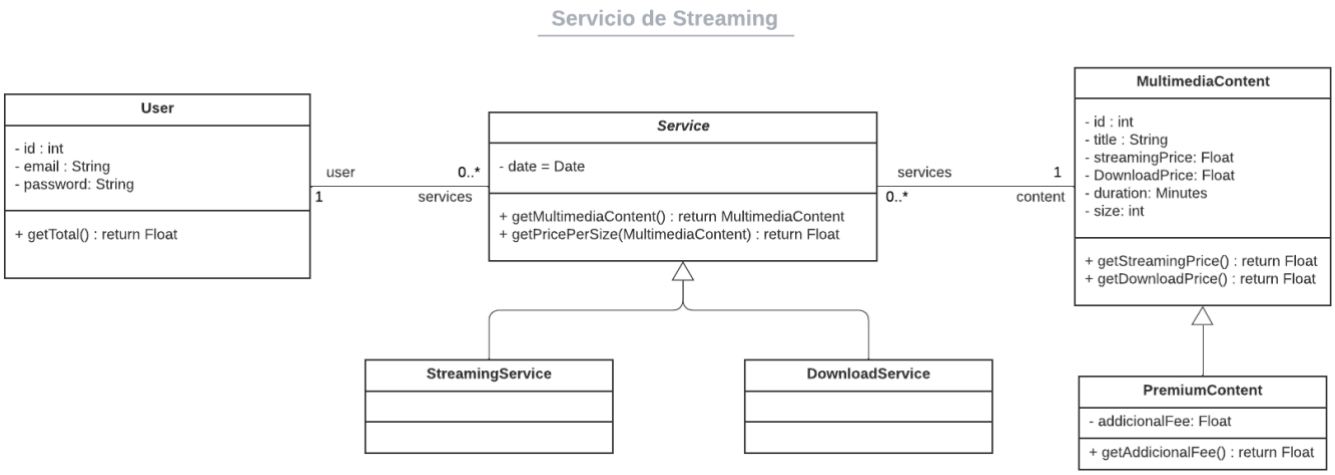
Principios de diseño

Práctica 1: Servicio de Streaming

Nos han contratado para desarrollar un software para gestionar el streaming de películas online. Disponemos de un análisis previo que podemos utilizar como punto de partida, pero que habrá que corregir y mejorar:



Como podemos ver en el diagrama de partida, los usuarios tienen una operación que devuelve el importe total pagado por todos los servicios que han solicitado. Los usuarios pueden contratar el servicio que deseen de cualquier contenido, escogiendo si lo quieren vía Streaming o vía Download.

Ejemplo: Un usuario quiere tener la película de Avatar 2 en sus dispositivos, ya que se va de viaje y necesita ver el contenido en local. Con lo cual, contratará un servicio Download del contenido multimedia Avatar 2. Además, como la película es muy reciente, está catalogada como un contenido premium, así pues, además de su coste de descarga tendrá un cargo adicional por premium.

A continuación, se muestra la implementación del método del usuario registrado que devuelve el total a pagar por todos los servicios que ha contratado.

```
public class User {

	private List<Service> services;

	public float getTotal() {

		float total = 0.0;
		foreach (Service s in services) {

			MultimediaContent mc = s.getMultimediaContent();

			if(typeof(s)==StreamingService) {
				total += mc.getStreamingPrice();
			}else if(typeof(s)==DownloadService){
				total += mc.getDownloadPrice();
			}

			if (typeof(mc)==PremiumContent) {
				total += mc.getAdditionalFee();
			}
		 }
		return total;
	 }
}
```

Ejercicio 1

Revisando el diagrama UML y el código del método getTotal() nos preocupa que su diseño sea un poco frágil y esté violando algunos principios de diseño. Nos planteamos las siguientes preguntas:

¿Este diseño satisface los siguientes principios de diseño? Justifica tu respuesta.

a) Ley de Demeter: No, esta solución no satisface la Ley de Demeter, ya que la clase User, en el método getTotal, utiliza la clase MultimediaContent, que no tiene directamente asociada. Es decir, obtiene instancias de MultimediaContent que no tiene asociadas, invoca métodos de estas y, por tanto, habla con desconocidos. En consecuencia, cualquier cambio en esta lógica propia de los contenidos multimedia, afectaría al código de User.

b) Abierto-Cerrado: No, esta solución viola el principio de diseño Abierto-Cerrado, ya que la clase User es consciente de las subclases de Service e invoca un método u otro según la subclase a la que pertenece el objeto de cada iteración. Si más adelante, ampliamos el sistema para admitir una nueva subclase de Service deberíamos modificar el código de User para obtener el precio correspondiente dentro de un nuevo else-if. Además, también se viola al tratar de manera distinta si es premiumContent o no, de tal manera que la clase User también es consciente de las subclases que heredan de MultimendiaContent.

c) Sustitución de Liskov: Esta solución si satisface la sustitución de Liskov. A pesar de que se comprueba si el servicio es download o streaming, podemos asegurar que las subclases DownloadService y StreamingService pueden trabajar como Service sin alterar el programa. En definitiva, en un bucle de Services podemos llamar a cualquier método sin importar que instancia de subclase sea (StreamingService o DownloadService)

d) Responsabilidad Única: Esta solución no satisface el principio de responsabilidad única. Service está calculando un método que no le corresponde. No es responsabilidad de Service calcular el getPricePerSize() ya que: ¿Quién dispone de la información del precio?, ¿Quién dispone del tamaño del archivo? MultimediaContent, de tal manera que parece lógico que si MultimediaContent dispone de todos esos datos sea ella quien se responsabilice de ese cálculo.

Ejercicio 2

Propón una solución a los errores que has encontrado en el primer ejercicio. Realiza los cambios que consideres oportuno tanto en el UML como en el código. Puedes generar código de otras clases si lo ves necesario.

```
public class User {
	private List<Service> services;
	public float getTotal() {
		float total = 0.0;
		foreach (Service s in services) total +=s.getServiceFee();
		return total;
	}
}

public abstract class Service {
	private MultimediaContent content;
	abstract float getServiceFee() {
	}
}

public class StreamingService extends Service {
	@Override
	public float getServiceFee(){
		return content.getStreamingPrice() +
		content.getAdditionalCharges();
	}
}

public class DownloadService extends Service {
	@Override
	public float getServiceFee(){
		return content.getDownloadPrice() +
		content.getAdditionalCharges();
	}
}
```

```
public class MultimediaContent {
	public float getDownloadPrice() {
		return downloadPrice;
	}
	public float getStreamingPrice() {
		return streamingPrice;
	}
	public float getAdditionalCharges() {
		return 0.0;
	}
	public float getPricePerSize(){
		return price/size;
	}
}

public class PremiumContent extends MultimediaContent {
	private float additionalFee;
	@Override
	public float getAdditionalCharges() {
		return additionalFee;
	}
}
```