1 Price Monitor (5 points)

Apply the observer pattern presented in the lecture to implement price monitors for products. A monitor should track if a product is a bargain. A product is considered as a bargain by the monitor if its price falls below a given threshold. If the price rises again above the threshold, it's no longer considered as a bargain.

Define interfaces `Observer` and `Product` and implementing classes `PriceMonitor` and `Car`. Write a unit test that checks the correct behavior of three price monitors with different thresholds monitoring a single car.

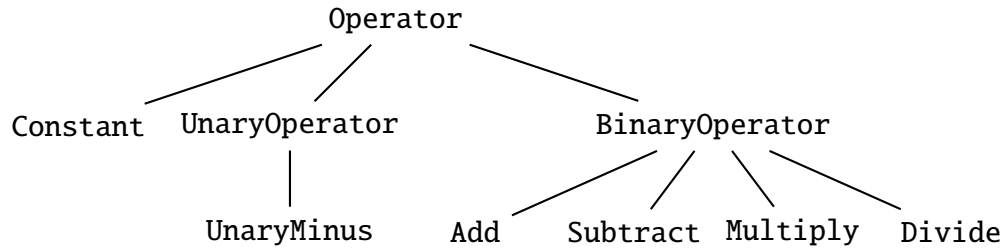
2 Operators for Arithmetic Expressions (4 points)

We want to implement a simple calculator that can parse and evaluate arithmetic expressions like the following:

```
7 * (3 * -3) - 8 * (7.5 + 6)
```

All the constants can be arbitrary floating point numbers, the available operators are addition, subtraction, multiplication, division, and unary minus.

Expressions are modeled as nested `Operator` instances, where `Operator` is an interface at the top of the following class hierarchy:



The Java files for `Operator`, `UnaryOperator` and `BinaryOperator` classes are provided in https://www.dfki.de/~steffen/advanced-java/calculator.zip
As you can see from the code template, `Operator` defines a method `double evaluate()` that needs to be implemented in the subclasses. A `UnaryOperator` has one internal `Operator` field, and `BinaryOperator` two, that are inherited to the subclasses.

When evaluating an `Operator`, `evaluate` has to be recursively called on the sub-`Operator`s (if any) and the results have to be combined depending on the concrete subclass. Note that the constructor of `Constant` must take a floating point number in form of a string as argument!

Implement the missing operators and write a unit test that manually constructs a (nested) operator for the arithmetic expression above. Test that evaluating the operator returns the correct result.

3 Operator Factory (6 points)

Manually creating nested operators is not efficient, so we provide a parser for creating them from a string (`Parser.java` in the provided archive). This parser is then used in the `Calculator` class (also provided).

The parser requires an `OperatorFactory` with a single static method `getOperator(String op, Operator ... args)`, that returns an object of the appropriate `Operator` subclass depending on the `op` string (`"+"`, `"-"`, `"*"`, `"/"`) and the number of arguments.

`Constant` is special in that there are no `args`, and the number representation is directly in the `op` string, which has to be turned into a double (for example in the constructor of `Constant`).

Implement the required `OperatorFactory` and write unit tests that check that `OperatorFactory`'s `getOperator` method produces the correct operators and that they evaluate to the correct results. Also write unit test for the provided `Calculator`.