Code Reuse

Background

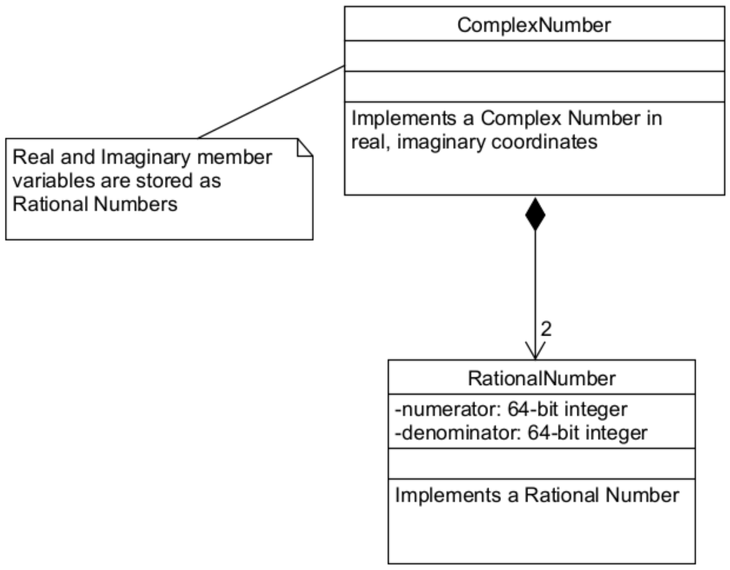
Previously you wrote two classes, one to implement a **`ComplexNumber`** and one to implement a **`RationalNumber`**. In this assignment you will demonstrate the utility of object-oriented programming, especially the concept of encapsulation and code reuse. Programs that use the **`ComplexNumber`** class should not be aware of the change.

Assignment

In your ComplexNumber class, replace the member variables of type **`double`** with variables of type **`RationalNumber`**.

E.g. **`double re; double im;`** becomes **`RationalNumber re; RationalNumber im;`**

An abbreviated class diagram is shown here:


ComplexNumber
Implements a Complex Number in real, imaginary coordinates
Real and Imaginary member variables are stored as Rational Numbers
2
RationalNumber
-numerator: 64-bit integer
-denominator: 64-bit integer
Implements a Rational Number

Your **`getReal()`** and **`getImag()`** should still return **`double`** types. That is, when **`getReal()`** or **`getImag()`** are called they should convert the **`RationalNumber`** to a **`double`** by dividing numerator by denominator.

Your **`setReal()`** and **`setImag()`** should still receive type **`double`** as the argument and convert it to a **`RationalNumber`** to a specified precision for storage. (The precision conversion constant will be stored as a **`private: const double`** in the class declaration. Set this value to store 5 significant digits for each value.) Your overload constructor will behave similarly (still received type `double` arguments and converts them to `RationalNumber`.)

Any functions involving the taking of a square root will covert the `RationalNumber` to a `double` (by division) then convert back to `RationalNumber` (abiding by the precision value) for storage.

Your `toString()` function should return a string the same as before. E.g. "double + double I"

Upon completion you will test the new code using ***EXACTLY*** the same test program you used for the ComplexNumber class. Do not change any code in the main function.

Notes

- This is a C++ assignment
- Maintain 5 digits of precision
- In your RationalNumber class you should declare your numerator and denominator as type `long long` or `int64_t`
- You will need to add an assignment operator overload to your ComplexNumber class similar to this

```
ComplexNumber& operator=(const ComplexNumber& rhs)
{
	this->r.setNumerator(rhs.r.getNumerator());
	this->r.setDenominator(rhs.r.getDenominator());
	this->i.setNumerator(rhs.i.getNumerator());
	this->i.setDenominator(rhs.i.getDenominator());
	return *this;
}
```

Deliverables

- All source code files
- An essay describing
  - successes
  - difficulties
  - changes you had to make to your **`ComplexNumber`** class
  - changes you had to make to your **`RationalNumber`** class
  - screen shots of results from the original `ComplexNumber` test (using `double`) and the new `ComplexNumber` test (using `RationalNumber`). The results should match to the specified precision value