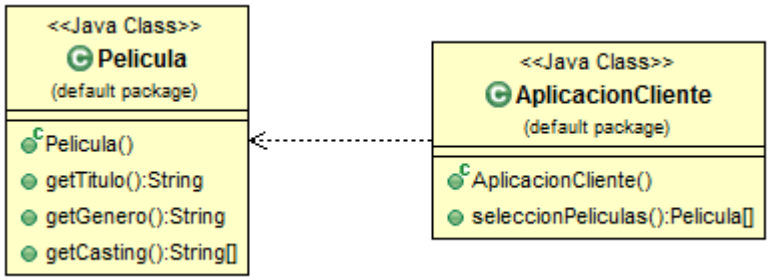
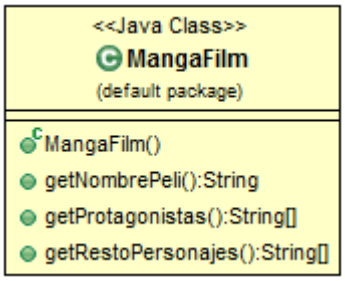
Se dispone de una aplicación **AplicacionCliente** de búsqueda de información sobre películas. Estas películas están almacenadas en una gran base de datos en Internet, cada una de ellas se maneja como un objeto de la clase **Pelicula**. El siguiente diagrama muestra en formato UML ambas entidades.



Los desarrolladores de esta aplicación han llegado a un acuerdo con los propietarios de una segunda base de datos de películas, aunque en este caso son todas películas del mismo género: "Manga". En esta base de datos cada película viene representada como un objeto de la clase MangaFIlm.



Aunque los propietarios de esta segunda base de datos nos proporcionan acceso a todas sus películas, no nos permiten introducir modificaciones en su clase **MangaFilm**. Nos gustaría que nuestra **AplicacionCliente** pudiera manipular en sus métodos (como por ejemplo en **seleccionPeliculas**()) de forma indistinta nuestras películas originales o películas manga.

Se pide:

1.- Selecciona el patrón más adecuado para esta tarea. Justifica tu respuesta. Si el patrón utilizado tiene varias versiones justifica cual utilizarías. **[2 puntos]**

En este caso necesitamos un patrón estructural Adapter que nos permita manejar las películas manga, clase **MangaFilm**, como si fuesen del tipo **Pelicula**. En este caso hay dos versiones: Adapter de clases y de objetos.

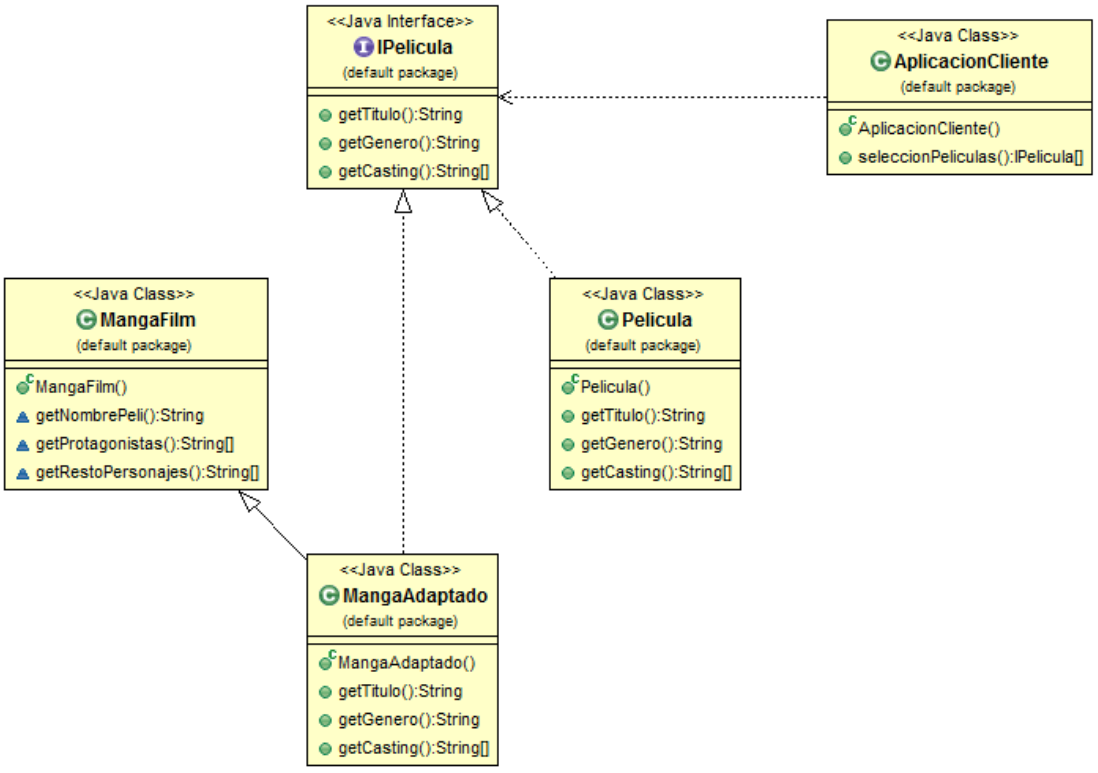
En nuestro caso podemos utilizar ambas versiones respetando que no podemos modificar **MangaFilm**.

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. Modifica la definición de la clase **AplicacionCliente** si fuese necesario **[3 puntos]**

Plantearemos a modo de ejemplo las dos versiones.

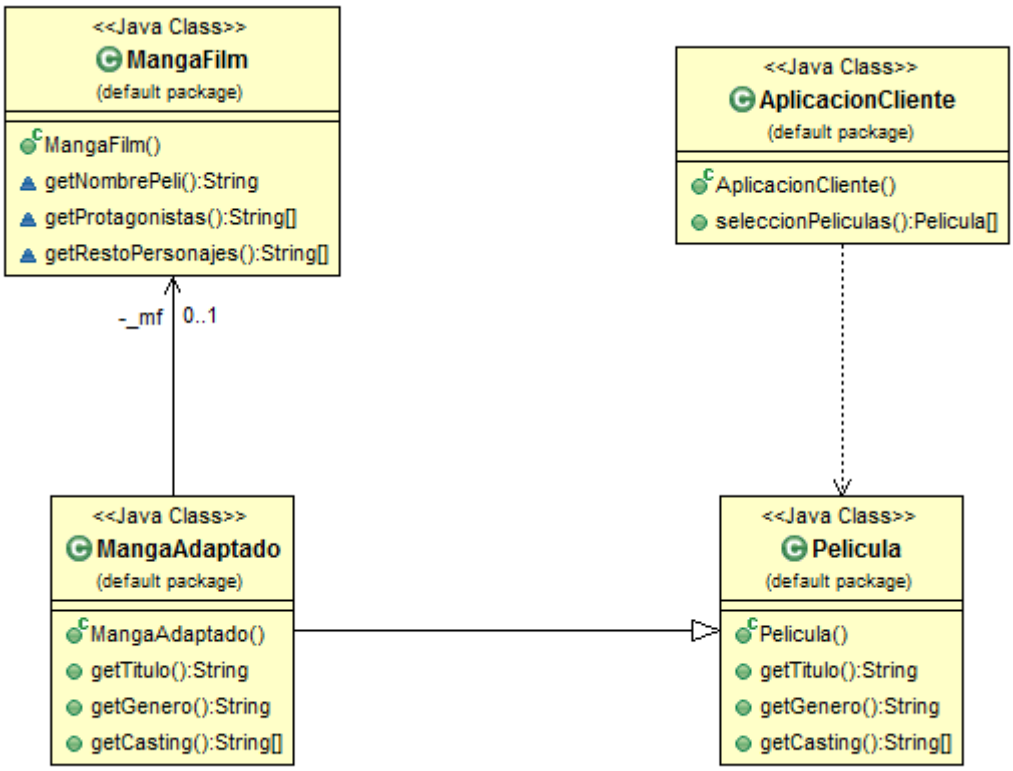
Adaptador de clases:

Creamos una interface **IPelicula** con los prototipos de los métodos de **Pelicula**. Creamos el adaptador de clases **MangaAdaptado** que hereda de **MangaFilm** e implementa **IPelicula**. **Pelicula** también implementará de forma natural la interface **IPelicula**. Finalmente **AplicacionCliente** pasa a manejar objetos **IPelicula** para poder manipular indistintamente los dos tipos de objetos originales: **MangaFilm** (a través del adaptador **MangaAdaptado**) y **Pelicula**.



Adaptador de Objetos:

En esta versión definimos la clase adaptadora **MangaAdaptado** que hereda de **Pelicula** y utiliza por composición **MangaFilm** en un atributo privado. Se sobrescriben los métodos de **Pelicula** en **MangaAdaptado** utilizando los métodos del atributo privado. La clase **AplicacionCliente** manipula indistintamente los dos tipos de objetos a través del tipo **Pelicula**, ya que por herencia **MangaAdaptado** también es de la clase **Película.**



La aplicación de móvil para consultar las películas utiliza la siguiente clase para acceder a la base de datos en la Web.

```
public class WebDataBase {
	private int totalConsultas; // número de consultas realizadas a través de este objeto

	public WebDataBase()
	{
		totalConsultas = 0;
		// Habilita el servicio de datos del móvil
		// Trata de conectar a través de Internet con la BD de películas
		// Si todo va bien crea una conexión remota y la mantiene abierta.
	}

	public int getTotalConsultas() {
		// devuelve el total de consultas
		return totalConsultas;
	}

	String[] consulta(String criteriosDeBusqueda){
		String[] resultados = null;
		// Buscar en la BD remota las películas que cumplan
		// los criterios de búsqueda y las devuelve en resultados

		totalConsultas++; // se cuenta la consulta hecha

		return resultados;
	}
}
```

Los inconvenientes que presenta esta aplicación son los siguientes:

a. Cada vez que se accede en la aplicación al menú de consultas de la aplicación, se crea un objeto **MenuConsulta** (e internamente un objeto de esta clase **WebDataBase** consumiendo recursos de datos del móvil aunque luego no se haga ninguna consulta a Internet).
b. Es engorroso reutilizar la misma instancia **WebDataBase** porque exigen "arrastrar" como argumento en cada método donde se desea realizar búsquedas.
c. Como consecuencia, cada vez que se accede al menú de consultas se crea un nuevo objeto **WebDataBase**, en lugar de reutilizar el que ya existe y perdiendo el contador de consultas.

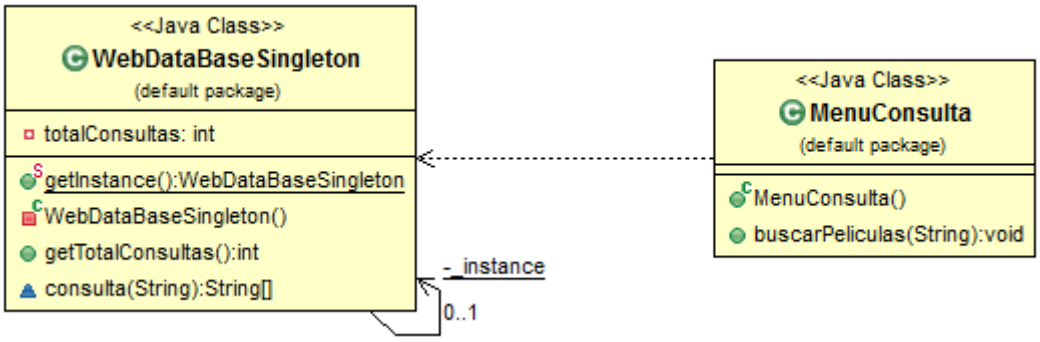
```
class MenuConsulta{
  private WebDataBase wdb;

  public MenuConsulta(){
    wdb = new WebDataBase(); // se crea una conexión remota con la BD
    //…
  }
  public void buscarPeliculas(String criterios){
    for(String titulo: wdb.consulta(criterios))
      System.out.println(titulo);
  }
}
```

3.- Propón un patrón de diseño que permita solventar los problemas anteriores. Justifica de qué forma se podría solucionar cada uno de ellos aplicando dicho patrón. Además Indica su tipo, y si hay más de una versión del patrón justifica que versión utilizarías. **[2 puntos]**

En este caso necesitamos un patrón creacional. Concretamente el patrón Singleton. Este patrón nos ofrece una solución a cada uno de los problemas planteados:

a. Por medio de su instancia privada, Singleton permite utilizar siempre el mismo objeto de la clase **WebDataBase**. Además su inicialización perezosa permite crear el objeto sólo cuando realmente se necesite.
b. Al ofrecer el acceso mediante métodos estáticos, estarán disponibles desde cualquier punto del código sin tener que recibir como argumento el objeto en cuestión en cada método que se necesite.
c. Al acceder siempre a la misma instancia sus atributos se conservan entre usos y no perdemos la cuenta del número de consultas. Además el bloquear el new con constructores privados evita crear instancias de la clase por error.

4.- Aplica el patrón seleccionado en el punto anterior. Describe qué función tiene cada nueva Interfaz/Clase nueva, e implementa las operaciones de cada una de ellas. Pon un ejemplo de cómo se crearía un objeto del **MenuConsulta** resultante y de invocación a su método **buscarPeliculas() [3 puntos]**




```
public class MenuConsulta {

	public MenuConsulta(){
		//…
	}
	public void buscarPeliculas(String criterios){
		WebDataBaseSingleton wdb = WebDataBaseSingleton.getInstance();

		for(String titulo: wdb.consulta(criterios))
			System.out.println(titulo);
	}

}
```

```
public class WebDataBaseSingleton {
	//Singleton - una única instancia
	private static WebDataBaseSingleton _instance = null;

	// Singleton - acceso a la instancia con inicialización perezosa
	public static WebDataBaseSingleton getInstance(){
		if (_instance == null)
			_instance = new WebDataBaseSingleton();
		return _instance;
	}

	// constructor original pasado a privado
	private WebDataBaseSingleton()
	{
		totalConsultas = 0;
		// Habilita el servicio de datos del móvil
		// Trata de conectar a través de Internet con la BD de
películas
		// Si todo va bien crea una conexión remota y la mantiene
abierta.
	}


	// resto de la clase original: area de datos y métodos


	// datos y metodos originales
	private WebDataBase wdb;

	public void buscarPeliculas(String criterios){
		for(String titulo: wdb.consulta(criterios))
			System.out.println(titulo);
	}

}
```