



# **Evitando Código Prodedural: Double Dispatching**

Alicia Díaz

`alicia.diaz@lifa.info.unlp.edu.ar`

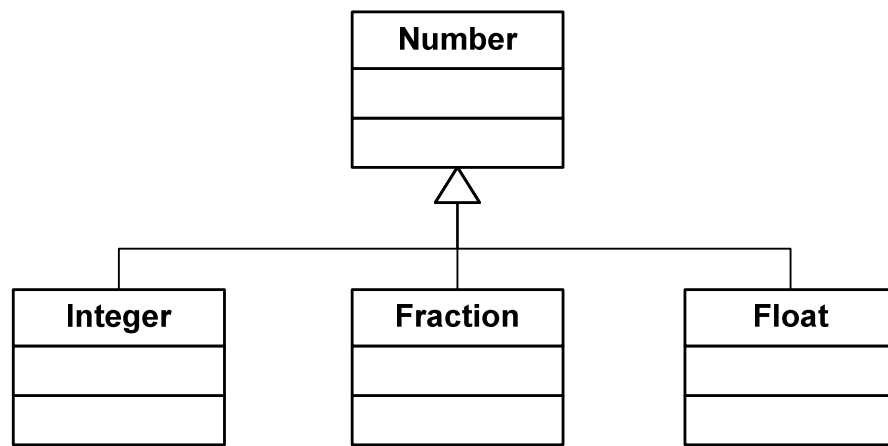
# Contexto

$$-3/5 + 14/27$$

**Fraction**>>**+** aFraction

```
^Fraction numerator: numerator * aFraction denominator +  
    (aFraction numerator * denominator)  
denominator: denominator * aFraction denominator
```

**Fraction**>>**+** aNumber



## Solución procedural

- Código que recupera información para luego tomar una decisión en base a esa información
- En los lenguajes procedurales se usan *if-then-else* o sentencias *switch*

# Solución procedural en Smalltalk

**Fraction>>+ aNumber**

aNumber isFraction

```
ifTrue:[^Fraction numerator: numerator * aNumber denominator +  
                                         (aNumber numerator * denominator)  
      denominator: denominator * aNumber denominator  
      ]
```

aNumber isInteger

```
ifTrue:[^Fraction numerator: aNumber * denominator + numerator  
      denominator: denominator  
      ]
```

aNumber isFloat

```
ifTrue:[ ??????  
      ]
```

.....

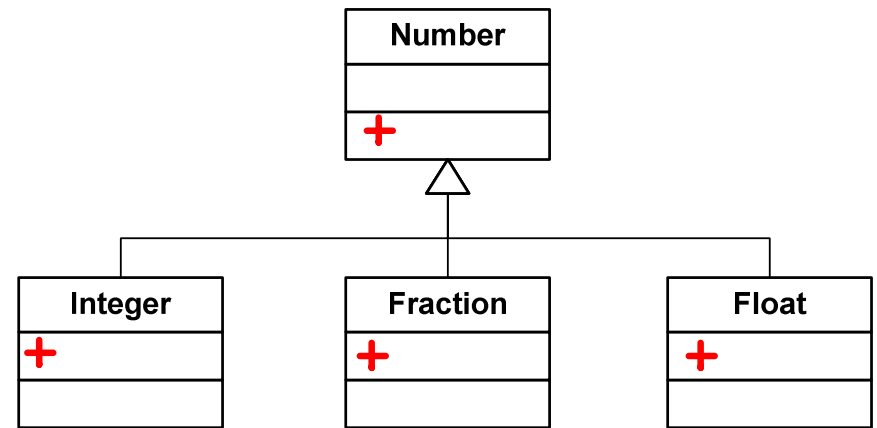
## Observaciones:

- El código depende del parámetro!!!!
  - Si en OO fuera conveniente escribir este código .....
- ¿Porqué Smalltalk no implementa una sentencia Switch?

# Solución OO clásica

## Usando polimorfismo

- Simplemente diciéndole al objeto, del cuál depende el cómputo, que lo haga el mismo
- Confiar en que cada objeto implementa ese cómputo de acuerdo a sus propias características
- Dar a los objetos responsabilidades y hacer que sean responsables de ejecutar la acción



# Solución OO con Polímorfismo

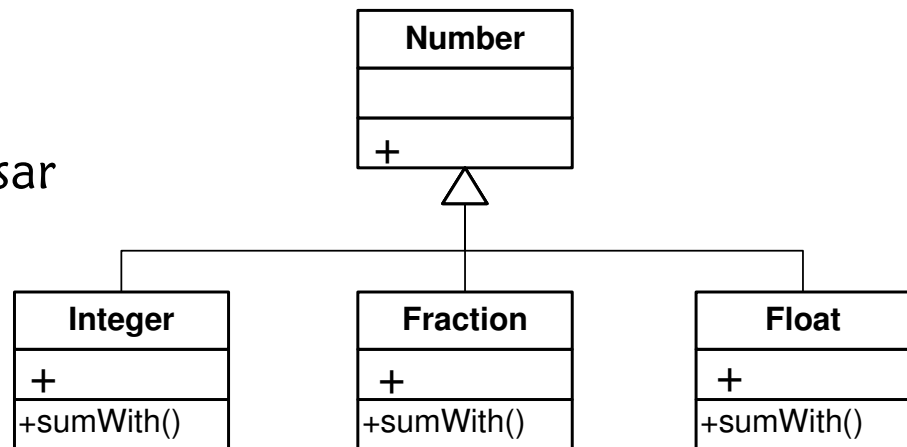
- 1º versión

```
Fraction>>+ aNumber
```

```
aNumber sumWith: self
```

- El **self** hace falta porque muchas veces se necesita pasar el contexto

.... Sin embargo esto no funciona...



**Number** no sabría de que clase es el parámetro del **#sumWith**

# Solución OO con Double Dispatching

- Double Dispatching

- El primer objeto le dice al segundo objeto que le diga al primer objeto qué hacer

**Fraction>>+ aNumber**

**aNumber sumFromFraction: self**



# Double Dispatching (1)

- 2º versión

```
Fraction>>+ aNumber
```

```
    aNumber sumFromFraction: self
```

```
Integer>>+ aNumber
```

```
    aNumber sumFromInteger: self
```

```
Float>>+ aNumber
```

```
    aNumber sumFromFloat: self
```

## Double Dispatching (2)

- En la clase **Integer**

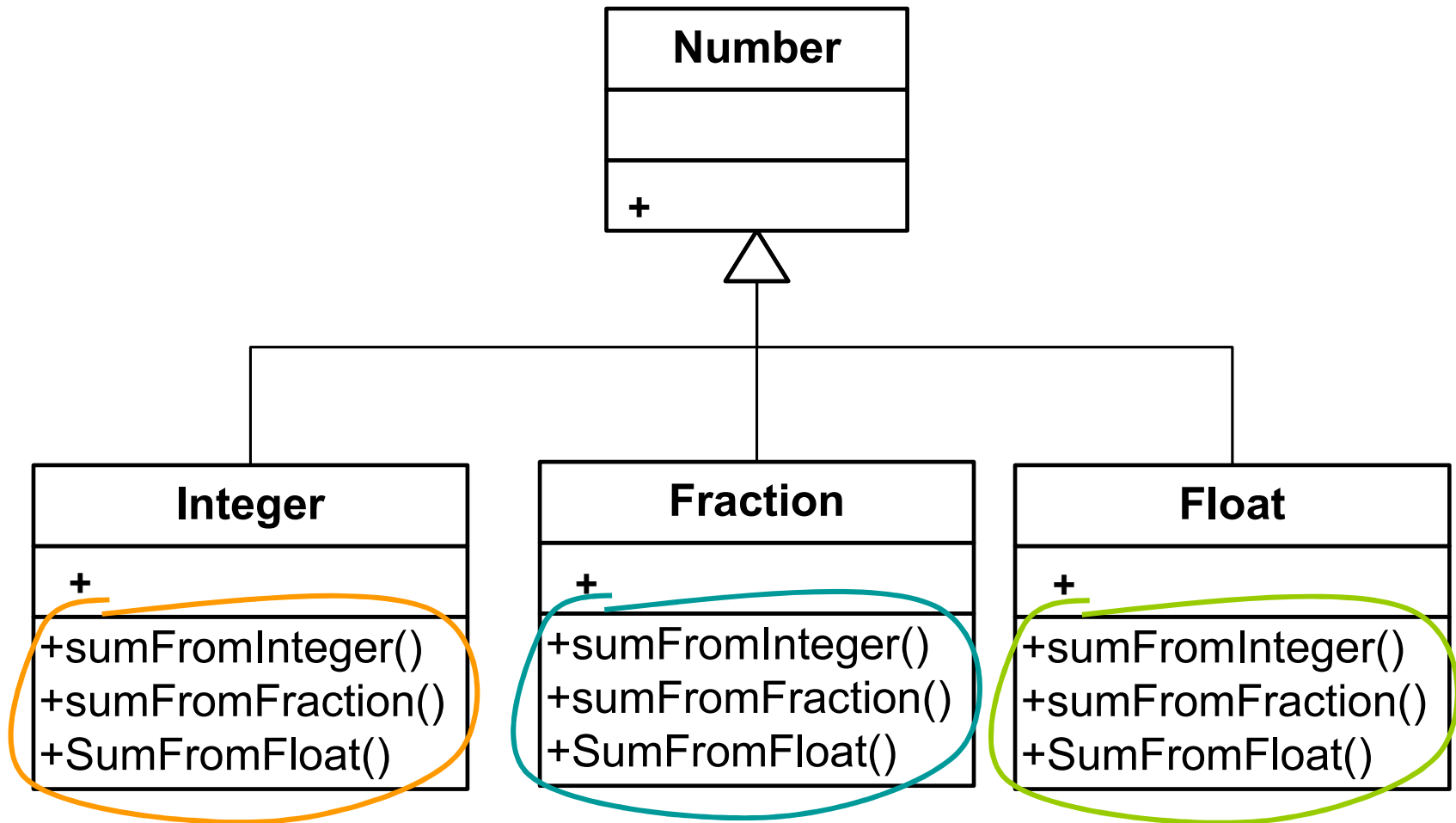
```
sumFromFraction: aFraction
    ^Fraction
        numerator: aFraction numerator +
                    (self * aFraction denominator)
        denominator: aFraction denominator
```


```
sumFromFloat: aFloat
    ^aFloat + self asFloat
```

```
sumFromInteger: aNumber
```

```
..... .
```

## Double Dispatching (3)





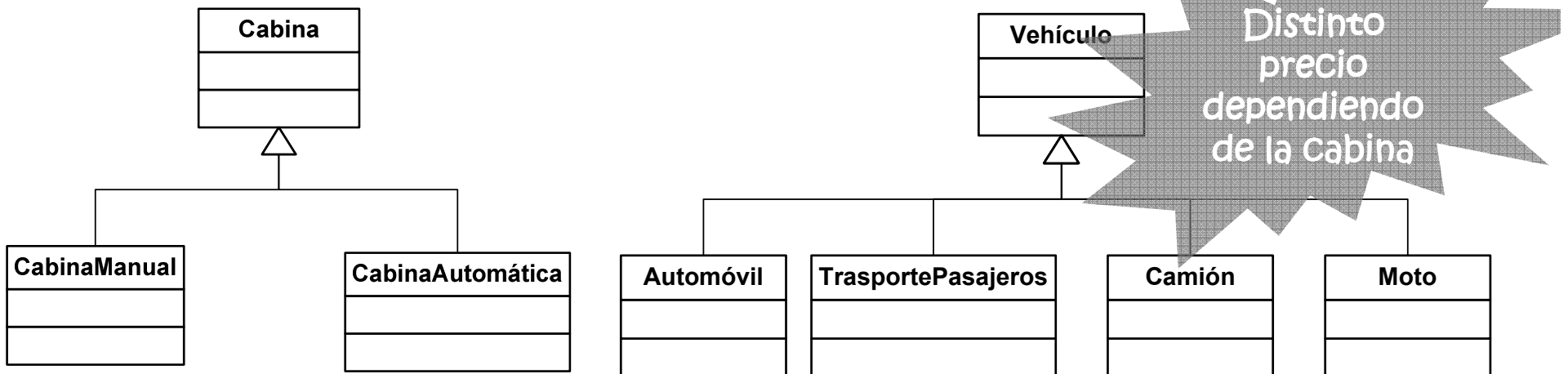
OTRO EJEMPLO

Cabinas de peaje

problema: ¿cuánto se debe cobrar de peaje?

**Cabina>>montoACobrarA: unVehículo**

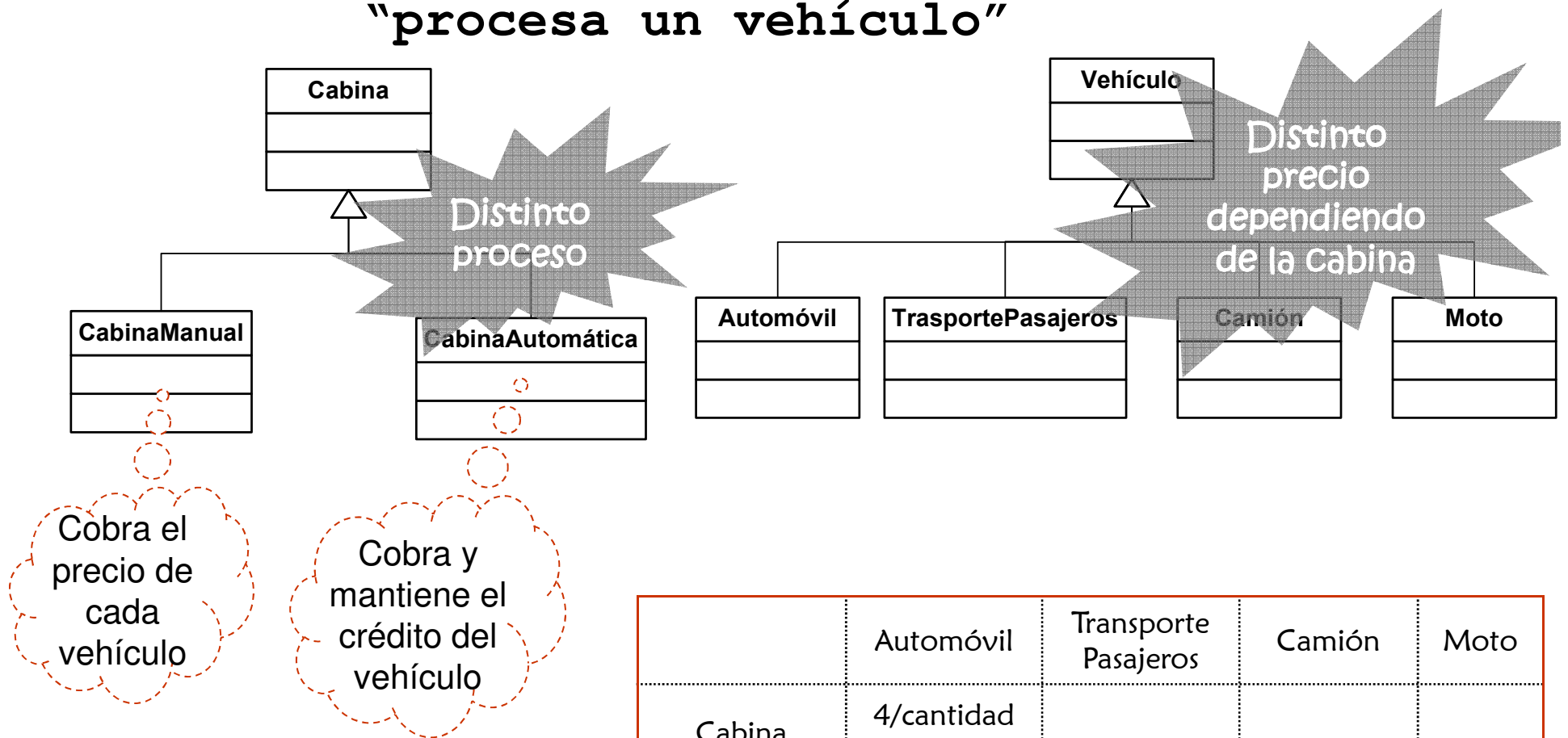
**"calcula el monto a cobrar a unVehículo"**



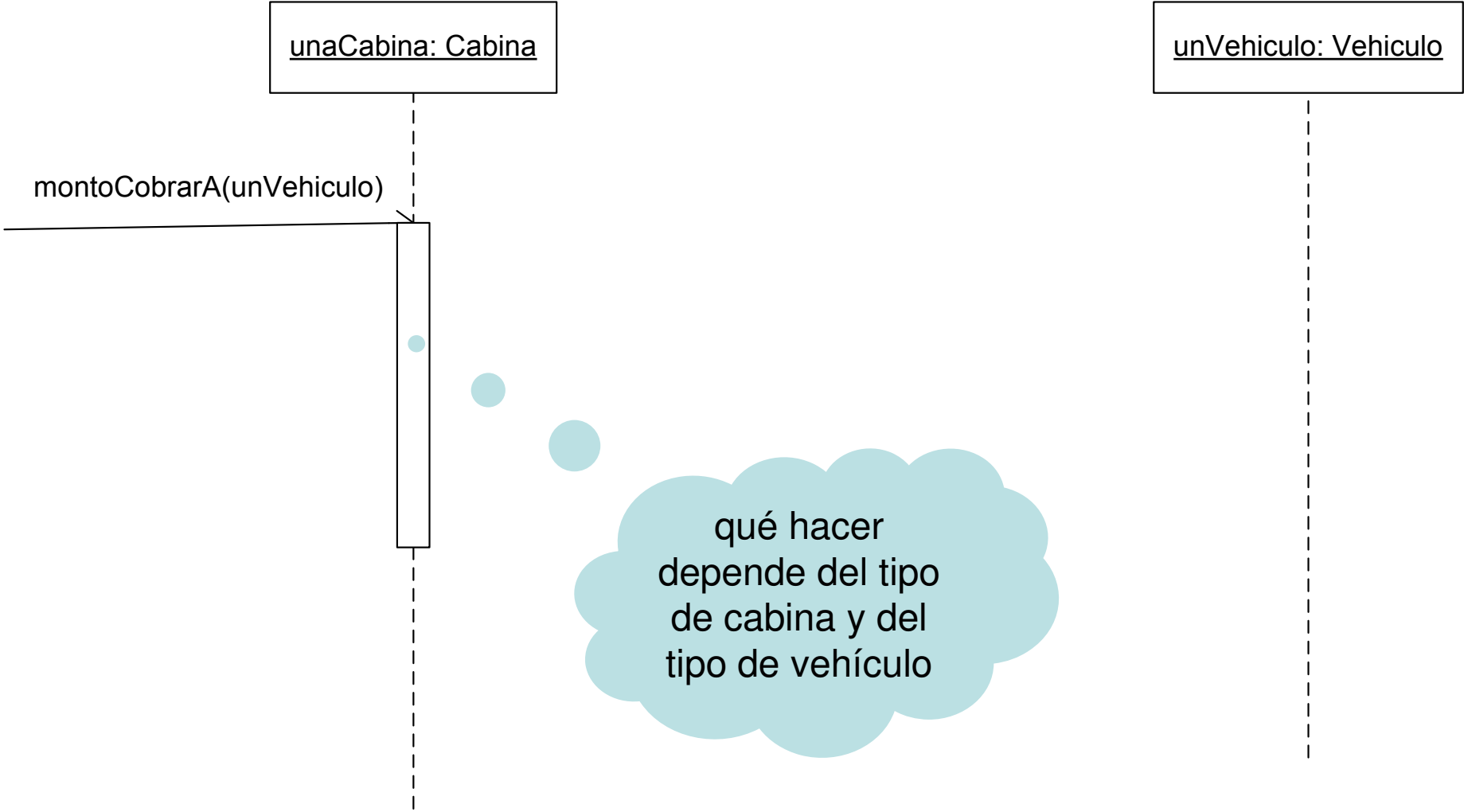
problema: ¿cuánto se debe cobrar de peaje?

**Cabina>>montoACobrarA:unVehículo**

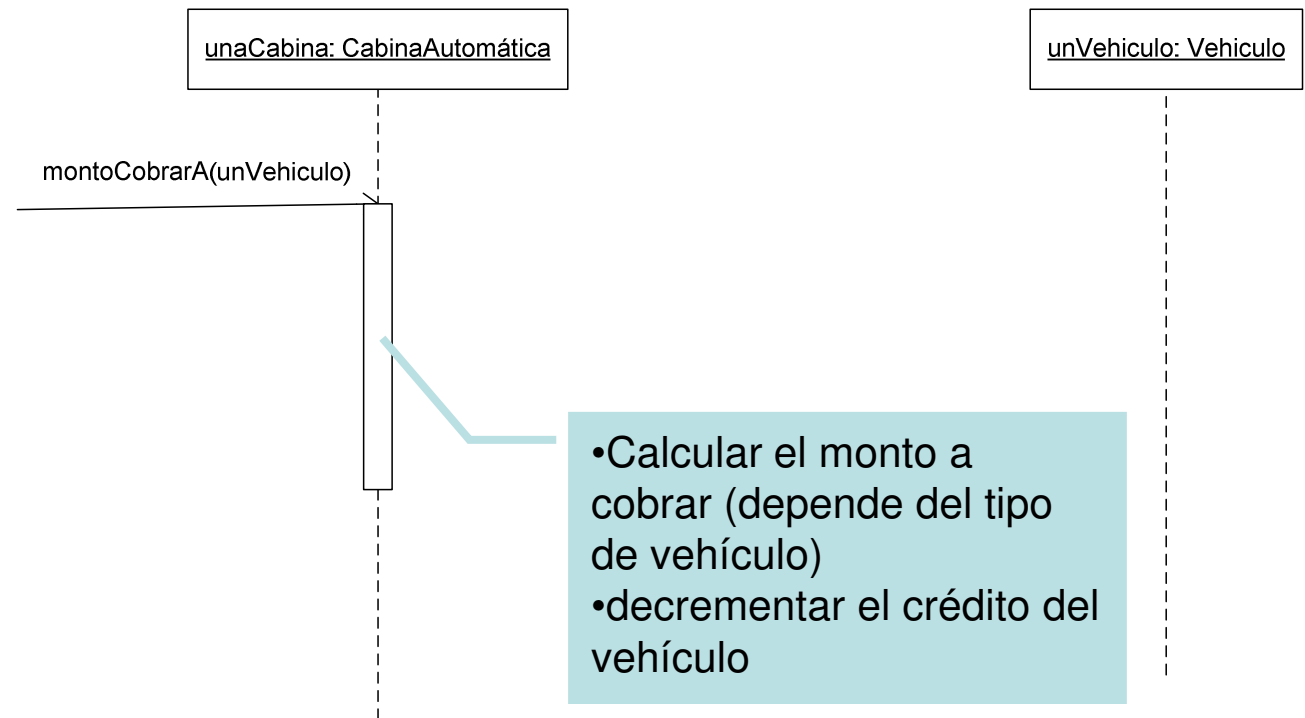
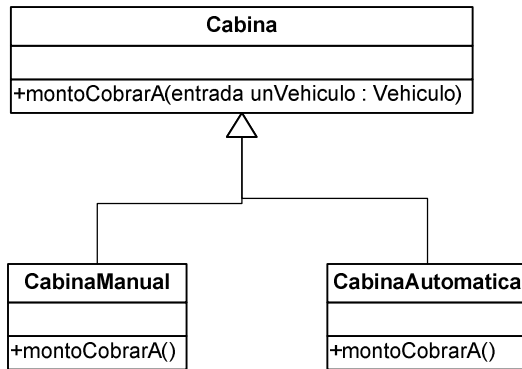
**"procesa un vehículo"**



	Automóvil	Transporte Pasajeros	Camión	Moto
Cabina manual	4/cantidad de integrantes	11.10	6.30	1.15
Cabina automática	\$1.90	-7%	idem	1.00

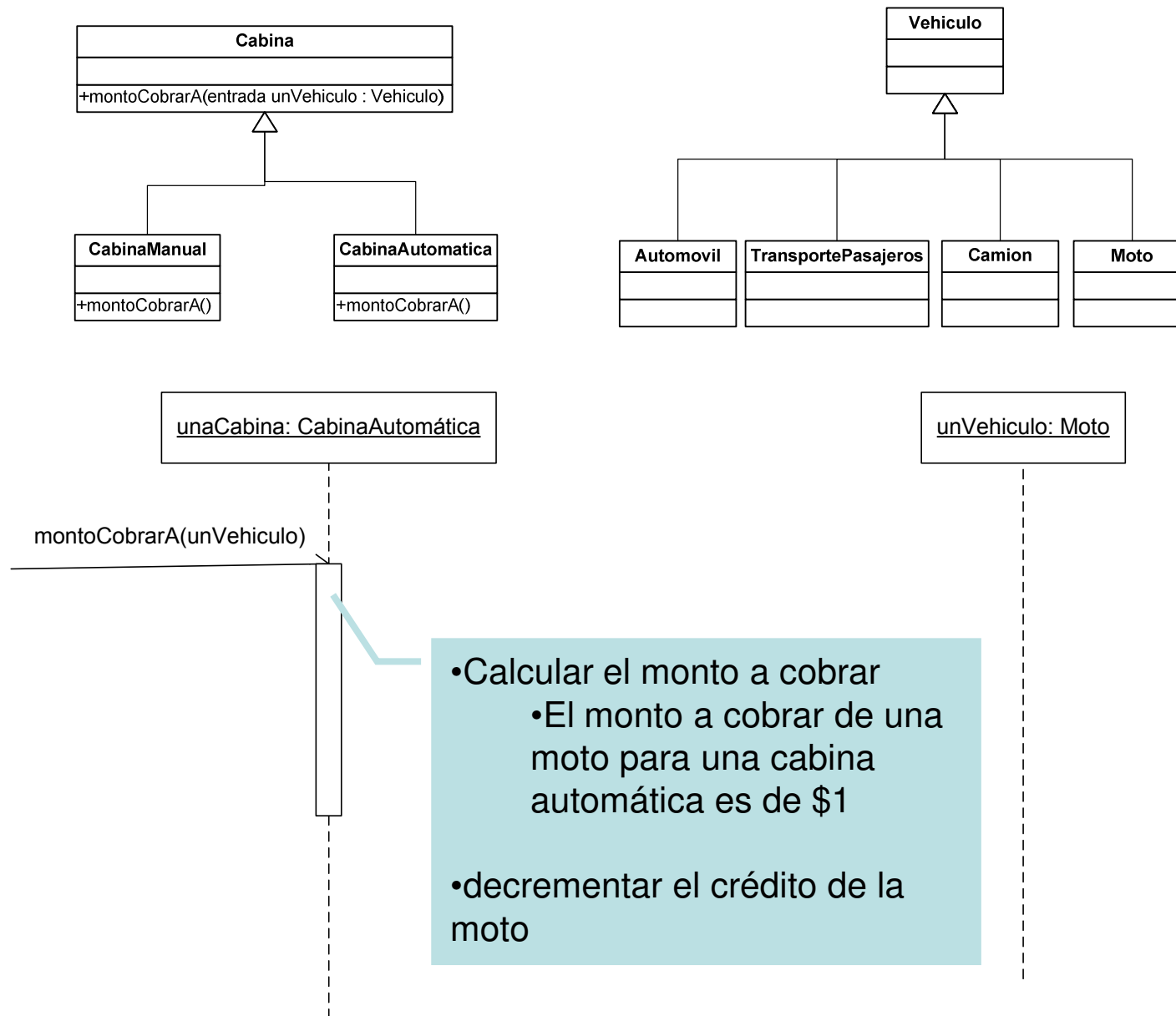


# Cuando es Cabina Automática

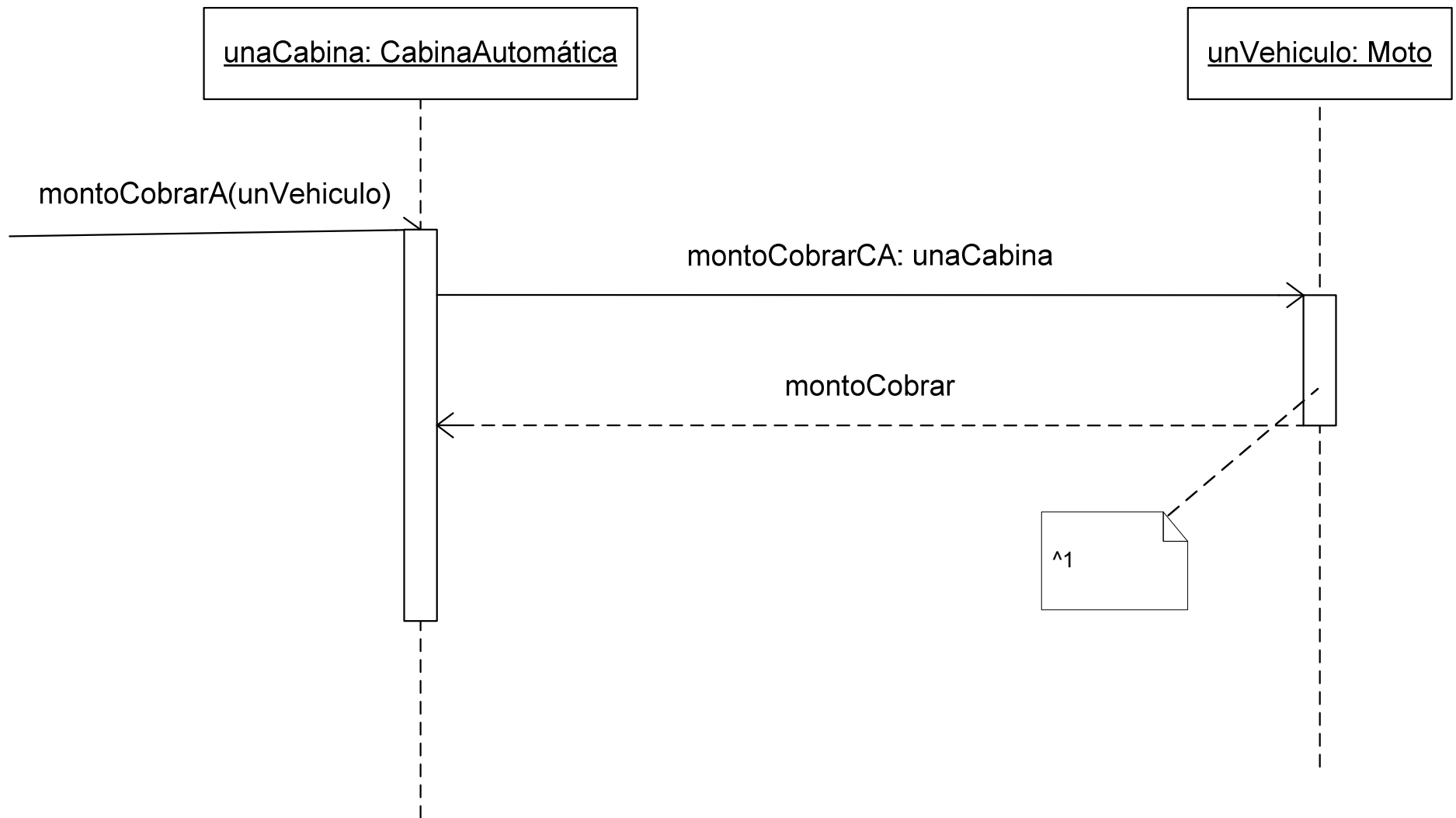




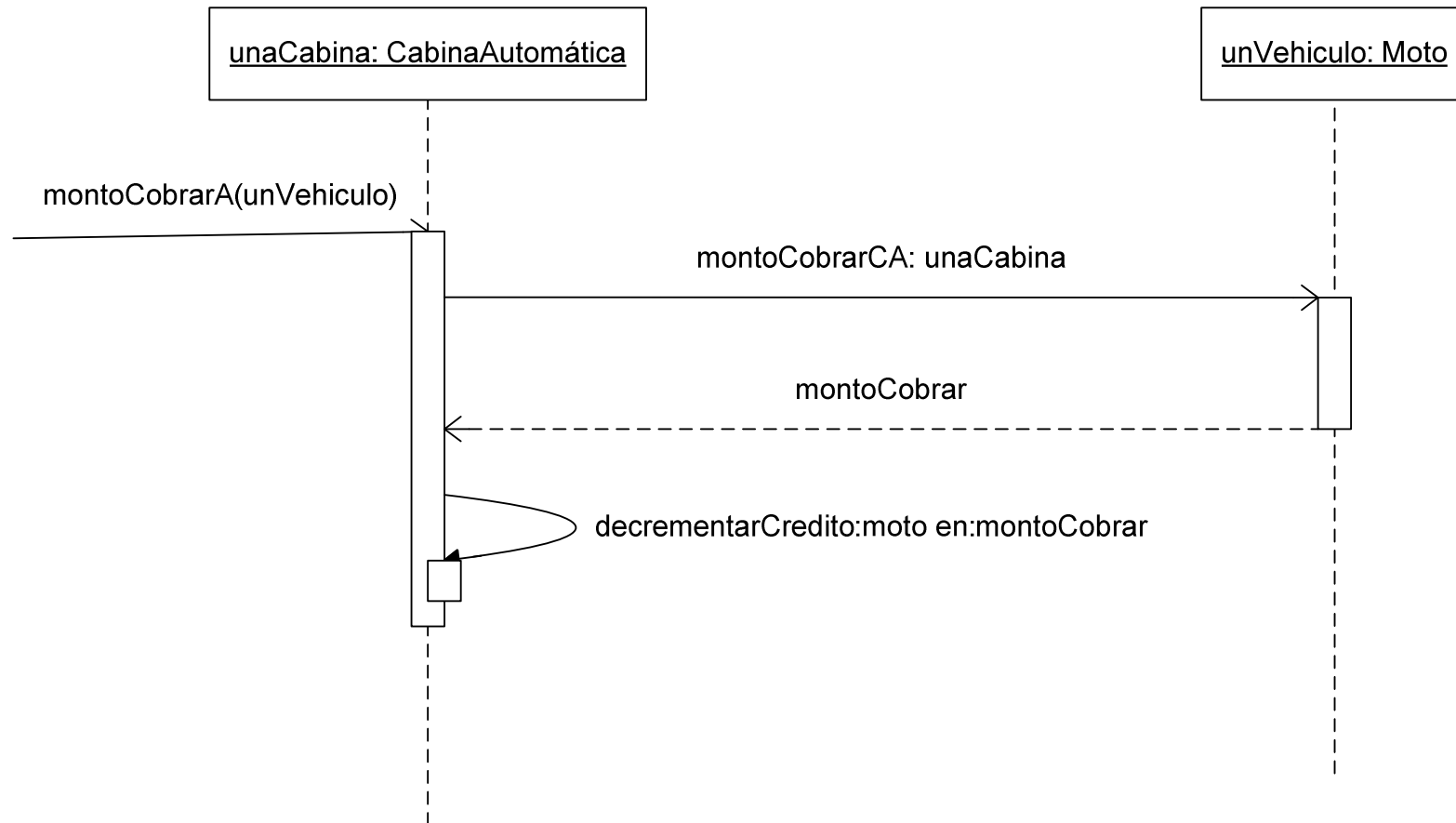
# Cuando es Cabina Automática y el vehículo es Moto



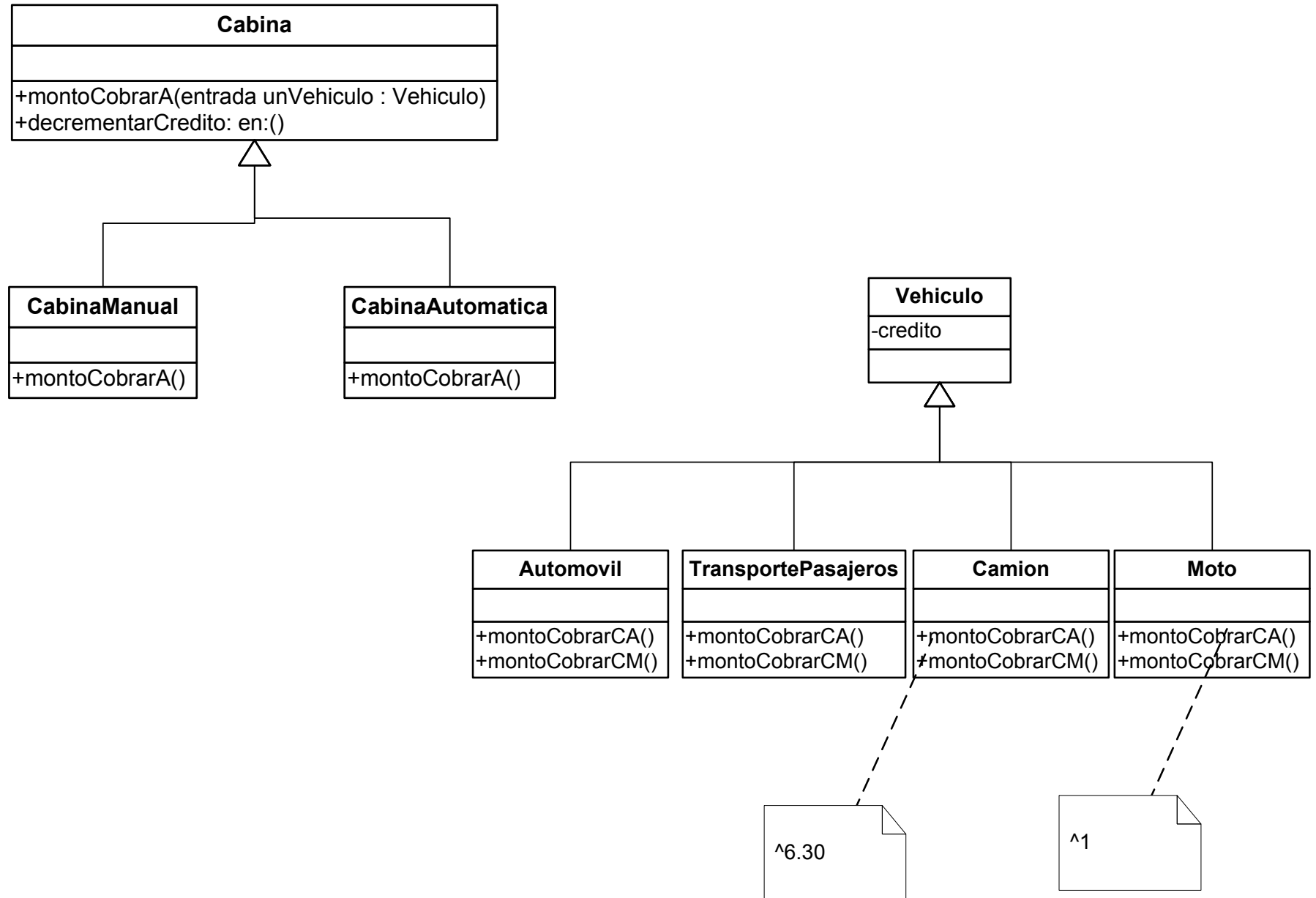
# Cuando es Cabina Automática y el vehículo es Moto



# Cuando es Cabina Automática y el vehículo es Moto



# Diagrama de Clases completo



## Ejercicio

1. Implementar todos los métodos
2. Resolver el mensaje  
**#decrementarCredito**

NOTA: Si hiciera falta completar el diagrama de clases