

METODOLOGÍAS DE PROGRAMACIÓN I

Patrón de comportamiento *Strategy*

Situación de ejemplo

- Se posee una clase llamada *SalaDeTeatro* la cual tiene una matriz de *Asientos*, donde cada elemento de la matriz corresponde a un asiento de la sala. El asiento posee un estado que es un booleano que posee el valor **true** si está disponible y **false** en caso contrario.
- La clase *SalaDeTeatro* posee el método *vender*(fila, columna) el cual permite vender la localidad recibida por parámetro, si es que está libre. Este método devuelve **true** si la venta se pudo concretar y **false** en caso contrario (el asiento ya estaba vendido)

Problema

- Se desea agregar a la clase *SalaDeTeatro* la capacidad de vender más de un asiento a la vez.

```
string vender(int cantidad)
```

- La característica que se pide es que la cantidad de asientos a vender deben estar contiguos en la misma fila.

-	-	-	-	-	-
-	X	-	X	X	X
X	X	-	-	-	X
-	-	-	-	-	-
-	X	-	X	X	X
X	X	-	-	-	X
-	-	-	-	-	-
-	X	-	X	X	X
X	X	-	-	-	X

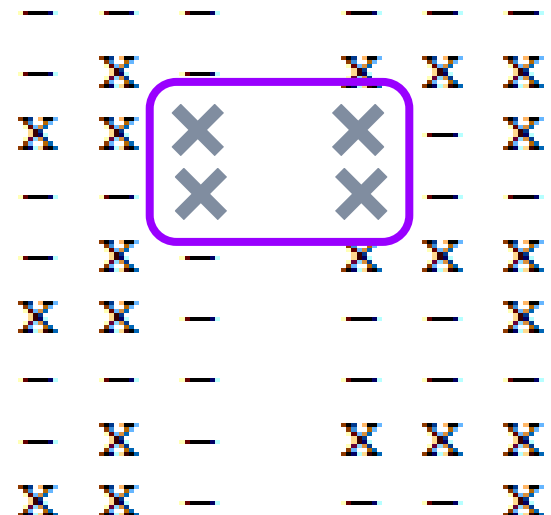
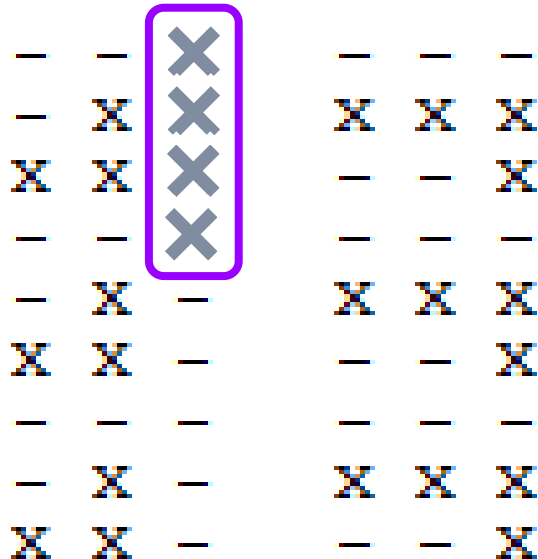
Se desean cuatro asientos



X	X	X	X	-	-
-	X	-	X	X	X
X	X	-	-	-	X
-	-	-	-	-	-
-	X	-	X	X	X
X	X	-	-	-	X
-	-	-	-	-	-
-	X	-	X	X	X
X	X	-	-	-	X

Problema

- ¿Cómo resolveríamos el problema si sabemos que a futuro se desean vender asientos contiguos en una columna o en bloques?
- ¿Cómo se resuelve el problema si pueden aparecer más formas de vender una cierta cantidad de asientos?



Problema

- Conociendo la implementación actual, sabiendo el nuevo requerimiento, sabiendo lo que se puede desear a futuro y sabiendo que pueden aparecer más opciones a futuro
¿Cómo se implementa el método vender?

```
string vender(int cantidad)
```



Problema

Una primera solución sería ...

```
string vender(int cantidad)
```

```
    if POR_FILAS: ...
```

```
    if POR_COLUMNAS: ...
```

```
    if POR_BLOQUE: ...
```

Problema

Una primera solución sería ...

```
string vender(int cantidad)
```

```
    if POR_FILAS: ...
```

```
    if POR_COLUMNAS: ...
```

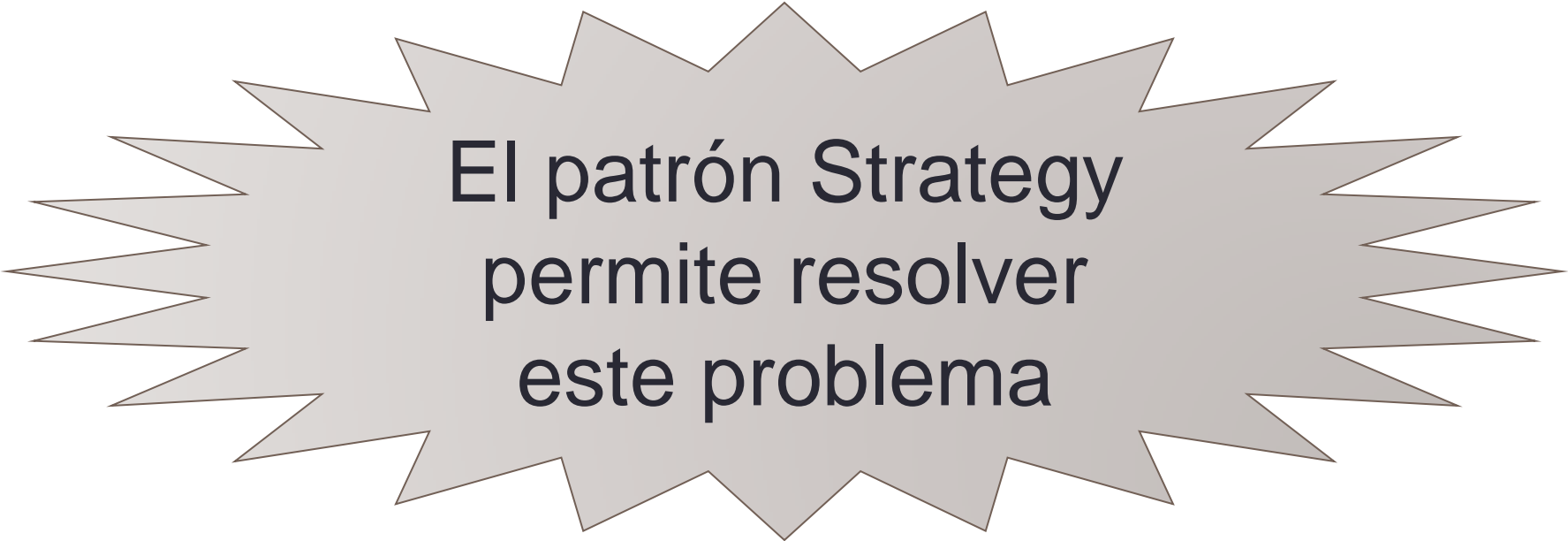
```
    if POR_BLOQUE: ...
```

¿Qué sucede si aparecen más políticas?

¿Cómo mantenemos este código si diferentes teatros (que usan la clase *SalaDeTeatro*) deseen tener distintas políticas de venta para sus salas?

Motivación

- Buscamos un diseño que permita agregar, eliminar e intercambiar políticas, según diferentes situaciones.



El patrón Strategy
permite resolver
este problema

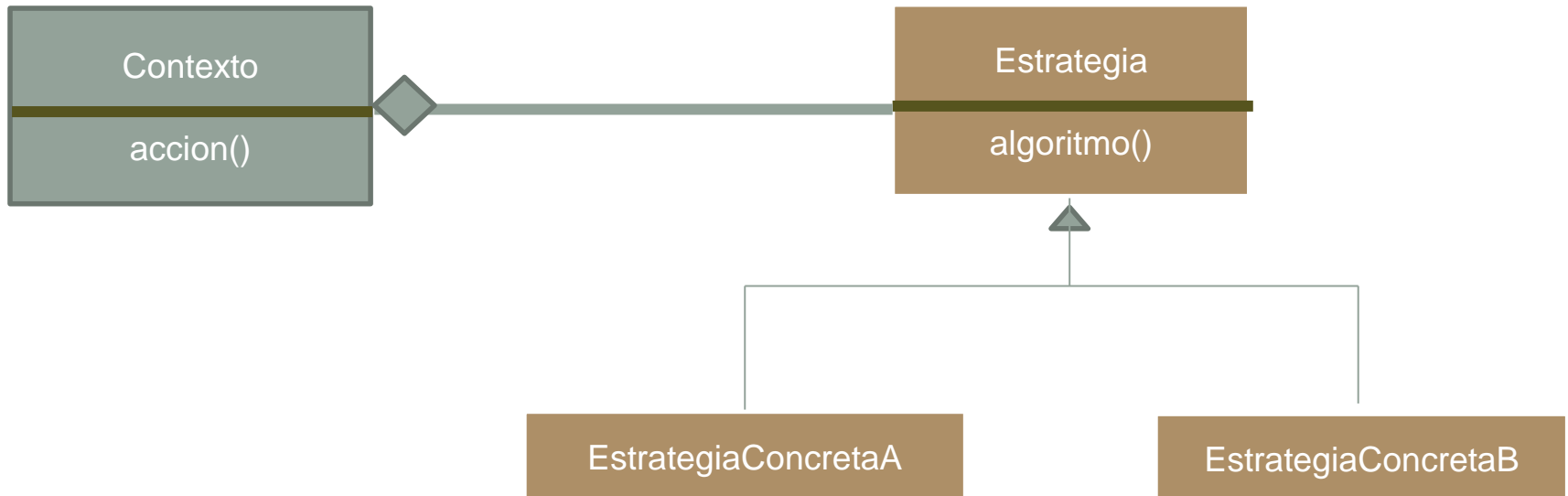
Strategy

Propósito: define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. Permite que un algoritmo varíe independientemente de los clientes que lo usan.

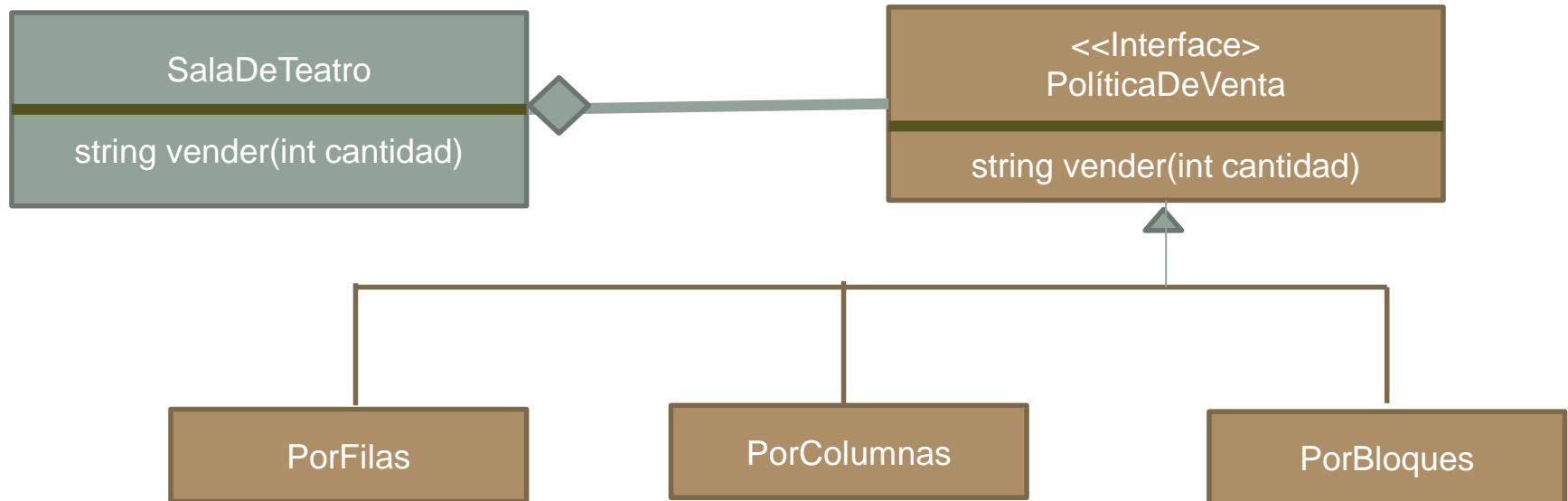
Aplicabilidad: usarlo cuando

- Muchas clases relacionadas difieren solo en su comportamiento.
- Se desee configurar una clase con un determinado comportamiento entre muchos posibles.
- Se necesitan muchas variantes de un mismo algoritmo

Strategy - Estructura



Strategy - Estructura



Strategy - Implementación

```
interface PoliticaDeVenta
```

```
    string vender(int cant, Asiento[,] asientos)
```

Declaramos *PoliticaDeVenta*
como una interface

Strategy - Implementación

```
class PorFila : PoliticaDeVenta
    string vender(int cant, Asiento[,] asientos)

        // Búsqueda de "cant" asientos consecutivos
        // en la misma fila
```

Strategy - Implementación

```
class PorColumna : PoliticaDeVenta
    string vender(int cant, Asiento[,] asientos)

    // Búsqueda de "cant" asientos consecutivos
    // en la misma columna
```

Cada clase que implemente
PoliticaDeVenta implementa su
propia versión de *vender*.

Strategy - Implementación

```
class SalaDeTeatro
```

```
    PoliticaDeVenta politica
```

```
    Asiento[,] asientos
```

```
    constructor (int filas, int columnas)
```

```
        . . .
```

```
        politica = new PorFilas()
```

```
    string vender(int cantidad)
```

```
        return politica.vender(cantidad, asientos)
```

```
    void cambiarPolitica(nuevaPolitica)
```

```
        politica = nuevaPolitica
```

Strategy - Implementación

```
class SalaDeTeatro
```

```
    PoliticaDeVenta politica
```

```
    Asiento[,] asientos
```

```
    constructor (int filas, int columnas)
```

```
        . . .
```

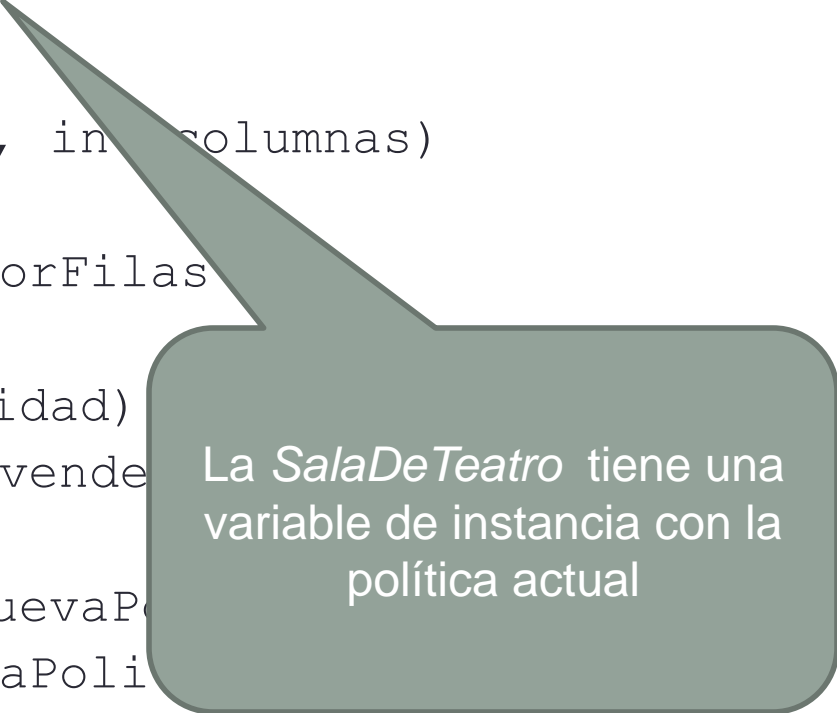
```
        politica = new PorFilas
```

```
    string vender(int cantidad)
```

```
        return politica.vender(cantidad)
```

```
    void cambiarPolitica(nuevaPolitica)
```

```
        politica = nuevaPolitica
```



La *SalaDeTeatro* tiene una variable de instancia con la política actual

Strategy - Implementación

```
class SalaDeTeatro

    PoliticaDeVenta politica
    Asiento[,] asientos

    constructor (int filas, int columnas)
        . . .
        politica = new PorFilas()

    string vender(int cantidad)
        return politica.vender(cantidad, asientos)

    void cambiarPolitica(nuevaPolitica)
        politica = nuevaPolitica
```

La *SalaDeTeatro* establece en su constructor una política por defecto.

Strategy - Implementación

```
class SalaDeTeatro
```

```
    PoliticaDeVenta politica  
    Asiento[,] asientos
```

```
    constructor (int filas, int columnas)
```

```
        . . .
```

```
        politica = new PorFilas()
```

```
    string vender(int cantidad)
```

```
        return politica.vender(cantidad, asientos)
```

```
    void cambiarPolitica(nuevaPolitica)
```

```
        politica = nuevaPolitica
```

La *SalaDeTeatro* delega en la política
la tarea de buscar los asientos

Strategy - Implementación

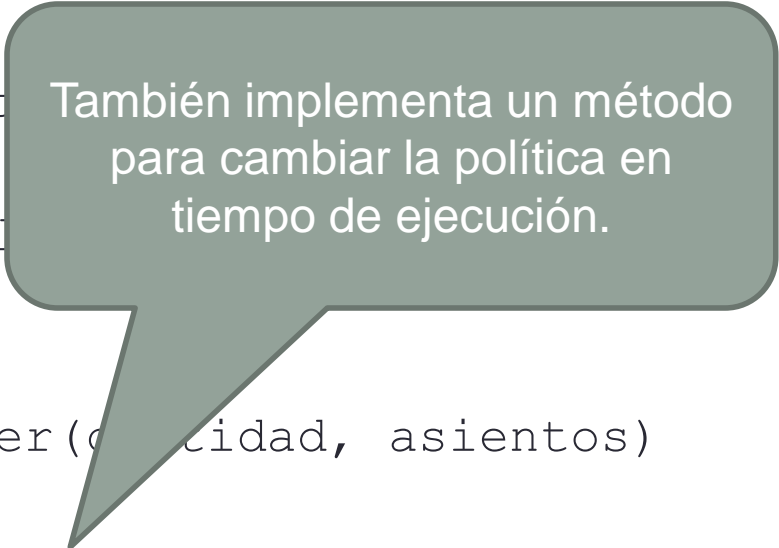
```
class SalaDeTeatro
```

```
    PoliticaDeVenta politica  
    Asiento[,] asientos
```

```
    constructor (int filas, int  
        . . .  
        politica = new PorFil
```

```
    string vender(int cantidad)  
        return politica.vender(cantidad, asientos)
```

```
    void cambiarPolitica(nuevaPolitica)  
        politica = nuevaPolitica
```



También implementa un método para cambiar la política en tiempo de ejecución.

Strategy – Ventajas

- Familia de algoritmos relacionados. La herencia puede ayudar a factorizar estos algoritmos.
- Las estrategias eliminan las sentencias condicionales.
- Las estrategias pueden proporcionar distintas implementaciones del mismo comportamiento
- Resulta muy simple agregar nuevas estrategias.