So we thought we knew money

Ying Hu
vhu@customhouse.com

Sam Peng
speng@customhouse.com

Here at Custom House Global Foreign Exchange, we have built and are continuing to evolve a system that processes online foreign exchange transactions. “Currency”, “money”, “rate” and “markup” are terms frequently used in conversations between business stakeholders. They are part of the Glossary and appear in almost every requirement document.

Not surprisingly, they show up in code quite frequently too, like this:

```
public class Contract
{
      decimal tradingRate;
      bool isRateDirect;
      string tradeCurrency;
      decimal tradeAmount;
      string settlementCurrency;
      decimal settlementAmount;
      decimal Markup;
}
```

(Code example 1)

In the example, rate, currency and markup are represented as decimal, string, decimal respectively. These important terms appear in the code, but only in variable names of primitive types and sometime in method names whose arguments and return types are all primitives.

The Problem

A primitive type does not capture a domain concept. Take rate, for example. A *rate* is *the ratio at which a unit of one currency may be exchanged for a unit of another currency*. A decimal 1.12 is meaningless unless we know it is the ratio between USD and CAD. And even that is not enough. We must also know the rate “direction” (is it a rate to convert USD to CAD, or to convert CAD to USD) before we know how to apply it (multiply or divide?). Consequently, business logic was duplicated in several classes that calculate amount from rate.

The Contract class provides a typical case. It contained a method to sort out these variations on rate:

```
decimal CalculateSettlementAmount(
      bool isRateDirect,
      decimal tradeAmount,
      decimal tradingRate)
{
      if (isRateDirect)
            return RoundAmount(tradeAmount * tradingRate);
      else
            return RoundAmount(tradeAmount / tradingRate);
}
                                                (Code example 2)
```

The Contract class then calculated settlement amount by calling:

```
decimal settlementAmount = CalculateSettlementAmount(
                  true, 200m, 1.12m);
```

Markup had the same duplication, and also had no explicit definition within the design. In foreign exchange, a *markup* is *an amount or a percentage added to the market rate to determine the rate being offered to customer.* You can think of this as similar to the profit a grocery store makes selling apples: They buy apples for $2/lb wholesale and sell them to customers at $2.50, a 25% markup. Markups on currency exchange rates are not so large, but if it costs 1.2 US Dollar to buy one Euro on the market and then manage it, then it might typically be marked up 2%, resulting in a price to customers of 1.224 USD. Applying a markup to a rate is not straightforward, it depends on

1. markup type: whether it's percentage markup, or a points markup, and
2. markup direction: whether you want to markup a rate that sells Euro to customer, or to markup a rate that buys Euro from customer.
3. markup algorithm: what formula should be used, whether there are exceptional cases or not .

This application logic was expressed by the following code:

```
decimal ApplyMarkup(
            decimal rate, decimal markup,
             bool isPercentageMarkup, bool isBuyMarkup)
{
      if (isBuyMarkup)
      {
            if (isPercentageMarkup)
                  return rate* (1-markup);
            else
                  return rate-markup;
      }
      else
      {
```

```
                if (isPercentageMarkup)
                      return rate* (1+markup);
                else
                      return rate+markup;
          }
     }
```

(Code example 3)

When a contract used a markup, it called:

```
     decimal newRate = ApplyMarkup(1.12, 0.02, true, false);
```

(Code example 4)

The above code doesn't show an explicit markup concept, only a messy function that manipulates numbers.

As for currency, we did, in fact, have a currency object, but it was not used consistently throughout the code. In a lot of cases currency was simply a string. Finally, there was no object to represent the money concept, but we saw that currency and amount *always* appeared together.

The Discovery

In early 2005, our team worked with Eric Evans, of Domain Language, Inc., to improve our domain model and design over a period of months. Early on, Eric pointed out a problem he thought must be addressed right away: the excessive use of primitive types, which indicated that a significant part of the language of the domain was only implicit in the model, leaving a big hole in the ubiquitous language, leading to bulky, inflexible object code.

The team spent several white board sessions discussing the business domain, important terms used by business and what they mean in our code. For some terms, we found discrepancies between the business language and the development language; for others, there was no explicit representation in the code at all.

From this process, a few essential value objects emerged and were refined, of which we consider Currency, Money, Rate, and Markup to be the most important.

Currency: a *unit of exchange*. At first glance, it seems acceptable to use a string because currency is simply a 3-letter string without much specific logic. However, consider the following code:

```
Before:
decimal GetRate(string unitCurrency, string referenceCurrency);

After:
Rate GetRate(Currency unitCurrency, Currency referenceCurrency);
```

The second method clearly tells its callers that it expects two "Currency" parameters. Even though currency can be represented by a string, a string is by no means a currency!

There is also a finite set of valid currencies allowed in the system; a factory method easily ensures the validation:

```
Currency gbp = CurrencyList.Get("GBP")
```

Money: *An amount of a particular currency*. The object's major responsibility is to handle common monetary arithmetic and comparison properly. In a financial system like ours, it is no coincidence that amount always appears side-by-side with currency. Having a Money object greatly simplifies code, like this:

Before	After

`public decimal CalculateProfit(`
`  string tradeCurrency,`
`  decimal tradeAmount,`
`  string settlementCurrency,`
`  decimal settlementAmount)`
	`public Money CalculateProfit(`
	`  Money tradeMoney,`
	`  Money settlementMoney)`

Note that the first method actually makes the assumption that the returned amount has the same currency as the settlementCurrency – logic that can only be discovered by the diligent developer who looks into the method implementation. So, not only is the new code shorter, it also helps to make hidden logic explicit.

Rate: *A conversion of money of one currency into money of another currency*. A Rate object derives one Money object from another. Rate knows which two currencies it relates, and how to calculate the conversion between them. The primary responsibility of rate is expressed through the method:

```
public Money Convert (Money money)
```

With the new smart Rate object, Contract calculates its settlement simply by calling:

```
Money settlementMoney = tradingRate.Convert(tradeMoney)
```

This works as well:

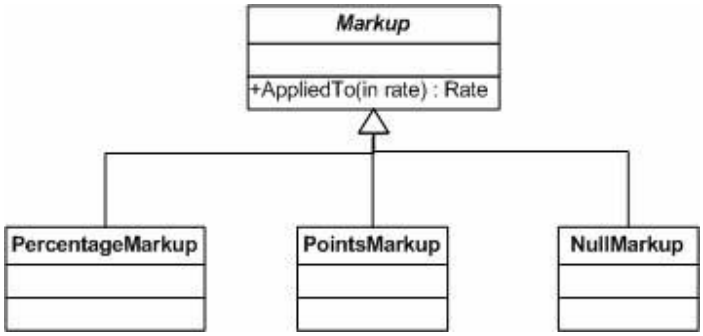
```
Money tradeMoney = tradingRate.Convert(settlementMoney)
```

Either way the calculation is stated, the Rate knows how to properly convert between the two currencies. By comparing to (Code example 2), you can see how completely calculation and rounding logic is encapsulated!

Markup: An amount of points or percentage added to wholesale cost to arrive at a resale price of rate. A Markup object can apply itself, following a certain formula, to a rate to produce a different rate. This is represented by the method on Markup:

```
public abstract Rate AppliedTo(Rate rate);
```

We designed markup as an abstract object with three subclasses (including the null object):



Once a markup object is obtained, it can take a rate and apply it's calculation logic on it. Now contract uses markup like this (compare with Code example 4)

```
Markup percentageMarkup = new PercentageMarkup(…);
Rate newRate = percentageMarkup.AppliedTo(oldRate);
```

Replacing primitive types with value objects based on fundamental domain concepts gives us other benefits too:

1. We are able to create and use Null objects, for example:

```
class NullPercentageMarkup: PercentageMarkup
{ //…}

public static readonly PercentageMarkup NULL
                = new NullPercentageMarkup();
```

2. We have a list of methods that have the nice property "Closure of Operations"

```
Rate.AppliedTo(Money) --> Money
Markup.AppliedTo(Rate) --> Rate
```

3. Those value objects are designed to be *immutable*. They are simply constructed with a constructor and are never changed except by full replacement. Calculations produce new instances. Immutable objects have well-documented advantages in designs. They mean the program can copy or share the objects freely. They mean a programmer does not need to be concerned about side-effects involving these

objects and their calculations. We found these properties helped us clarify our code and reduce bugs.

Next Step: Retrofitting

In a complex system like ours, it was a significant job to replace types that thread throughout the system and form its very heart. Here is an example modification of one of our existing domain objects:

Before

```
public class Order
{
  //...
  private decimal settlementAmount;
  private string settlementCurrency;

  public string SettlementCurrency
  {
     get { return settlementCurrency; }
  }

  public decimal SettlementAmount
  {
     get { return settlementAmount;}
   }
}

Payment payment = new Payment();
payment.TotalAmount = order.SettlementAmount;
payment.TotalCurrency = order.SettlementCurrency;
```

After

```
public class Order
{
   //...
   private decimal settlementAmount;
   private Currency settlementCurrency;

   public Money SettlementMoney
   {
      get{ return new Money(
               settlementCurrency,
               settlementAmount);
          }
    }
}

Payment payment = new Payment();
payment.Total = order.SettlementMoney;
```

The new Order class exposes SettlementMoney property, which makes its client, the Payment class, clearer and more concise.

But wait a second, the legacy settlementAmount and settlementCurrency still exist inside Order as private members. Isn't it better to replace them with Money too? Yes, ideally, all private members should be replaced too. They are still there because of the shortage of time: due to the massive usage of primitive types, we found it impractical to exhaustively replace every occurrence. As a compromise, we decided to *refactor important interfaces first*. This way, even though many of the ugly bits are still around, they are hidden within a well defined boundary.

Before	After

`public interface IRateSource`
`{`
`  decimal GetSpotProfitRate`
`  (`
`  string unitCurrency,`
`  string referenceCurrency,`
`  TradeDirection direction,`
`  string specifiedCurrency,`
`  decimal specifiedAmount`
`  );`
`}`
	`public interface IRateSource`
	`{`
	`  Rate GetSpotProfitRate`
	`  (`
	`  Currency unitCurrency,`
	`  Currency referenceCurrency,`
	`  TradeDirection direction,`
	`  Money specifiedMoney`
	`  );`
	`}`

Currency, Money, Rate and Markup - they seem straightforward enough—perhaps obvious, in hindsight. Yet we walked a long way to make them explicit with well defined responsibilities. Having these domain fundamentals represented in our model allowed our design and code to start to speak the same language as people in the business do. As these concepts work their way into the system, the habit of thinking in domain terms also gradually works its way into the team. More value objects have been designed to express the model such as BidAsk, ProfitRateQuote and PaymentMethod; more primitive types have been replaced

The Future – And a Few Pitfalls

This is only the start. These value objects are now widely used in the system. But because of their prominent expressive power, developers start to put stuff in just because of convenience. Consider the following method defined inside Rate class:

```
public Money AddMoney(Money target, Money moneyToAdd)
{
      return target + Convert(moneyToAdd);
 }
```

This method adds the “moneyToAdd” to the “target” and makes sure the resulting Money object is expressed in the target Money object’s Currency. Whoever put it in probably wanted to encapsulate the conversion between two Money objects, and perhaps some form of this function would have made sense there. However, adding money certainly is not Rate’s responsibility, and so this new function weakened Rate’s representation of the rate concept and at the same time subtly coupled it to a higher layer of the system that adds Money for a particular reason.

Another issue we deal with is the unclear message sent by the fact that a fair amount of legacy primitive types still stick around side-by-side with their value object counterparts. Instead of refactoring, some developers continue using them. They have even added operator overloading to convert value objects back to primitive types!

```
public static implicit operator decimal(Rate rate)
{
    return rate.Value;
}
```

This code allows a rate to be implicitly cast into decimal – the exact opposite of our intention in creating the value objects!

And yet, in spite of all the incompleteness and imperfection of our ongoing development work, these "domain fundamentals" have raised our level of abstraction above the level of decimals and strings, and have been one of the keys to allowing our system to continue to develop and produce rich applications. Our domain model is evolving, and so is our domain- driven design journey.