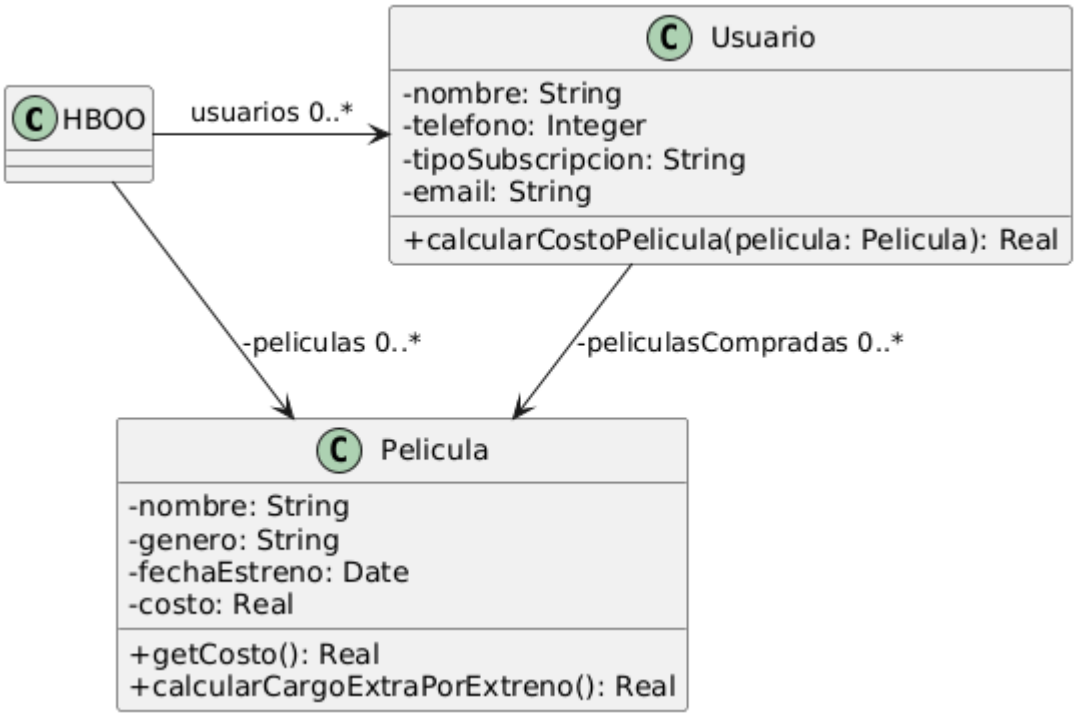
Ejercicio 2.6 Películas




```
public class Usuario {
    String tipoSubscripcion;
    // ...

    public void setTipoSubscripcion(String unTipo) {
     this.tipoSubscripcion = unTipo;
    }

    public double calcularCostoPelicula(Pelicula pelicula) {
     double costo = 0;
     if (tipoSubscripcion=="Basico") {
         costo = pelicula.getCosto() + pelicula.calcularCargoExtraPorEstreno();
     }
     else if (tipoSubscripcion== "Familia") {
         costo = (pelicula.getCosto() + pelicula.calcularCargoExtraPorEstreno()) *
     }
     else if (tipoSubscripcion=="Plus") {
```

```
        costo = pelicula.getCosto();
     }
     else if (tipoSubscripcion=="Premium") {
        costo = pelicula.getCosto() * 0.75;
     }
     return costo;
    }
}

public class Pelicula {
    LocalDate fechaEstreno;
    // ...

    public double getCosto() {
     return this.costo;
    }

    public double calcularCargoExtraPorEstreno(){
    // Si la Película se estrenó 30 días antes de la fecha actual, retorna un cargo
    // caso contrario, retorna un cargo extra de 300$
    return (ChronoUnit.DAYS.between(this.fechaEstreno, LocalDate.now()) ) > 3
    }
}
```

Paso uno: Análisis

- Se está utilizando una variable tipo String para decidir cuanto cobrar por una película. Esto huele mal tanto por el método largo como por el switch statement. Acá pueden haber dos opciones creo:
    - Resolver el problema usando herencia: Crear una jerarquía de subclases Usuario que cuenten con uno de los cuatro planes (UsuarioBasico, UsuarioFamilia...). Cada subclase se encargaría de como calcular el costo de la película por su cuenta.
    - Resolver el problema usando composición: Crear una interfaz Subscripción y varias clases (SubscripcionBasica, SubscripcionFamilia...) que implementen dicha interfaz. Cada clase se encargaría de ver como calcular el costo de la película.

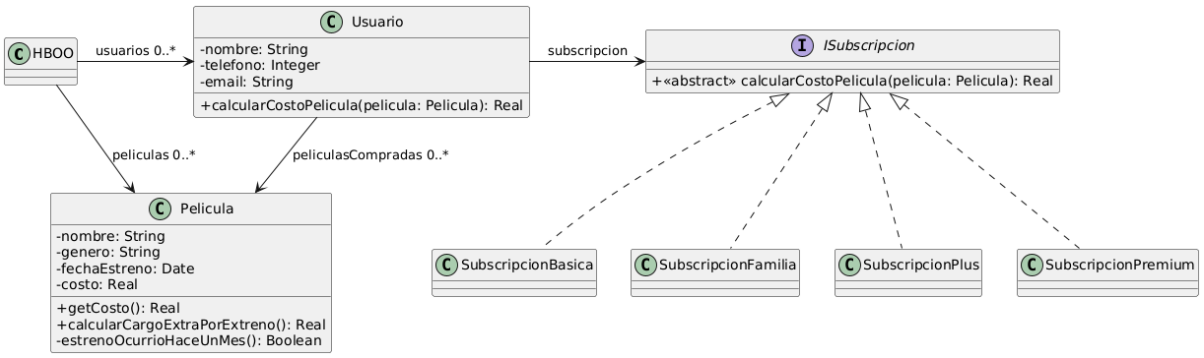
- Creo que la segunda opción tiene mas sentido porque la relación Subscripcion tiene mas pinta de ser una relación “tiene-un” que una relación “es-un”, las subscripciones se podrían cambiar fácilmente, y de paso se ahorra romper encapsulamiento.
- En el método **calcularCostoPelicula(Pelicula pelicula)** además se repite dos veces la siguiente sentencia “*pelicula.getCosto() + pelicula.calcularCargoExtraPorEstreno()*”. Esto huele a Envidia de Atributos y se podría simplificar creando un método en la clase **Pelicula** que haga lo mismo.
- En el método **calcularCargoExtraPorEstreno()** hay un comentario que clarifica lo que hace la cadena de mensajes que devuelve el valor. Se podría descomponer

Paso dos: Refactoring

1. **Code Smell:** Switch Statement → Se cobra, de una u otra forma, una película en base a lo que diga una variable local
    a. **Refactoring a aplicar:** Replace Type Code with State/Strategy → Crear una interfaz ISubscripcion y varias clases que representen diferentes tipos de subscripciones e implementen esta interfaz. Crear un método abstracto *calcularCostoPelicula(Pelicula pelicula)* y colocar en cada clase el código que le correspondía en el Switch Statement. A continuación, reemplazar “*String tipoSubscripcion*” por “*ISubscripcion subscripcion*”, borrar el Switch Statement del método en la clase **Usuario**, y en su lugar retornar la llamada del método de la subscripción.
2. **Code Smell:** Envidia de Atributos → El método calcularCostoPelicula() usa dos métodos de la clase Película para calcular el costo total de la película
    a. **Refactoring a aplicar:** Extract Function → Crear un método calcularCostoTotal() que contenga el código extraido
    b. **Refactoring a aplicar:** Move Function → Ubicar el método en la clase Película y cambiar las referencias que hagan falta
3. **Code Smell:** Comentarios → El método **calcularCargoExtraPorEstreno()** utiliza un comentario para explicar que retorna

a. **Refactoring a aplicar:** Extract Function → Crear un método *estrenoOcurrioHaceUnMes()* y meter dentro la condición del operador ternario. Reemplazar el código original por una referencia al método

Resultado final




```
public class Usuario {
    String nombre;
    int telefono;
    String email;
    ISubscripcion subscripcion;
    List<Pelicula> peliculasCompradas;

    public double calcularCostoPelicula(Pelicula pelicula) {
        return subscripcion.calcularCostoPelicula(pelicula);
    }
}

public class Pelicula {
    LocalDate fechaEstreno;
    //...

    public double getCosto() {
        return this.costo;
    }

    public double calcularCargoExtraPorEstreno(){
        return (estrenoOcurrioHaceUnMes() ? 0 : 300);
    }
```

```
    double calcularCostoTotal() {
        return getCosto() + calcularCargoExtraPorEstreno();
    }

    private boolean estrenoOcurrioHaceUnMes() {
        return (ChronoUnit.DAYS.between(this.fechaEstreno, LocalDate.now()) >
    }
}

public interface ISubscripcion {
    public double calcularCostoPelicula(Pelicula pelicula);
}

public class SubscripcionBasica implements ISubscripcion {
    public double calcularCostoPelicula(Pelicula pelicula) {
        return pelicula.calcularCostoTotal();
    }
}

public class SubscripcionFamilia implements ISubscripcion {
    public double calcularCostoPelicula(Pelicula pelicula) {
        return pelicula.calcularCostoTotal() * 0.90;
    }
}

public class SubscripcionPlus implements ISubscripcion {
    public double calcularCostoPelicula(Pelicula pelicula) {
        return pelicula.getCosto();
    }
}

public class SubscripcionPremium implements ISubscripcion {
    public double calcularCostoPelicula(Pelicula pelicula) {
        return pelicula.getCosto() * 0.75;
    }
}
```