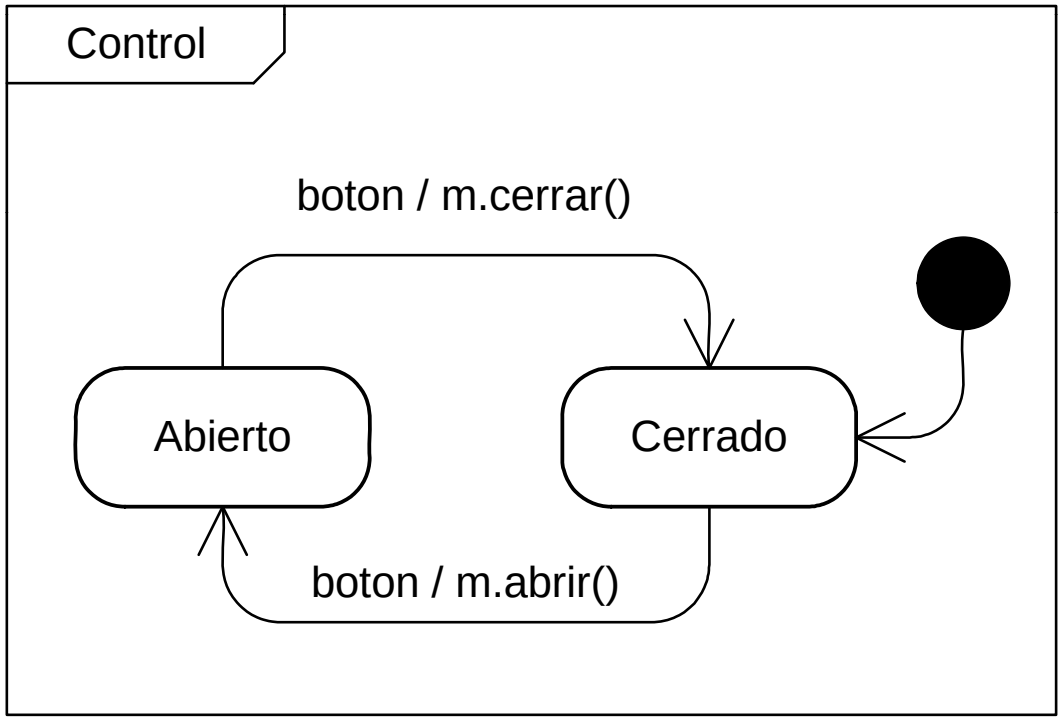
State (6)

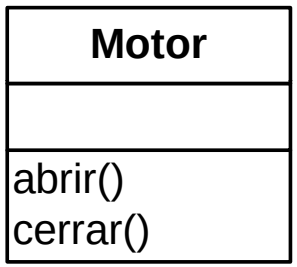


- Consecuencias (cont.)
  - Es simple agregar nuevos estados
    - Sin embargo es necesario modificar estados concretos existentes para incluir transiciones al estado nuevo
  - Existen diferentes variantes en la implementación de las transiciones
  - El Contexto puede pasarse como parámetro en los eventos
  - Permite eliminar la lógica condicional del Contexto
  - El estado concreto actual puede usarse para determinar el estado del Contexto

State (7)

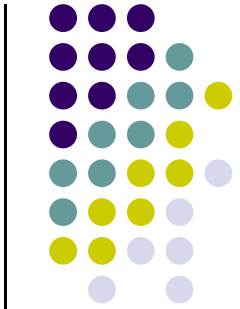
- Ejemplo
  - Puerta automática controlada por control remoto de un solo botón





`Motor` maneja el motor que abre y cierra la puerta

State (8)

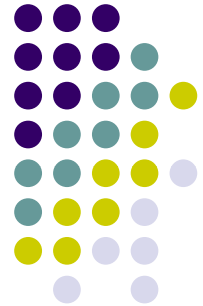


```
class Control {
    private EstadoControl estado;
    private Motor motor;

    public Control() {
        estado = new Cerrado();
    }
    public void boton() {
        EstadoControl nuevo;
        nuevo = estado.boton(motor); //delega el comportamiento
        estado = nuevo;              //provoca la transicion
    }
}

abstract class EstadoControl {
    public abstract EstadoControl boton(Motor);
}
```

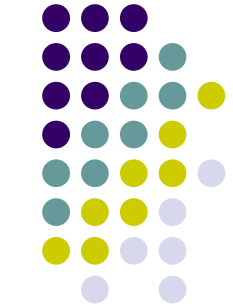
State (9)



```
class Abierto subclass of EstadoControl {
   public EstadoControl boton(Motor m) {
      EstadoControl nuevo;
      m.cerrar();
      nuevo = new Cerrado();
      return nuevo;
   }
}

class Cerrado subclass of EstadoControl {
   public EstadoControl boton(Motor m) {
      EstadoControl nuevo;
      m.abrir();
      nuevo = new Abierto();
      return nuevo;
   }
}
```

Observer



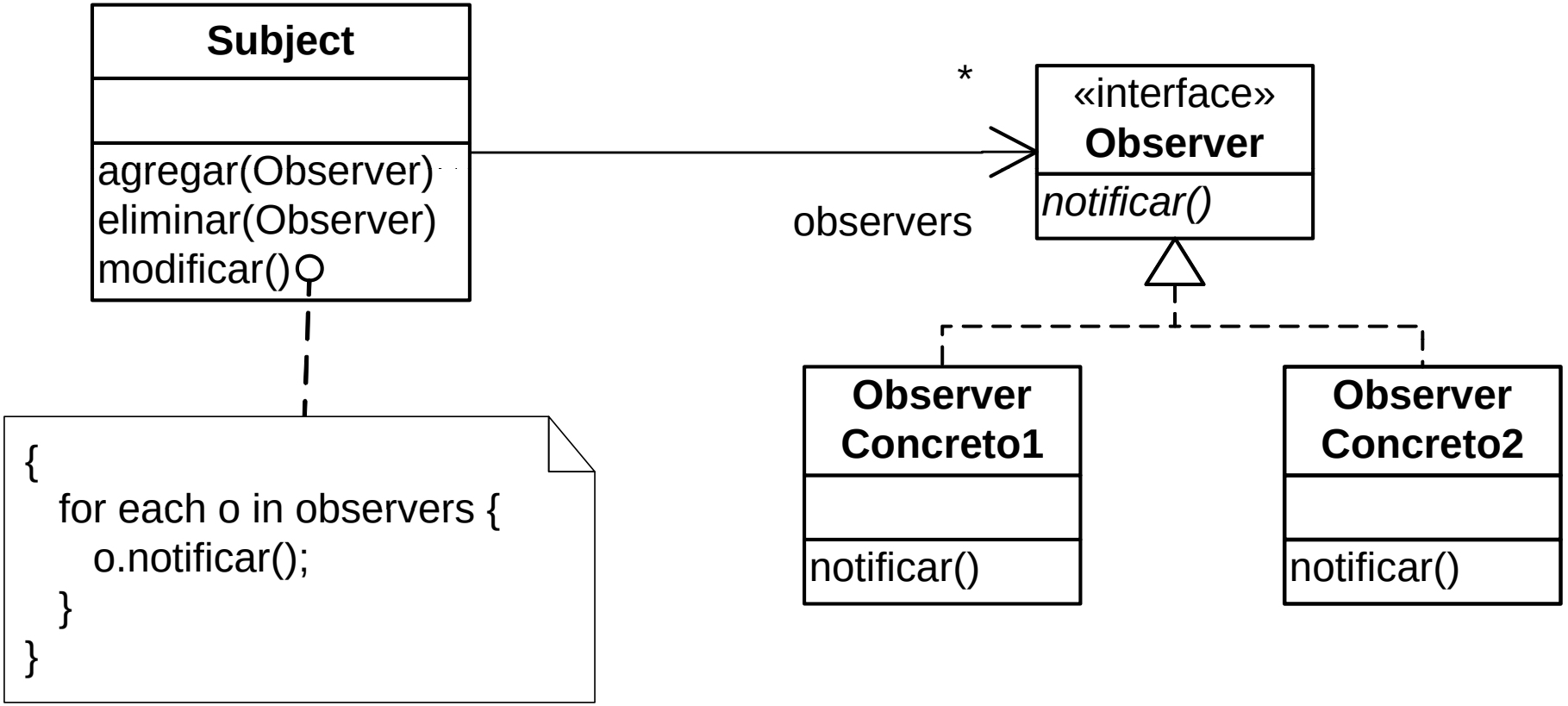
- Problema Tipo:

> "Definir una dependencia 1-n entre objetos, de forma que cuando uno cambie de estado todos los dependientes sean notificados"

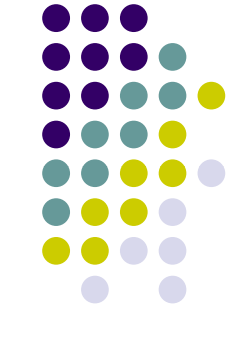
- Aplicabilidad
  - Cuando un cambio en un objeto requiere cambiar otros y no se desea saber cuántos son
  - Cuando un objeto debe notificar otros objetos de diferente naturaleza y sin estar acoplado a ellos

Observer (2)

- Estructura



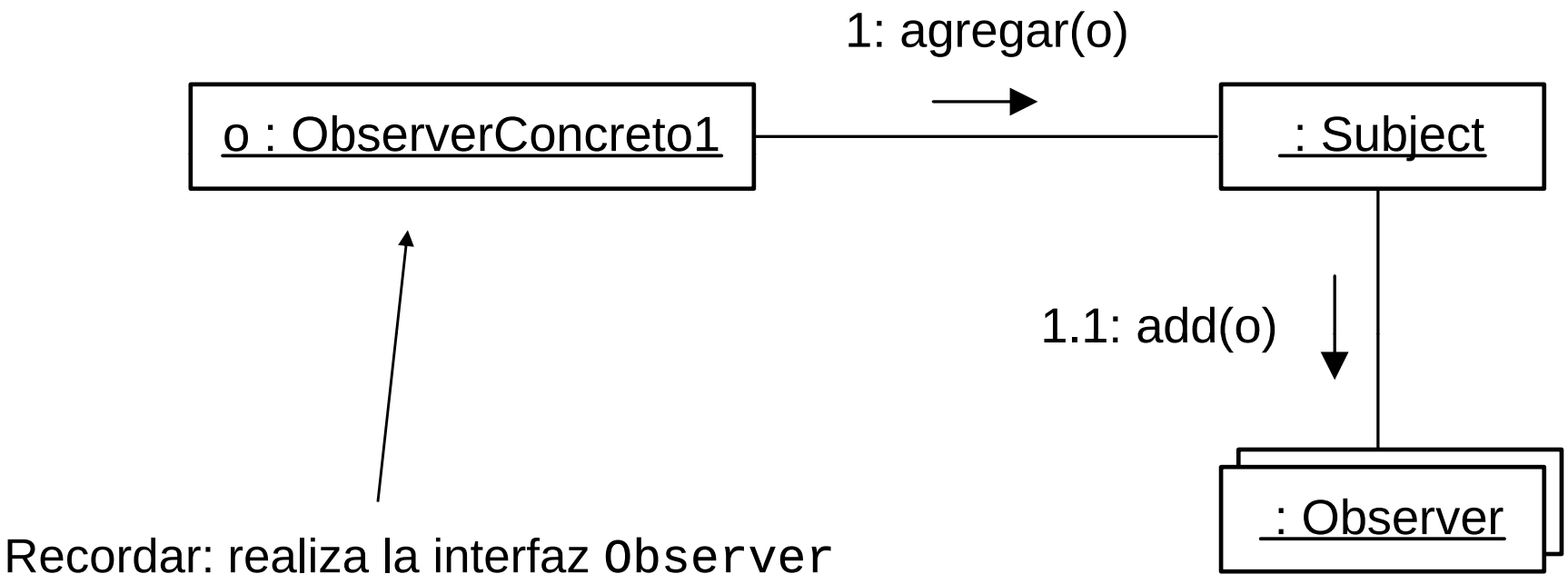
Observer (3)



- Participantes
  - **Subject:**
    - Dispone de información que es de interés para otros objetos
    - Registra un conjunto de objetos interesados (condición: ser compatible con la interfaz Observer)
    - Notifica a los interesados cuando considera necesario
  - **Observer:** declara la operación por la cual los interesados son notificados
  - **ObserverConcreto:** un interesado en los avisos que el Subject tenga para enviar

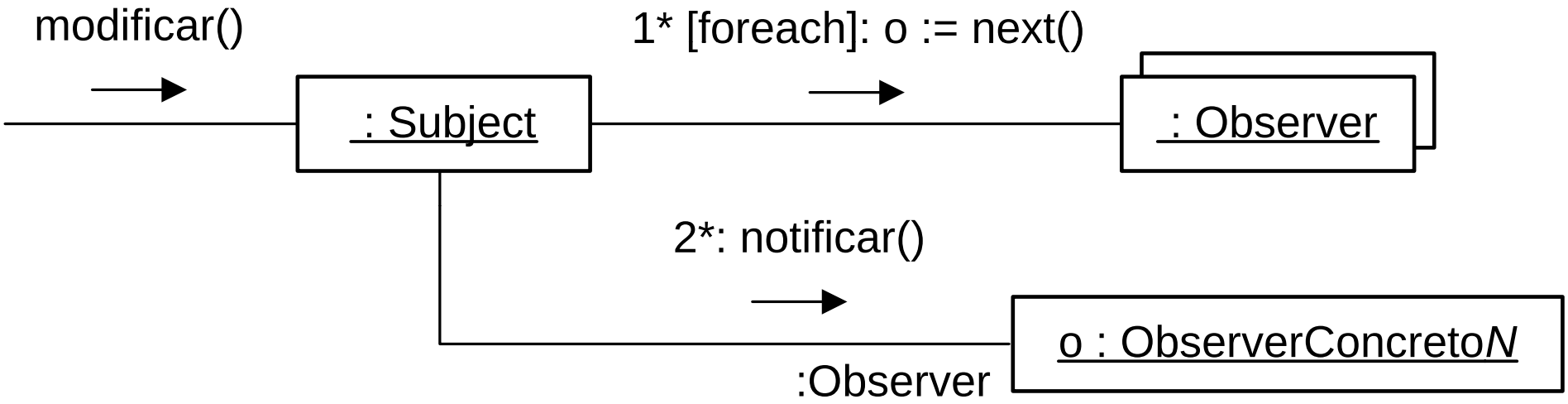
Observer (4)

- Interacciones – `agregar()`

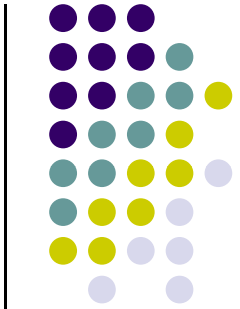


Observer (5)

- Interacciones – `modificar()`

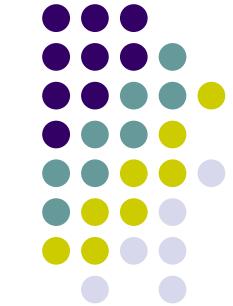


Observer (6)



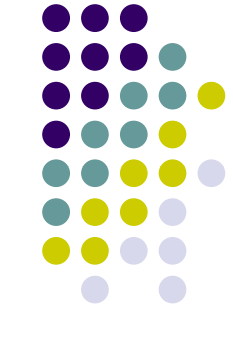
- Consecuencias
  - No existe acoplamiento entre el Subject y los observadores concretos
  - La interfaz Observer es propuesta por el Subject
  - Este mecanismo es aplicable a la realización de broadcasts
  - Cambios inesperados en el Subject por parte de un observador concreto puede causar notificaciones en cascada hacia los otros observadores concretos

Observer (7)



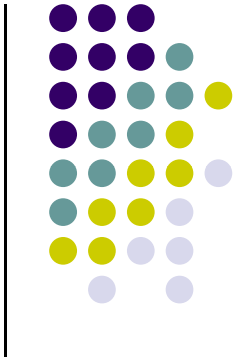
- Consecuencias (cont.)
  - Por defecto, el orden de notificación es aleatorio o al menos no se puede asumir nada al respecto
  - La notificación es secuencial
    - Si a un observador concreto le toma demasiado tiempo procesar la notificación, los demás quedarán esperando
  - La notificación puede llevar parámetros
  - En casos en que un observador observe a mas de un Subject a la vez puede ser necesario que éste se identifique al momento de notificar

Observer (8)



- Ejemplo
  - Aplicación con reloj digital y agenda de tareas programadas
    - Clase `Timer` (Subject) que mantiene la hora actual recibiendo del ambiente invocaciones a `tick()` (modificar) a intervalos regulares predefinidos
    - Interfaz `TimeObserver` (Observer) que declara la operación `notifyTick()` (notificar)
    - Clase `RelojDigital` (ObservadorConcreto) que actualiza la hora en pantalla
    - Clase `Agenda` (ObservadorConcreto) que busca tareas a ser comenzadas en la hora actual

Observer (9)



- Ejemplo (cont.)
  - Cada vez que el `Timer` recibe un `tick()` notifica a los observadores (`RelojDigital` y `Agenda`)
  - Obtención de la hora actual
    - El `Timer` puede pasarse a sí mismo como parámetro en la notificación para que le sea preguntada la hora
    - El `Timer` puede pasar la hora como parámetro en la notificación para que los observadores la usen directamente
  - El `RelojDigital` y la `Agenda` lo único que tienen en común es el interés en enterarse del paso del tiempo

Strategy

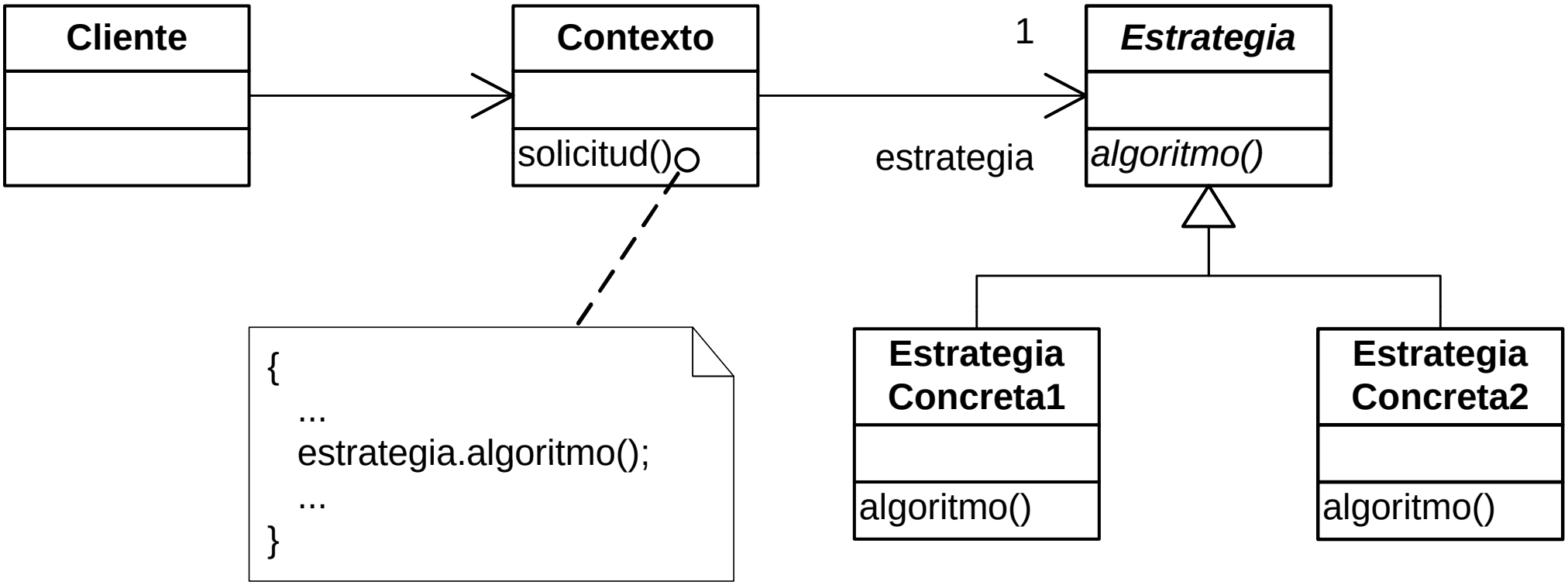
- Problema Tipo:

> "Definir una familia de algoritmos, encapsularlos y hacerlos intercambiables. Strategy permite que un algoritmo varíe independientemente de los clientes que lo usan"

- Aplicabilidad
  - Cuando se necesitan diferentes variantes de un algoritmo
  - Cuando un algoritmo utiliza datos que los clientes no deben conocer

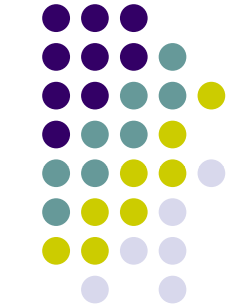
Strategy (2)

- Estructura



El Cliente depende además de las estrategias concretas

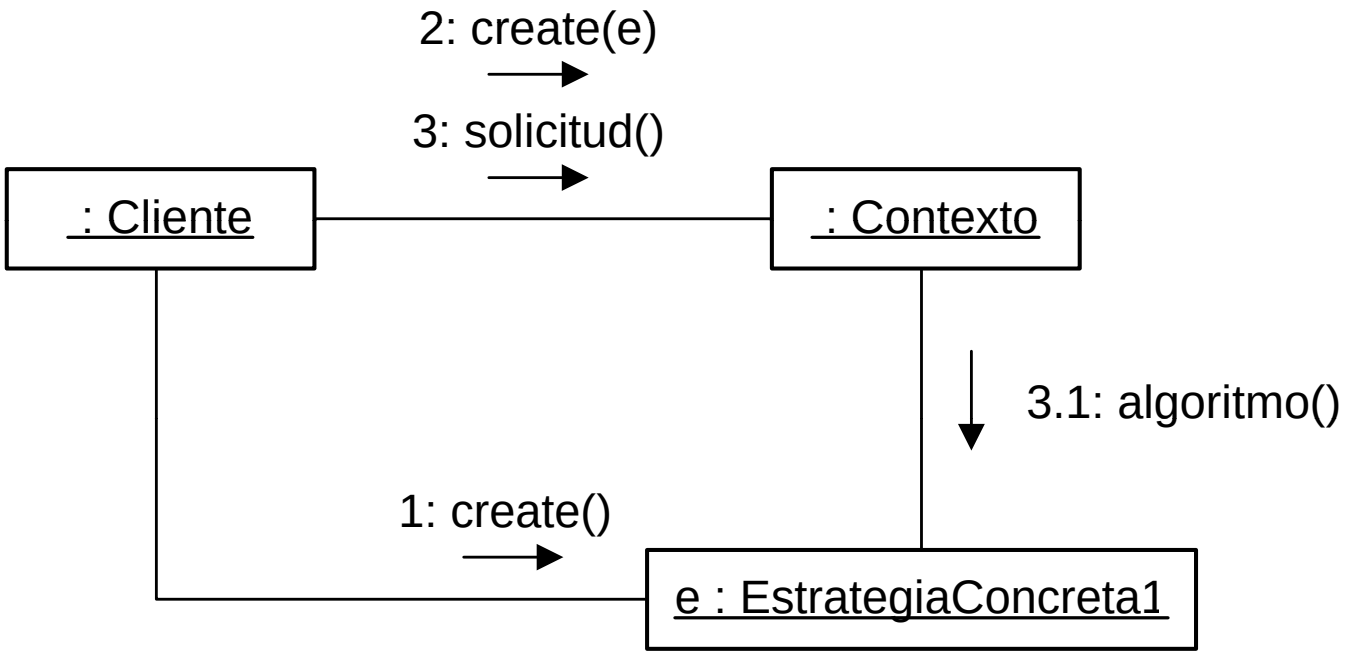
Strategy (3)



- Participantes
  - **Cliente:** Tiene una cierta responsabilidad y delega parte de ella al contexto
  - **Contexto:**
    - Resuelve parte del trabajo del cliente (para ello utilizará variantes de una misma estrategia)
    - Mantiene una referencia a una estrategia y está configurado con una estrategia concreta
  - **Estrategia:** Generaliza las diferentes estrategias
  - **EstrategiaConcreta:** Implementa el algoritmo que representa una variante de la estrategia

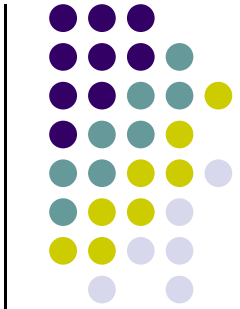
Strategy (4)

- Interacciones



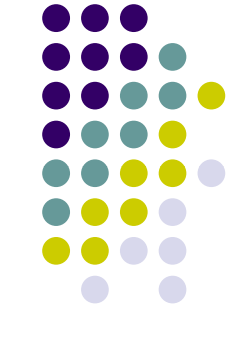
El Contexto al recibir la solicitud (entre otras cosas) utiliza la variante del algoritmo encapsulada en el objeto que recibió en su creación

Strategy (5)



- Consecuencias
  - Se define una familia de algoritmos similares mediante la cual es posible elegir una implementación (variante) particular
  - La lógica condicional es eliminada del Contexto
  - Los clientes quedan acoplados con las estrategias concretas por lo que es recomendable utilizar este patrón solamente cuando las variantes sean relevantes para los clientes

Strategy (6)

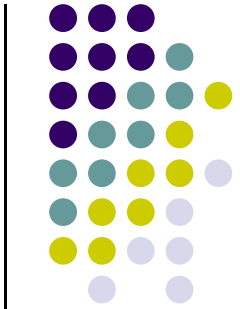


- Consecuencias (cont.)
  - Diferentes alternativas para que las estrategias tengan disponibles los datos que necesitan para funcionar
    - El Contexto pasa dicha información como parámetro
    - El Contexto se pasa a sí mismo como parámetro y las estrategias concretas solicitan la información correspondiente
  - Las estrategias aumentan la cantidad de objetos de la aplicación
    - De ser posible implementarlas como objetos sin estado

Strategy (7)

- Ejemplo
  - Las transacciones realizadas con tarjetas de crédito presentan variantes en su liquidación según el país donde haya sido realizada
    - Clase `Transaccion` (Cliente) que debe ser liquidada
    - Clase `Liquidador` (Contexto) que tiene la responsabilidad de liquidar una transacción
    - Clase `EstrategiaLiquidacion` (Estrategia) que declara las operaciones `totalImpuestos()` y `darComision()`
    - Clases `LiquidacionUruguay` y `LiquidacionBrasil` (EstrategiaConcreta) encapsulan las variantes de los algoritmos

Strategy (8)

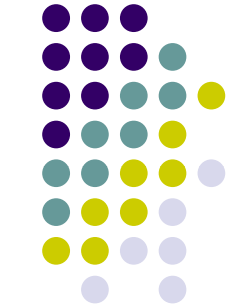


```
class Liquidador {
   private EstategiaLiquidacion estrategia;
   public Liquidador(EstrategiaLiquidacion e) {
      estrategia = e;
   }

   public void liquidar(Transaccion t) {
      ...
      t.setImpuestos(estrategia.totalImpuestos(...));
      ...
      t.setComision(estrategia.darComision(...));
      ...
   }
}

// La Transaccion (sabiendo de donde es) tan solo debe
// crear la estrategia correcta, con ella crear un
// Liquidador e invocarle liquidar()
```

Strategy (9)



```
abstract class EstrategiaLiquidacion {
   // Podria tener algun atributo para mantener
   // datos del contexto a ser usados en los algoritmos
   public abstract float totalImpuestos(...);
   public abstract float darComision(...);
}

class LiquidacionUruguay subclass of EstrategiaLiquidacion {
   public float totalImpuestos(...) {
      ... // Calculo de impuestos para una compra en Uruguay
   }
   public float darComision(...) {
      ... // Calculo de comision para una compra en Uruguay
   }
}

// La clase LiquidacionBrasil es analoga a LiquidacionUruguay
```