Bonds – Yield Calculation Mechanism

Prepared by Cosmin Budac

Contents

Objective

This document describes the design behind the Yield Calculation mechanism.

Business Needs

1. A client should be able to calculate yields not necessarily the exact type of the bond.
2. The client and the bond classes should not be altered as they are owned by different teams[1]
3. Calculation might be invoked from several places[2]
4. Calculation will change frequently and new instruments (bond types) will continue to be added. Impact on existing code should be minimal.

The Solution

Considering the business needs we are looking at a solution in which the algorithm is separated from the information carrier. Also to limit the impact on the client, we will have to put in place some sort of factory that will be able to instantiate the algorithm specific to a certain bond type.

In this design, I consider that the implementation of the factory is the most interesting part. There are two main strategies in dealing with this problem:

- instantiation of algorithm classes based on the type of the bond, either through conventions or through type checking
- using a variation of the visitor pattern (based on static polymorphism). While this solution might be seen as bending the 2nd business need, it is nevertheless the one that will be used.

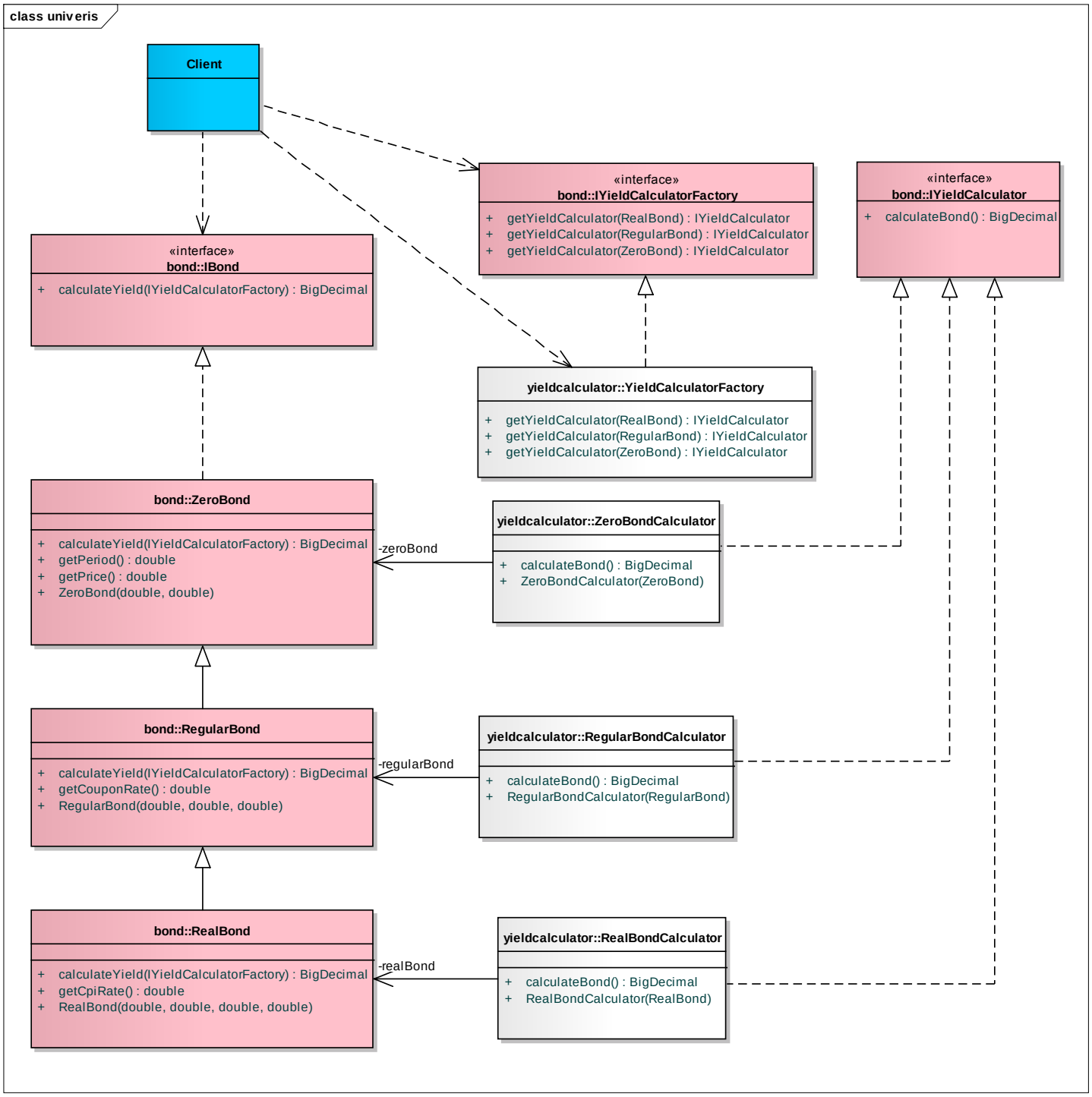
In a nutshell, these are the responsibilities of objects involved in the yield calculation:

1. the bond object accepts a visitor – the yield calculator factory (YCF)
2. the YCF, based on the type of the object that is visiting, will instantiate the proper Yield Calculator (YC)
3. the YC will be used to calculate the yield

The main elements of the solution are presented in the following class diagram:

[1] Should this statement be understood as “you should not touch the bond classes at all” or should it be “bond classes should not be touched when algorithms are changed or new instruments are added? However the design will consider both cases.

[2] I am failing to understand what the author envisioned with this statement. The design and solution will simply ignore it for now.



Characteristics of the solution

1. Each class defining a financial instrument type should implement the IBond interface and provide the concrete implementation of the “calculateYield” method. This method is practically the port through which an algorithm is associated with a bond.

```
public interface IBond {
    BigDecimal calculateYield (IYieldCalculatorFactory calculatorFactory);
}
```

```
public class ZeroBond implements IBond {

    @Override
    public BigDecimal calculateYield(IYieldCalculatorFactory calculatorFactory) {
        return calculatorFactory.getYieldCalculator(this).calculateBond();
    }
```

2. The implementation IYealdCalculatorFactory contains factory methods specific to every bond type:

```
public interface IYieldCalculatorFactory {
    IYieldCalculator getYieldCalculator(RealBond bond);
    IYieldCalculator getYieldCalculator(RegularBond bond);
    IYieldCalculator getYieldCalculator(ZeroBond bond);
}
```

```
public class YieldCalculatorFactory implements IYieldCalculatorFactory {

    @Override
    public IYieldCalculator getYieldCalculator(RealBond bond) {
        return new RealBondCalculator(bond);
    }

    @Override
    public IYieldCalculator getYieldCalculator(RegularBond bond) {
        return new RegularBondCalculator(bond);
    }
```

As a result, **adding a new bond** type requires the following operations:

- The team owning the bond code will create the new bond and update the IYieldCalculatorFactory interface.
- The new bond library is then passed to the team in charge of the calculation algorithms to provide the implementations.
- The client – with no code change - will use both libraries to calculate the yields

However, just modifying an algorithm is simpler, involving only the algorithm team and the client – to add the new libraries.

Calculating yields for different types of bonds

The following code sample shows the responsibilities of the client code:

```
IYieldCalculatorFactory ycf = new YieldCalculatorFactory();
for (IBond bond : bonds) {
    System.out.println(bond.calculateYield(ycf));

}
```

This sample shows that:

- The client can calculate any bond yield without knowing the type of bond it handles.
- The client code is totally insulated from algorithm changes or introduction of new bond types.