The Money Object Pattern

RUSS RUBIS, Florida Atlantic University

IONUT CARDEI, Florida Atlantic University

Money is so prevalent in our lives that we often don't think about its content or implementation from a computer science perspective. Whenever money functionality is needed in given application, it is often left up to designers and developers to implement this functionality. Within single currency systems the implementation of money is fairly simple, as just a monitory amount needs to be captured. However, with continuous and rapidly expending internationalization and globalization of enterprise applications, there is an ever increasing need to have access to reusable and extensible implementation of the Money Object and Money API which supports multi-currency environments. In this paper we present the Money Object Pattern which should be a useful tool when implementing money object in single as well as multi-currency enterprise applications.

General Terms: enterprise applications

Additional Key Words and Phrases: money object, money api, currency, business object, data object

1. INTRODUCTION

The internationalization and globalization of enterprise applications increases the need for implementation of Money as one entity rather than a collection of disjoined data pieces. Limited implementations of Money API have been recently introduced (Microsoft in 2011 as part of the Microsoft.Xrm.Sdk Namespace, and Oracle as part of the java.util package in its upcoming Java SE 9 release). One of the earliest references of Money as object or API can be found in Martin Fowler's book "Patterns of Enterprise Application Architecture", where he introduced the concept of Money as "class data type in any mainstream programming language" [4].
Our pattern will build on Martin Fowler's original Money class proposal, and will include the missing pieces.
Martin Fowler's Money object is a great start to represent money in a given application; however it does not take into account two main factors which are applicable to just about any business in any country:

1. Legally businesses are established in their "home" countries, and therefore assumed to have "home" currency as the base currency for transactions, book keeping, financial reporting and taxation. For example, if a business is registered in the USA, then its tax liabilities are expected to be in US Dollars. That is not to say that businesses cannot or do not transact in currencies other than their "home" currency.
2. Businesses can and do transact in other currencies (i.e. purchase goods and services in non-home currency), but the "home" currency is used to calculate the "actual" price of the good or service purchased for the purposes of book keeping, financial reporting and taxation record keeping.

In addition to the amount and currency, the money object should also contain other properties that make it more useful and complete. In this pattern we introduce the base currency, currency exchange rate, and currency date, and arithmetic operations which are essential in order to render the money object complete. Our design pattern is simple enough, and at the same time, detailed enough, to be found useful when implementing a money object in an enterprise application, using any language or architecture. This paper continues in section 2 with the detailed presentation of the pattern, then, in Section 3, we describe known uses. Section 4 lists related works, and we conclude in Section 5.

2. THE MONEY OBJECT PATTERN

2.1 Intent

In most enterprise applications supporting monetary activities, a base currency is necessary in order to correctly track the money coming in and the money going out (a.k.a. P&L, Profit and Loss). For example, how does a company calculate P&L when it sells one million euros worth of services, and buys five hundred thousand dollars' worth of office supplies? A Money object within a given application must not only contain an amount and a currency of transactions but also the base currency, the date, and the exchange rate used. In this paper we introduce a language neutral money object pattern which can be applied in enterprise applications to address the ever growing internationalization and globalization of businesses and their applications. Our money pattern must also be able to handle single currency transactions without any changes to the structure of the money object so that it can be used for both, single as well as multi currency transactions.
Ontologies sometimes distinguish between two major categories of entities, "continuants" (entities that have a duration, a start and an end), and "occurrents" (entities that have a timestamp, e.g. transactions). In this pattern we treat the money as an occurent, i.e. as a monetary transaction [10].

2.2 Example

A business headquartered in the United States needs to purchase a product from a French company, which sells its products in Euros. Because the US based business is headquartered in the United States, its base currency is the US dollar, and its transactions are recorded in US dollars. The US based business profit and loss is also calculated in US dollars. Thus if a US based business purchases 100 Euros worth of goods from a French company, the cost of those goods needs to be recorded in US dollars in the US business's purchasing and accounting systems. At the same time, we want to preserve the original amount (in Euros) for proper record keeping. Therefore the transaction must be recorded as follows:

Amount in foreign currency: 100.00 EUR
Amount in base currency: 130.00 USD
Exchange rate: 1 EUR = 1.3 USD
Transaction date: March 12, 2015

In the conventional/legacy application the storage and retrieval of above transaction is left to the individual developers/designers. Some might chose to store the foreign currency amount in one database table, then derive the base amount using the exchange rate derived elsewhere based on the transaction date. While others might opt for storing the base and the foreign currency amounts together, then leave the exchange rate derivation based on the two values stored.
Our pattern eliminates the guess work which is often unnecessarily performed by the developers/designers when the requirements call for a money object implementation by presenting the money object as a single unit which stores the monetary related values in one place.

2.3 Context

As businesses are becoming more global, the support for multi-currency transactions within a given business application is becoming unavoidable. You are designing a new business application which will be used to process transactions in multiple currencies. Each transaction needs to be stored as a single unit, containing the necessary information about monetary data used for each transaction. It is assumed that this application will have a base currency, which will be set based on the country where the business running the new applications is headquartered.

2.4 Problem

How can one structure a general purpose monetary object that contains a unified and complete monetary data, and that can be used for both, single currency and multi-currency transactions?

2.5 Forces

- A monetary transaction in single or multi-currency environment should be storable and retrievable as a single object or unit. Naturally, we could implement it as a container of disjoint values, but it would be much more beneficial from application design and development perspective, as well as efficiency of the storage/retrieval process if the money object is treated as a unit, and not as a container of disjoined values.

- A monetary object should be able to support single and multi-currency transactions. One could argue for the benefits and of a perceived simplicity of the single currency money object in a single currency system. But a conversion or an upgrade of such system to a multi-currency architecture, if one is ever needed, would be a daunting process. Therefore there should not be a separate or different implementation for single or multi-currency transactions. Consequently the same money object can be implemented without any modifications in applications supporting single or multi-currency business processes.
- A monetary object for a single or multi-currency unit needs to support arithmetic operations, such as addition, subtraction, multiplication, and division. Granted, these calculations can be implemented outside of the money object, but without these basic operations, the money object would not provide the full set of benefits to application designers and developers.
- A monetary object for multi-currency unit should store the date and the exchange rate of the transaction. Although the exchange rate can be derived externally for the given date, adding both the exchange rate and the transaction date to the monetary object provides transaction record as well as the means of performing mathematical calculations between monetary objects with different currencies.
- Ability to display a monetary amount in base or actual/transaction currency. If transactions were performed in multi-currency, we could always display them in one currency or the other, but it would be beneficial from a usability perspective for end users to be able to see both amounts if they so desired, or even toggle between one or the other.
- When needed, a monetary object should be capable of supporting the Net Present Value (NPV) when performing arithmetic calculations. NPV calculations can be complex, and are generally a subject of discussion in themselves, but it would add to the money object's usefulness if it could support NPV in its arithmetic calculations.

2.6 Solution

Our solution is the Money Object Pattern which stores the amount in two currencies, and supports various arithmetic operations.

The Money object contains the following attributes:

- Base currency
- Amount in base currency
- Transaction currency
- Amount in transaction currency
- Date of transaction (more precisely the date used to derive the exchange rate)
- Exchange rate (always 1 when single currency is used as there's nothing to convert the single currency to)

The base currency is the "home" currency of the application. Each application must have a base (or home) currency declared. For example, a business that is registered in the United States would have US dollar as the base (or home) currency because its tax liabilities are expected to be in US Dollars. All transactions and book keeping are expected to be performed in the base (or home) currency. If implementing in Java, java.util.Currency would be the currency type.

The amount in base currency is the amount of transaction after it is converted from the actual currency to base currency, using the value of the exchange rate. Thus a product purchased for 100 euros (EUR), would be recorded as 130 dollars (USD) in base currency, using the exchange rate of 1 EUR = 1.30 USD. The data type for amount in base currency is Double, as many currencies have fractions or precision (i.e. 2 decimal places to represent the cents amount in US dollar, or 0 for Italian Lira), however the type can also be different, depending on the language implementation, For example, in Java a friendlier type to use would be java.math.BigDecimal as it provides public methods for rounding and precision setting. Note: the precision of a currency is generally defined in the Currency object (java.util.Currency in the case of Java), and not the Money object.

Transaction currency is the currency that was used to purchase goods or services. In our money object it can be (and often is) the same as the base currency if the goods or services were purchased using the same currency as

the base currency of the system/application. If implementing in Java, java.util.Currency would be the currency type.

Amount in transaction currency is the value of purchase or transaction in the currency requested by the selling party. In above example, the amount in transaction currency would be 100 euros. The data type and fraction derivation logic is similar to that described for the base amount about, except a different currency would be used to perform the derivation.

Date of transaction is used to store the date and time of the transaction, which in turn can be used to determine the exchange rate. The time zone of the transaction date is application dependent. Some applications are configured with the time zone of the business header quarters, while others might use a GMT time zone to have a single point of reference, yet others might use strictly local time. The date type is also implementation dependent. For example, in Java one can use a java.util.Date or java.util.Calendar.

Exchange rate is the value used to convert from transaction currency to base currency. Note: when transaction currency and base currency are the same, then exchange rate is 1.

2.6.1 Implementation Details

The following is an example of the Money class implementation in Java.

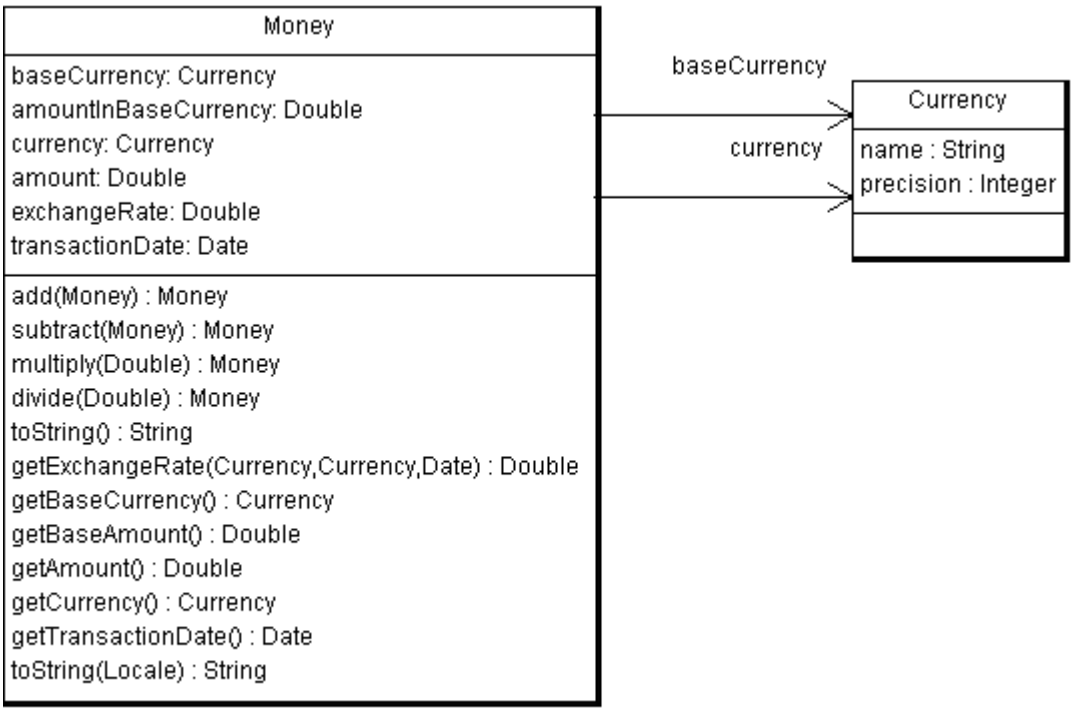

Figure 1: The Money Object UML class diagram.

For simplicity and clarity, we assume that all null checking has been performed by the Money object constructor. Also for simplicity, we avoid the use of Factory classes, which would be preferable in the actual implementation of the Money class.

There could be numerous constructors to instantiate a Money object, so we'll provide a few examples of those we feel would be most often used. Also, the base currency is generally set/configured as part of application-specific properties, and would not normally be passed as a parameter to the Money constructor. However, in the examples below we use it as a parameter for the sake of simplicity, and to improve the flexibility of the

constructors. In a real world application, we would use (in Java) Currency.getInstance(Locale.getDefault()), or some other API to obtain a base currency of the application.

```
class Money...
	//In cases where exchange rate is known…
	public Money (BigDecimal amount, Currency currency
		, Currency baseCurrency, bigDecimal exchangeRate) {
		this.amount = amount;
		this.currency = currency;
		this.baseCurrency = baseCurrency;
		this.amountInBaseCurrency = amount*exchangeRate;
		this.exchangeRate = exchangeRate;
		this.transactionDate = new Date(Date.getTime());
	}

	//In cases where both, the exchange rate and date, are known…
	public Money (BigDecimal amount, Currency currency, Currency baseCurrency
		, BigDecimal exchangeRate, Date date) {
		this.amount = amount;
		this.currency = currency;
		this.exchangeRate = exchangeRate;
		this.amountInBaseCurrency = amount*exchangeRate;
		this.baseCurrency = baseCurrency;
		this.transactionDate = date;
	}

	//In cases where we are dealing with single currency transaction
	//(i.e. base and transaction currencies are the same)…
	public Money (BigDecimal amount) {
		this.amount = amount;
		this.currency = getBaseCurrency();
		this.exchangeRate = 1;
		this.amountInBaseCurrency = amount;
		this.baseCurrency = getBaseCurrency();
		this.transactionDate = new Date(Date.getTime());
	}

	//If our application has base currency configured and has access to the latest exchange rates,
	//then the ultimate constructor would be as follows…
	public Money (BigDecimal amount, Currency currency) {
		Date date = new Date(Date.getTime());
		this.amount = amount;
		this.currency = currency;
		this.exchangeRate = getExchangerate(currency, getBaseCurrency(), date);
		this.amountInBaseCurrency = amount*exchangeRate;
		this.baseCurrency = getBaseCurrency();
		this.transactionDate = date;
	}

	//In cases where we can use some sort of web service to obtain the currency exchange rate
	//based on the date…
	public Money (BigDecimal amount, Currency currency, Currency baseCurrency, Date date) {
		this.amount = amount;
		this.currency = currency;
		this.exchangeRate = getExchangerate(currency, baseCurrency, date);
```

```
            this.amountInBaseCurrency = amount*exchangeRate;
            this.baseCurrency = baseCurrency;
            this.transactionDate = date;
    }
```

Note: getExchangerate() is assumed to be some external function (e.g. implemented by a web services) which will return the exchange rate between the 2 currencies specified and the requested date.

Once we instantiate the Money object using one of the above constructors, we can now take advantage of its public method to perform various operations on this object.

One of the main advantages of the Money object is that the arithmetic operations can be performed on Money objects with different currencies. This is easily accomplished by the use of base amount value of the two Money objects. Since the arithmetic operation is performed within a specific application, which we assume is configured with a specific locale, we can assume that the base currency of our Money objects is the same as it is derived from the locale of the application. Thus the use of the base amount is ideal for performing arithmetic operations on the Money objects.

Add and subtract are very powerful operations offered by the Money object. In a real life scenario, a business could have thousands of transactions in many different currencies. These transactions can be easily summed up simply by summing up the base amounts of all transactions.

For simplicity and clarity, we assume that all null checking has been performed by the Money object constructor.

```
class Money...
    //add one money object to another
    public Money add (Money money) {
        if (money != null) {
            this.baseAmount = this.baseAmount + money.baseAmount;
            this.amount = this.baseAmount * this.exchangeRate;
        }
        return this;
    }

    //add one money object to another
    public Money subtract (Money money) {
        if (money != null) {
            this.baseAmount = this.baseAmount - money.baseAmount;
            this.amount = this.baseAmount * this.exchangeRate;
        }
        return this;
    }

    //multiply Money by some number…
    public Money multiply (Double arg) {
        this.baseAmount = this.baseAmount * arg;
        this.amount = this.amount * arg;
        return this;
    }

    //divide Money by some number. Note, we are not checking for 0 and are letting application
    //environment handle division by 0.
    public Money divide (Double arg) {
        return multiply(1/arg);
    }
```

```
//compare Money objects
public static int compareTo (Money money1, Money money2) {
        money1.baseAmount.compareTo(money2.baseAmount);
}
```

Some helper methods to round out the base features of the Money object.

```
class Money...
        //get base currency of the application
        //this is usually application/implementation specific and can be accomplished
        //in many different ways. Here we give example of one approach.
        public static Currency getBaseCurrency () {
                return Currency.getInstance(Locale.getDefault());
        }

        //ultimately a Money object would need to be displayed to the end users,
        //and toString is just one possible way to do it in a predefined format.
        public String toString () {
                return this.currency.getSymbol()
                        + this.amount.setScale(this.currency.getDefaultFractionDigits())
                        + this.currency. getCurrencyCode();
        }
```

Numeric values, and especially money amounts, are often displayed differently in different parts of the world. For example, in North America a period is used to indicate the beginning of decimal numbers. Elsewhere, a comma is the accepted decimal numbers separator. The Money object can thus format and display the amount based on specific locale if the view layer ever requires it.

```
        public String toString (Locale local) {
                //return locale-specific string representation of this money object.
        }
```

We believe that above methods provide sufficient set of functionalities to make the Money object useful in just about any application.

2.7 Consequences

The Money Object Pattern addresses a number of shortcomings when only a number and a currency are used to represent a monetary unit. In this pattern we introduced the concept of base amount and base currency, which enables us to easily perform arithmetic operation on a money object, as well as maintain a single point of reference, even in a multi-currency system.

In addition, the Money object supports single and multi-currency application configuration/setup, and can thereby be deployed in just about any environment.

As with any currency exchanges, the exchange rate and exchange date play a crucial role in determining the relationship between different currencies. We addressed this relationship by making the exchange rate and exchange date an integral part of the Money Object Pattern.

The Money Object Pattern also provides great flexibility when it comes to presentation of the Money object. By maintaining the base and actual currency information within the object itself, we are able to display the object in different currencies, depending on the needs of the application employing the Money object.

This pattern does not take into account the NPV (Net Present Value) when describing arithmetic operations of the money object. The NPV determines what interest rate to apply, and is not necessarily the interest rate set by

Federal Reserve for US dollars (or a central bank of another country for its home currency). Companies, industries and even specific projects can have their own internal interest rates. All interest rates can change over time. Consequently, there is a factor of uncertainty when calculating the net present value. Due to a multitude of complex considerations related to NPV, a more elegant solution is to let an external component or service handle the NPV calculations. For that reason, the NPV, as it relates to the money object, is assumed to be calculated outside of the money object itself. Instead, the money object can, when needed, take the NPV into account when arithmetic calculations are performed by calling an external component or service.

3. KNOWN USES

Most major Commercial Off The Shelf (COTS) Enterprise Resource Planning (ERP) developers such as Oracle and SAP, to name the two biggest, have their own custom implementations of handling the money and multi-currency data. Ariba Inc. [9], which has recently been acquired by SAP [10] uses a custom implementation of the Money object which is very similar to that described in this paper. A good part of the pattern which we propose in this paper is the result of our study of Ariba's Money object implementation. Ariba's Money object also uses base currency, actual currency, the date, and exchange rate to record the value of a transaction.

4. RELATED PATTERNS AND FRAMEWORKS

4.1 Money, by Martin Fowler

Martin Fowler introduced the concept of a Money object in his book "Patterns of Enterprise Application Architecture" [4], in which he proposed a monetary unit in the form of an object. Our pattern extends on his original ideal that, at the very least, the money object needs to have an amount and currency, and needs to be able to handle certain arithmetic operations.

4.2 Money Class, Microsoft .NET

Microsoft introduced the Money Class to its flagship .NET framework [3]. The Money Class in Microsoft .NET is more of a data object and does not currently support arithmetic operations.

4.3 JSR 354 (Money & Currency) - Specification

The Java framework is introducing the Money support in its upcoming release 9 [1]. The proposed specification is very detailed, and includes a very large number of new objects and interfaces to support Money & Currency features in release 9. Some of the concepts such as monetary object and various arithmetic operations between two monetary objects with different currencies are covered in this pattern, and are similar to those introduced in the proposed JSR specification. The JSR specification introduces the javax.money.MonetaryAmount interface, which provides public methods for performing arithmetic operation on a money object. The main difference between the pattern in this paper and the JSR 354 is that the latter is a detailed specification of how the Money & Currency will be implemented in Java, and is less of a pattern. As such, it is much more complex than the pattern introduced in this paper.

5. CONCLUSIONS AND FUTURE WORK

The Money Object Pattern proposed in this paper offers a simple, unified and flexible solution for handling money type data in just about any system or application. The pattern supports single and multi-currency environments, as well as arithmetic operations on money type data. Our pattern also provides alternatives for money data formatting and presentation, making it an ideal solution for User Interface and Reporting implementations. As part of our future work, we would like to address the specifics of currency conversion rates, their structure, and derivation, ultimately coming up with a new pattern that would complement the Money Object Pattern introduced in this paper. We believe that a new pattern addressing NPV (Net Present Value) calculations as part of the Money object could provide significant benefit when designing enterprise level applications.

6. ACKNOWLEDGMENTS

We would like to thank Pavel Hruby for his invaluable feedback and support during the shepherding of this paper.

7. REFERENCES

[1] Java Community Process, JSR 354: Money and Currency API, https://jcp.org/en/jsr/detail?id=354
[2] Java 1.4.2 API Documentation, Oracle, https://docs.oracle.com/javase/1.4.2/docs/api/ [3] Money Class, Dynamics CRM 2015, Microsoft .NET Library, Microsoft, https://msdn.microsoft.com/en-us/library/microsoft.xrm.sdk.money(v=crm.7).aspx
[3] Fowler M., 2002. Patterns of Enterprise Application Architecture, Addison Wesley
[4] Fowler M., 1997. Analysis Patterns: Reusable Object Models, Addison-Wesley Longman.
[5] Daum B., 2003. Modeling Business Objecs with XML Schema, Morgan Kaufmann Publishers
[6] Eriksson H.-E., Penker M., 2000, Business Modeling with UML: Business Patterns at Work, OMG Press / Wiley Computer Publishing
[7] Adams J., Koushik S., Vasudeva G., Calambos G., 2002, Patterns for e-business: A Strategy for Reuse, IBM Press
[8] Ariba Inc., 2013. DOI=http://www.ariba.com
[9] Wong K. and Bass D., May 23, 2012. SAP to Acquire Ariba for $4.3 Billion in Push Into Cloud. Bloomberg
[10] Sowa, John F., Knowledge Representation: Logical, Philosophical, and Computational Foundations, Brooks / Cole, 1999.