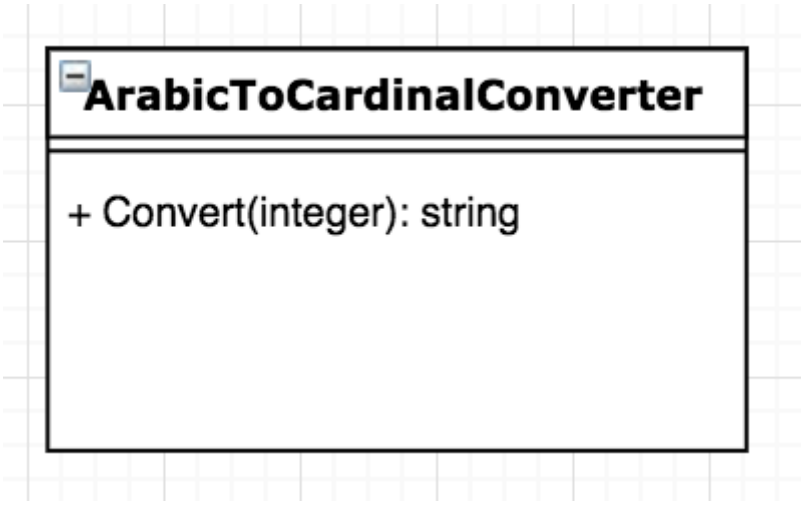
Let’s suppose you work for a client that wants to convert Arabic Numerals (such as 1, 2, 3…) to Cardinal Numbers (such as One, Two, Three).

Without thinking too much about it, you have an integer as an input and you need a method that consumes this integer and transform it into a string. You’ll maybe end up with something like that:



A superpower class!! What do we expect from this superpower class?

- We give it a number and it returns with a string.
- The class also knows all the possible results.
- Maybe we want to get to some quotients and remainders of divisions, so it does math operations too.
- It also does the conversion itself.

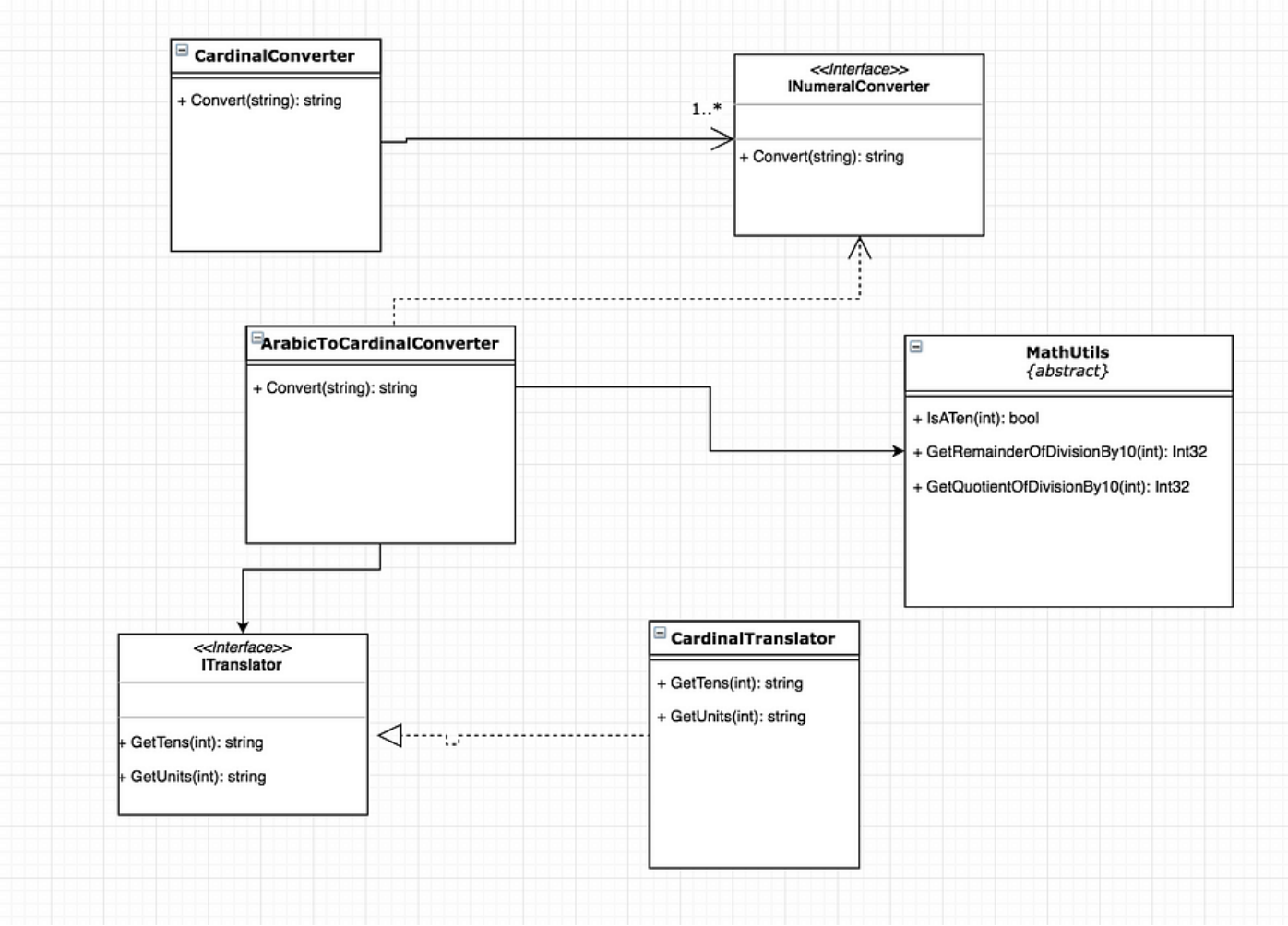
But….

What if the customer decides that now they want to convert Roman to Cardinal Numbers? What if they want the solution to be released in Portuguese??

In this case, you will definitely have to REWRITE all of your code!

But...

If you have had applied SOLID principles, your solution may be look like this:



Applying the SRP (Single Responsibility Principle), each class has its own responsibility. Who knows the cardinal numbers is the class that implements ITranslator. Who does the math operations is the static class MathUtils. And the class that implements INumeralConverter (in this case ArabicToCardinalConverter) does the conversion itself.

Defining the scope of each class, you are also applying the OCP (Opened Closed Principle). Unless you find a bug, you don't have any reason to make changes in the classes.

LSP (Liskov Substitution Principle) isn't applicable to this case, but if you have a subclass of ArabicToCardinalConverter, an instance of ArabicToCardinalConverter should be replaceable by this subclass.

When you implement the conversion between Roman and Cardinal numbers, you may not need to use the same Math operations that Arabic to Cardinal number does. That's why the only method in the INumeralConverter interface is the Convert itself. Applying ISP (Interface Segregation Principle), you don't force the classes to implement unnecessary methods.

Using the interfaces, you make sure that everything depends on abstraction, and you're also applying DIP (Dependency Inversion Principle). The main class (CardinalConverter) doesn't need to know how the conversion is performed. ArabicToCardinalConverter doesn't know the string list result and doesn't matter if it is English or Portuguese. All the class wants to know is that there will be a class implementing ITranslator and that's enough.

Going back to the problems regarding changes in this solution, if the customer asks you to convert Roman to Cardinal number, what you need to do is create a new class that implements INumeralConverter. And if you want to translate the cardinal numbers in another language, you have just to create a new class that implements ITranslator. Simple like that!