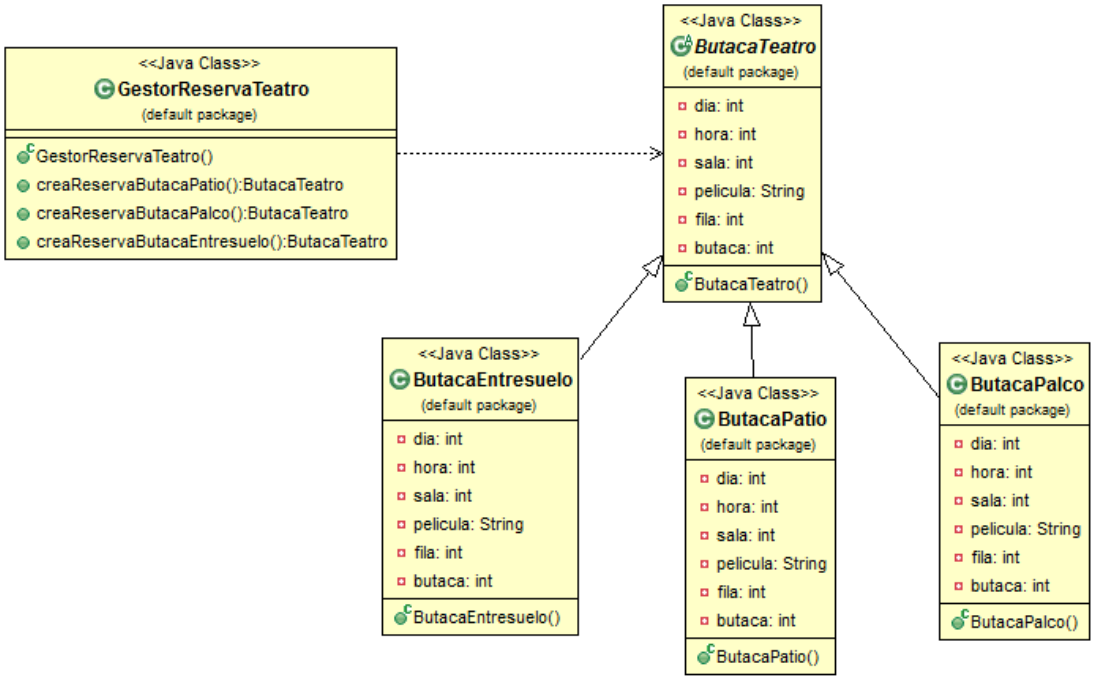
En un teatro en el que se proyectan películas de cine, se dispone de un gestor de reservas modelado por la clase GestorReservaTeatro. Para ello utiliza objetos de la jerarquía de ButacaTeatro.




(Suponer que todas las clases tienen get/set para sus atributos)

A modo de ejemplo se incluye el código de la operación creaReservaButacaPatio(), el resto de operaciones de reserva son similares:

```
public ButacaTeatro creaReservaButacaPatio(){
            return new ButacaPatio();
      };
```

Se desea simplificar el proceso de creación para que con una única operación se puedan generar objetos de cualquiera de estas tres clases, o de otros tipos de butaca que se pudieran considerar en un futuro.

Se pide:

1.- Selecciona el patrón más adecuado para esta tarea. Justifica tu respuesta. Si el patrón utilizado tiene varias versiones justifica cual utilizarías. **(1,5 puntos)**

En este caso necesitamos un patrón de tipo creacional. Ya que hay solo un tipo de producto a generar, el patrón más adecuado es un FactoryMethod. De sus versiones, ya que se pueden utilizar distintos tipos de entradas en un mismo cliente, utilizaremos un FactoryMethod parametrizado.

2- Aplica el patrón seleccionado del punto 1. Describe qué función tiene cada nueva Interfaz/Clase nueva, e implementa las operaciones de cada una de ellas. **(2,5 puntos)**

Creamos el método factoría como un método estático en la clase base común a todas las butacas.

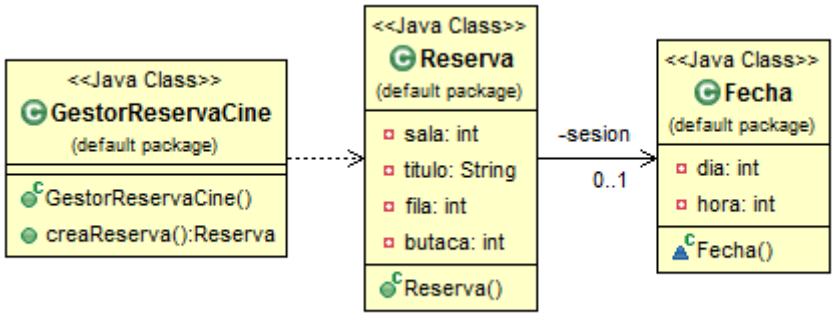
```
public abstract class ButacaTeatro {
	…

	public static ButacaTeatro creaButaca(String tipo){
		switch(tipo){
		case "patio": return new ButacaPatio();
		case "palco": return new ButacaPalco();
		case "entresuelo": return new ButacaEntresuelo();
		default: return null;
		}
	}
}
```

Así crear una butaca, por ejemplo de patio se reduce a invocar a:

ButacaTeatro.creaButaca(“patio”);

No es necesario realizar ningún cambio en la jerarquía, a lo sumo eliminar los métodos craeButacaXXX() anteriores y sustituir sus llamadas por llamadas parametrizadas a creaButaca().

El teatro absorbe un cine que disponía de su propio gestor de reservas tal y como aparece en el siguiente diagrama:




(Suponer que todas las clases tienen get/set para sus atributos)

El teatro decide unificar en su sistema de reservas el del cine. Como consecuencia se pretende que se puedan gestionar objetos "Reserva" como si fueran otro tipo de Butaca.

Se pide:

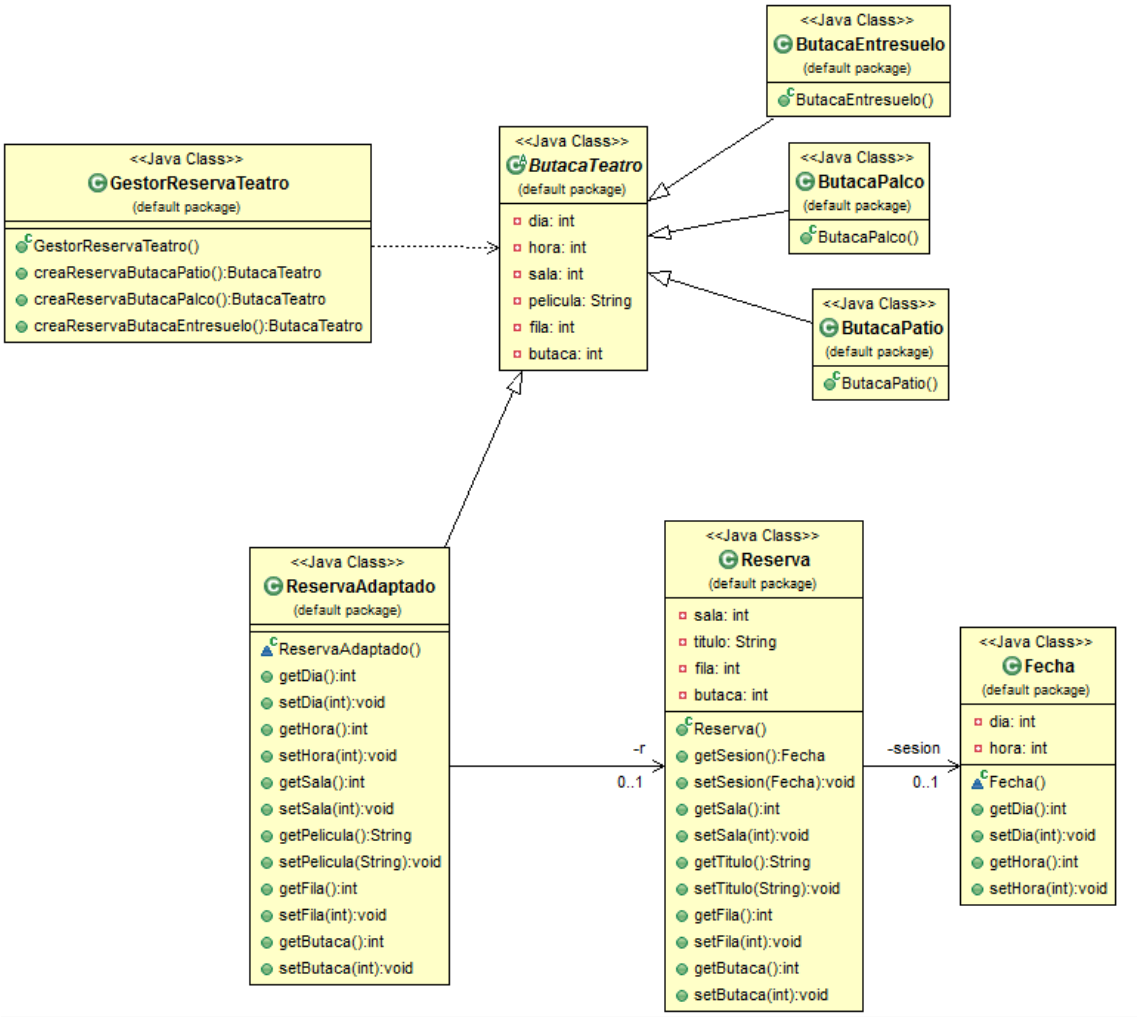
3.- Selecciona el patrón más adecuado para esta tarea. Justifica tu respuesta. Si el patrón utilizado tiene varias versiones justifica cual utilizarías **(1,5 puntos)**

En este caso necesitamos un patrón estructural que nos permita manejar objetos Reserva como si fueran otra subclase de ButacaTeatro. Para ello vamos a usar un patrón Adapter. Ya que tanto Reserva como ButacaTeatro son clases es necesario utilizar un Adapter de Objetos ya que no podemos heredar de ambas clases simultáneamente (al menos en Java).

4.- Aplica el patrón seleccionado del punto 3. Describe qué función tiene cada nueva Interfaz/Clase nueva, e implementa las operaciones de cada una de ellas. **(2,5 puntos)**

Creamos la clase ReservaAdaptado que hereda de ButacaTeatro, para comportarse como una butaca más, y maneja por composición un objeto Reserva para gestionar toda la información.

Como resultado, tenemos que sobrescribir todos los métodos de get/set de ButacaTeatro para que accedan al campo privado de tipo Reserva.




La clase en cuestión quedaría de la siguiente manera:

```
public class ReservaAdaptado extends ButacaTeatro {
	private Reserva r;  // objeto Reserva por composición
	ReservaAdaptado(){
		r = new Reserva(); // inicializamos el objeto Reserva
	}

	// sobrescribimos observadores y modificadores, enmascarando los campos
	// heredados y utilizando en su lugar los de r
	@Override
	public int getDia() {

		return this.r.getSesion().getDia();
	}
	@Override
	public void setDia(int dia) {

		r.getSesion().setDia(dia);
	}
	@Override
	public int getSala() {

		return super.getSala();
	}
	// y de forma similar el resto de los métodos
}
```

Se pretende incorporar a una aplicación móvil el gestor de reservas (nuestra clase GestorReservaTeatro). Para minimizar los recursos utilizados en el dispositivo, y simplificar los cambios en la aplicación, sería deseable que:

i. Hasta que no se necesite por primera vez no debería existir ninguna instancia de esta clase.
ii. No exista más de una instancia de esta clase en toda la aplicación.
iii. Dicha instancia debería estar disponible desde cualquier punto del código de la aplicación.

5.- Selecciona el patrón más adecuado para esta tarea. Justifica en tu respuesta de qué forma se cubrirían esos tres puntos deseables en la aplicación. Si el patrón utilizado tiene varias versiones justifica cual utilizarías. **(2 puntos)**

En este caso haremos uso de un patrón creacional, en concreto el patrón Singleton ya que cubre todas nuestras necesidades. De mano oculta los constructores para que no se puedan crear nuevas instancias. Dispone además de un método estático que nos asegura la accesibilidad desde cualquier punto del proyecto. Y finalmente el método anterior genera en la primera llamada una instancia de la clase, pero las posteriores llamadas devuelven siempre esa misma instancia generada inicialmente.