Ejercicio F

En la clase ComputeFee hay muchos comentarios y, además el método finalPrice tiene más de 10 líneas de código lo cual no lo hace muy clean y las variables no son significativas.

Como algunas condiciones no se entendían mucho como esta `minutesToPrice > 0;` así que para resolverlo lo extraje en un método a parte con un nombre más significativo. También algunas variables como d, t y p tampoco eran demasiados significativas así que lo mejor fue renombrarlas. A su vez, algunas porciones de código también las extraje a un método a parte como, por ejemplo:

```
private int feeInOrder() {
    for (Fee fee : feesSet) {
        if (minutesToPrice / fee.getTimeFraction() > 0) {
            int units = minutesToPrice / fee.getTimeFraction();
            price += units * fee.getFractionPrice();
            minutesToPrice -= units * fee.getTimeFraction();
        }
    }
    return minutesToPrice;
}
```

Como en la nueva clase ShowAtributes debía usar 2 métodos que refactorice en la clase ComputeFee lo mejor fue hacer una clase a parte llamada Ordering con dichos métodos y que ambas clases usen los métodos directamente desde la clase ya que es una clase estática

```
protected static void sortAscending(List<Fee> feesSet) {
    feesSet.sort(new Comparator<Fee>() {
        @Override
        public int compare(Fee fee1, Fee fee2) {
            return fee1.getTimeFraction() - fee2.getTimeFraction();
        }
    });
}

protected static void sortDescending(List<Fee> feesSet) {
    feesSet.sort(new Comparator<Fee>() {
        @Override
        public int compare(Fee fee1, Fee fee2) {
            return fee2.getTimeFraction() - fee1.getTimeFraction();
        }
    });
}
```

A su vez, el método addFees() me pareció que no estaba muy bien diseñado así que lo mejor fue hacerlo directamente desde cero. Al hacer eso tuve que eliminar la clase OtherFee ya que como lo había implementado no era necesaria

También a la hora de aplicar los descuentos la mejor forma de hacerlo fue en la clase Fee donde quedo de la siguiente manera:

```
public float getFractionPrice() {
    float resultFinal = fractionPrice;
    for (Discount discount : discountList)
        resultFinal = discount.applyDiscount(resultFinal);
    return resultFinal;
}
```

El otro principio SOLID que no se cumple es Open Closed ya que si se quisieran agregar descuentos a más tarifas se tendría que modificar el código y eso no respetaría los principios SOLID. Esto lo solucionaría creando una interfaz Discount y la clase DiscountImpl donde implementa esa interfaz, de esta manera si se quisieran agregar otros tipos de descuentos en un fututo no se tendría que modificar el código.

Como tampoco se cumplía el principio de Single Responsability ya que además de calcular las tarifas también se encarga de mostrar por pantalla los atributos, el cual lo solucione creando una nueva clase ShowAtributes en el cual traslado esos métodos a esas clases.