Arithmetic with Measurements on Dynamically-Typed Object-Oriented Languages

Hernán Wilkinson
Mercap Development Manager
Tacuarí 202, 7mo Piso
C1071AAF, Buenos Aires, Argentina
54-11-4878-1118 (ext. 120)
h.wilkinson@mercapsoftware.com

Máximo Prieto
Lifia – Facultad de Informática
Universidad Nacional de La Plata and
Universidad Nacional de la Patagonia Austral, (UNPA-UACO)
cc11, 1900, La Plata, Argentina
+54 221 422-8252 (ext. 215)
maximo.prieto@lifia.info.unlp.edu.ar

Luciano Romeo
Mercap Software Architect
Tacuarí 202, 7mo Piso
C1071AAF, Buenos Aires, Argentina
54-11-4878-1118
l.romeo@mercapsoftware.com

ABSTRACT

In physics, like in other sciences, formulas are specified using explicit measurements, that is, a number with its unit. The first step to determine the validity of a physics formula's evaluation is to verify that the unit of the result corresponds with the prospective unit. In software development, physics, financial and other sciences formulas are programmed using mathematical expressions based only on numbers, being the units of these numbers implicitly given by the semantics of the program or assumed by the programmer's knowledge. Consequently, it is common that errors result from operating with values expressed in different units, e.g., dividing a quantity of years by a quantity of months, without obtaining any type of indication or objection to this error from the system. In this report, we discuss our experience designing and implementing a model that solves this problem reifying the concept of measurement, unit and their arithmetic. Our model relieves the programmer from the arduous task of verifying the validity of the arithmetic expressions regarding units, delegating that responsibility to the system, thereby, diminishing the errors introduced by the incorrect use of values expressed in different units. We also show that having implemented this model with a dynamically typed language simplified its programming and increased its reusability.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software – *Reuse Models.* D.3.3 [**Programming Languages**]: Language Constructs and Features – *Frameworks, Polymorphism.* J.1 [**Computer Applications**]: Administrative Data Processing – *Financial.*

General Terms

Design, Reliability, Languages

Keywords

Measurements, Units, Arithmetic expressions, Dynamic typed, Smalltalk

1. INTRODUCTION

Mercap Software Corp., based in Buenos Aires, Argentina, develops financial systems. Therefore, it is essential to us the validity of the results obtained when applying financial formulas. For this reason, after struggling for several years with problems related to using formulas with incorrect parameters, we designed a model in which all arithmetic operations between measurements are automatically validated by the system when they are evaluated. This model is based on representing measurements as first class objects, that is, an object that encapsulates a number with its unit. This representation allows the programmer to use measurements in arithmetic expressions as if they were numbers, but with the advantage of providing explicit information to the system—specifically, the measurement's units.

We see this model's benefits analogous to the benefits that took place when programmers went from performing arithmetic operations using bytes in assembly language, where it was the programmer's responsibility to verify carrying problems, among others, to use more abstract types like Integer, Double or Float, where the programming language took care of representation problems. The number abstraction relieved the programmer from all the validations and verifications that she had to explicitly do with an assembly language and delegated that responsibility to the system, achieving therefore, a smaller error rate when performing arithmetic operations.

Using this measurement model and not only numbers to conduct arithmetic operations, we obtained greater results than the prospective ones in several aspects of the software construction process. Among them, we can name the sense of security that it creates for the programmer; the fact that it is the system and not she who must assure the result's validity regarding the units, therefore, diminishing the error rate produced by the incorrect evaluation of formulas or financial functions. If the programmer performs incorrect operations with measurements, then this error will not go unnoticed, and the system will inform it when evaluating a mathematical expression.

Modeling measurements as "first class objects" would have helped to solve or prevent famous errors like the incorrect use of units in the Mars Climate Orbiter [1] or during the test of the Star Wars laser-beam missile defense experiment [12]. Additionally, such an approach also aids in solving less grandiloquent, although recurrent, costly and well known errors in the scope of financial systems development.

The aim of this report is to briefly present the design and implementation of this model, and how it influences our current financial system development. We will discuss the main problems we faced when designing and implementing this model, and how

OOPSLA'05, October 16–20, 2005, San Diego, California, USA.
ACM 1-59593-193-7/05/0010.

we solved them. We will also mention the advantages of using an Object-Oriented dynamically typed language to implement it.

The remaining of this paper is organized as follows. Section 2 shows a motivating example of what can happen when performing operations with values of incorrect units. Section 3 talks about the design decisions that we had to make and presents our design. Section 4 comments on some implementation details. Section 5 highlights the concrete benefits, learned lessons and related work. And finally, Section 6 presents our conclusions and future work.

2. MOTIVATING EXAMPLE

Financial systems use financial formulas to get data that affects everything from strategic corporate investment decisions to client's interest and dividend payments. It is crucial to assure the data's correctness in any current or future evaluation context, in a natural and encapsulated way.

For example, Figure 1 shows the formula that should be applied to calculate the result of carrying out a simple investment with a fixed interest rate (all code examples are given in Smalltalk).

```
finalCapital := initialCapital * (1 +
                (interestRate * investmentTime))
```

Figure 1: Simple investment formula

This formula implicitly specifies each variable's unit that composes it. The variable "`initialCapital`" is expected to be a money quantity. The variable "`interestRate`", a percentage related to a unit of time and for "`investmentTime`", a value that expresses a quantity of time is also expected, but that value can not be defined with any unit, it has to be the same unit of time that is used for "`interestRate`". As we can see, this information is not explicitly given by the formula.

A common decision in software development is to use objects that simply represent "numbers" for these variables, being the programmer's responsibility to assure that the implicit information (i.e. units) of the numbers involved in arithmetic expressions is consistent with the expected result. For example, the programmer is the only one who knows that "`initialCapital`" represents a quantity of money. The system only requires of it to respond to the message "* anObject", no assertions are made by the system regarding the unit's correctness. This allows incorrect values to be used when evaluating this financial formula without obtaining any type of objection from the system, although most surely getting one from the user.

In this example of investment calculation, if the interest rate is expressed in annual terms and the variable "`investmentTime`" is expressed in terms of months, the expression will be arithmetically solved but the result obtained will be incorrect, since an amount expressed in terms of years has been mixed with one expressed in months without making any type of unit conversion.

Figure 2 shows a code example that could be used in these type of systems.

```
initialCapital := 100.
interestRate := 0.1.
investmentTime := 6.
finalCapital := initialCapital * (1 +
                (interestRate * investmentTime)).
finalCapital := 100 * (1 + (0.1 * 6)).
finalCapital := 160
```

Figure 2: Investment formula evaluation

The value of "`finalCapital`" will be 160 for this case instead of 105, since the formula was calculated for an investment of 6 years instead of 6 months (or 0.5 years).

One possible solution to this problem could be to ensure that all parameters of the formula are explicitly converted to the correct units (i.e. calling a function) before the formula is evaluated. The main problem with this approach is that the programmer has to make sure that all formulas in the system make use of this unit conversion code, which is both inefficient and error prone.

A solution following the good Object-Oriented design practices would be to encapsulate this responsibility in objects. For example, if the objects used to evaluate this formula would have been measurements (as defined in our proposed model), not only the correct arithmetic would have been done by the system, but also it could assure the result's validity regarding its unit. Evaluating the investment formula using an "`interestRate`" expressed in years with an "`investmentTime`" expressed in months would cause the system to convert these measurements (in this case from months to years) to obtain the right result.

Figure 3 shows how this snippet code would be written using our model. In this case, the objects used as variables in the formula are not solely numbers but a number with its unit. The variable "`initialCapital`" references to an object representing "100 dollars" (obtained for readability reasons by sending the message "dollars" to the object representing the number 100).

```
initialCapital := 100 dollars.
interestRate := 0.1 / 1 year.
investmentTime := 6 months.
finalCapital := initialCapital * (1 +
                (interestRate * investmentTime)).
finalCapital := 100 dollars * (1 + (0.1 / 1 year *
                6 months)).
finalCapital := 105 dollars.
```

Figure 3: Investment code using our measurement model

The system not only converts the measurements to the correct unit to get the right result, but it also takes care of the simplification between the "`investmentTime`" and "`interestRate`" units, in such a way that it returns a measurement expressed as a quantity of money as it was expected. We can see in Figure 4 the system's steps for the formula's evaluation.

```
fc := 100 dollars * (1 + (0.1/1 year * 6 month)).
fc := 100 dollars * (1 + (0.1/1 year * 0.5 year)).
fc := 100 dollars * (1 + (0.1/1 year * 0.5 year)).
fc := 100 dollars * (1 + (0.1/1 * 0.5)).
fc := 100 dollars * (1 + 0.05).
fc := 100 dollars * 1.05.
fc := 105 dollars.
```

Figure 4: How the formula is evaluated

As a result "finalCapital" is not just the number 105 (that is an instance of SmallInteger in the case of Smalltalk) but "105 dollars", an instance of **Measurement**, the class that represents a measurement in our model.

3. DESIGN

There were a number of requirements that this model had to fulfill that influenced its design. They are presented in the following sections.

3.1 Transparency and Uniformity

Transparency and uniformity for the programmer when operating with numbers and measurements was the main objective that we intended to obtain when designing this model. We realized that to fulfill that objective, measurements and numbers should be polymorphic, regarding arithmetic protocol such as +, -, / and * (among others). This decision allowed us to have generic arithmetic expressions without caring if they were evaluated with numbers or measurements, as in the capital investment formula used as example (Figure 1) that can be evaluated with numbers or measurements.

This design decision brings the advantage of not having to rewrite existing code nor to have to teach the programmers how to use the measurement model.

3.2 Measurement and Units Design

A simplified class hierarchy diagram for the measurement's model is shown in figure 5.

The abstract class ***MeasurementBehavior*** defines shared arithmetic protocol between measurements and any other arithmetic object (like SmallInteger, Float, etc.). ***ComparableMeasurement***, also an abstract class, provides shared protocol and implementation to its subclasses regarding comparison messages (i.e. <,>,<=,>=). The **Measurement** class is the one used to represent measurements throughout its instances, such as "1 meter" or "100 dollars". The **NullMeasurement** class is used to solve the "zero problem" (See section 3.7) and it implements the Singleton design patterns [6] and Null Object [15]. Finally the **MeasurementBag** class is used to represent

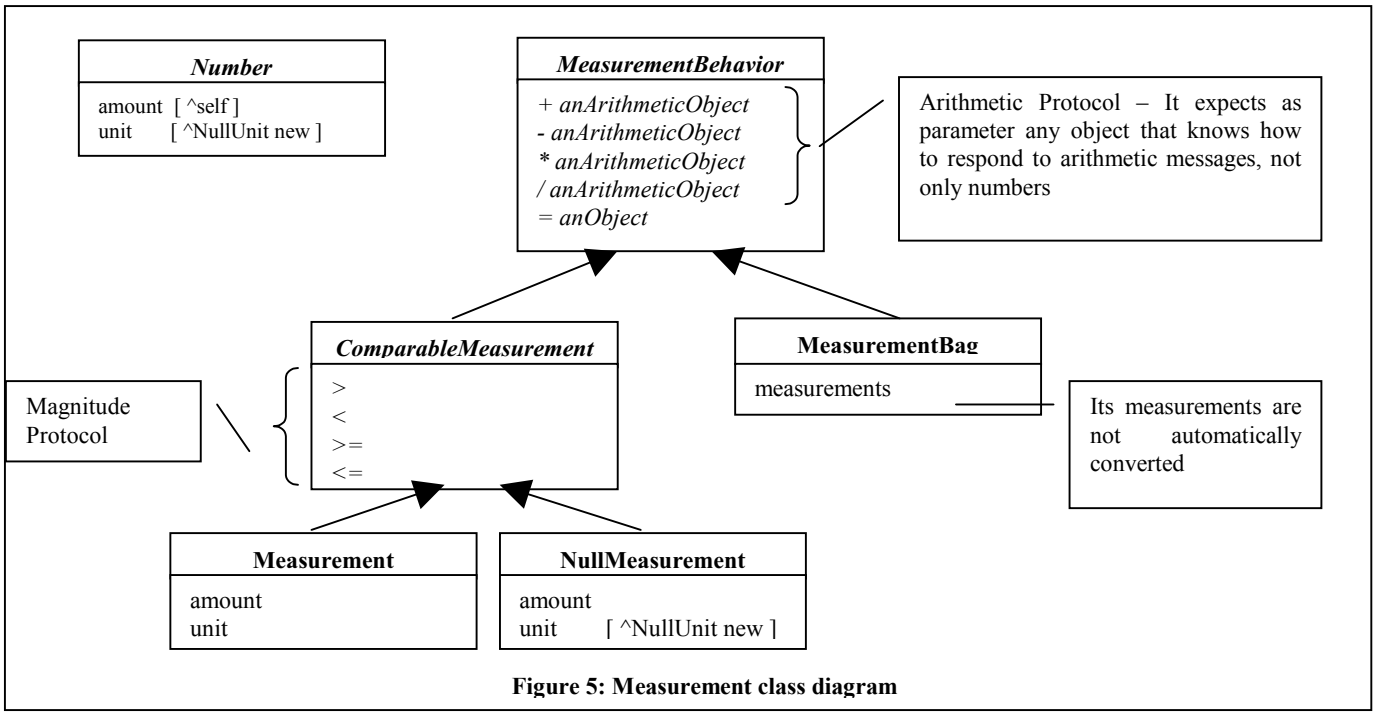

Figure 5: Measurement class diagram

measurement sets of different measurement dimensions. This class plays an important role in the financial domain to represent multi-financial instrument account balances, where a client can have investments of different currencies or securities like "100 dollars + 200 euros". The ***Number*** class was extended to provide the messages #amount and #unit to be polymorphic with **Measurement.**

Figure 6 shows a simplified class hierarchy diagram for units. Modeling the units was not as simple as it seemed at first. Initially, we had to solve the problem where each measurement dimension (e.g. "Length") had several units related to each other, creating a transitive relationship between them. To solve this problem, we decided to create a "base" or "canonical" unit to which all the other units could convert to (see [13]). Base units in our model are instances of the **BaseUnit** class while related (or

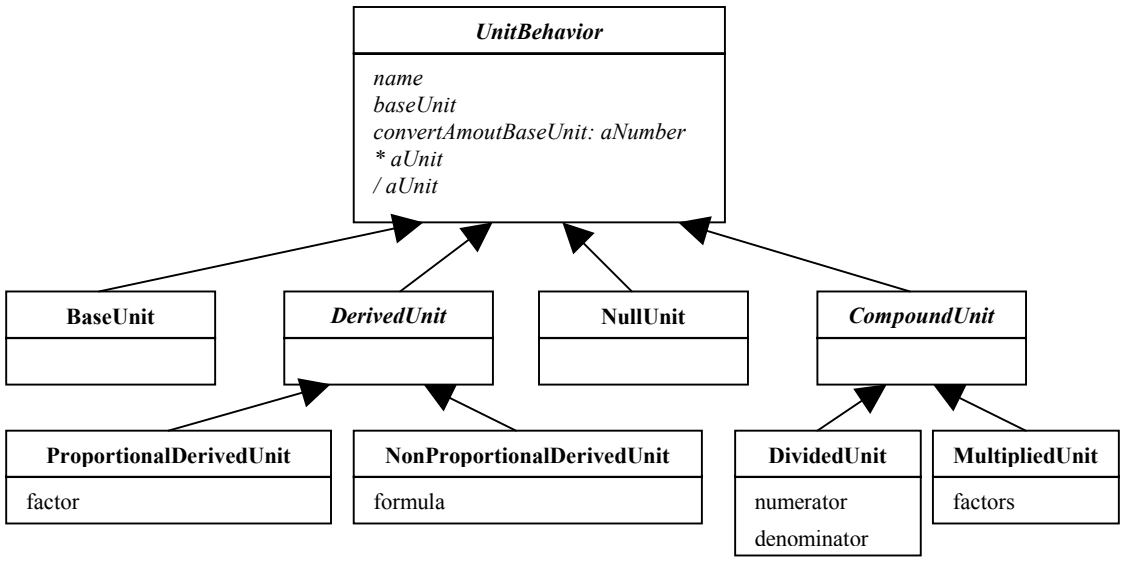

Figure 6: Unit class diagram

derived) units are instances of **ProportionalDerivedUnit** or **NonProportionalDerivedUnit**. Figure 7 shows a code example of how to create units for the "Length" dimension.

```
meter := BaseUnit named: 'meter'.
centimeter := ProportionalDerivedUnit
                baseUnit: meter
                proportionalFactor: 1/100
                named: 'centimeter'.
mile := ProportionalDerivedUnit
                baseUnit: meter
                proportionalFactor: 1609.344
                named: 'mile'.
```

Figure 7: How Units are created

It was also necessary to model composed units to represent the results of multiplication and division of measurements (**MultipliedUnit** and **DividedUnit** respectively).

3.3 Equality

When we started using the first implementations we realized that there are interchangeable measurements because they belong to the same measurement dimension. For example, "1 dollar" and "4 quarters" represent the same thing from the arithmetic point of view. We expect to get the same results when operating with them. However, they are different entities in reality and, therefore, different objects in the system.

It became essential to automatically convert related measurements of the same dimension without the programmer's explicit intervention. For example, it is expected for the system to return "true" when comparing "1 meter" with "100 centimeters" (they represent the same distance). To obtain this behavior the units are declared by means of an equivalence relationship among them, having a "base" unit to which all measurements of the same dimension can be converted to, making it possible to compare their quantities. The solution that we adopted models base units and derived units, being among them convertible according to a multiplication factor (for the case of proportional units like those of distance) or according to a formula (as is the case of units of temperature such as Kelvin, Celsius and Fahrenheit).

However, what was more important for us was the fact that we realized that some entities in reality are representations of others. For example, "100 centimeters" is a view, a representation of another entity like "1 meter" (and vice versa). The model must support this observation of reality, but not only that, the systems should assure that two non-identical objects (that is, they are objects at different memory locations) that represent the same entity in reality are treated as equal. For example, two objects representing the measurement "1 meter" represent the same unique real entity "1 meter", but as they are two objects that occupy different memory locations the system has to return "true" if they are compared via the equality protocol. It was therefore fundamental to make measurements behave as "values", name commonly used to denominate objects that are immutable and that can be the same beyond the area they occupy in memory (as for example String, LargeInteger, etc. See [2]).

We concluded that there were non-identical objects that could be equal, for example two objects "1 meter", and at the same time equal to other objects with no apparent relationship, except for the equivalence given by the units, as "100 centimeters" or "0.001 kilometers." Figure 8 shows a representation of this problem.

This problem also holds for comparison protocol such as <, >, <=, >=, where measurements have to be converted to a common unit before their amounts can be compared.

3.4 Numbers

Because we choose to treat numbers polymorphically with measurements, their protocol was extended to respond to the messages #amount and #unit. The message #amount was implemented to return "self", while #unit returns a null unit (instance of **NullUnit**). This decision allowed us to simplify the

implementation of certain methods when mixing numbers and measurements (as comparison, adding, etc.). Not making this decision would have forced the programmer to constantly verify if the object to collaborate with was a number or a measurement.

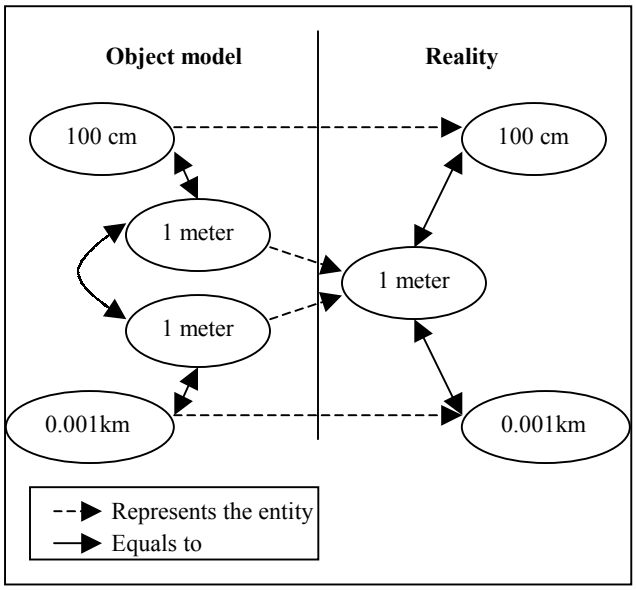

Figure 8: Reality and its object model

3.5 Adding and Subtracting Measurements

To add and subtract measurement of the same dimension does not imply any special challenge. The system must only check that it is operating with the same units. If not, it must convert the measurements to a common unit before adding or subtracting them. Figure 9 shows some examples.

```
distance := 1 meter + 50 centimeters.
distance := 3/2 meter.
distance := 100 centimeters + 50 centimeters.
distance := 150 centimeters
```

Figure 9: Adding measurement

Unfortunately, the same solution can not be adopted when adding or subtracting measurements of different dimensions (like meter and Kelvin) or not automatically convertible among them (like dollars and euros whose conversion depends on the quoting market, point in time, etc.). What should the system do under these circumstances?

One option would be to raise an exception indicating that an incorrect operation was performed. Some research work proposes this solution and others even propose these type of "violations" to be detected at compile time like [1] and [12]. However, we took a totally different approach creating for this case an object that represents a result for this operation.

These objects are instances of **MeasurementBag**, which are polymorphic with any measurement or number from the arithmetic point of view, but not from the comparison one.

This design decision allows the model to handle cases where some measurements "look" invalid at a certain point in time, when evaluating arithmetic expressions which become valid later on. But more important, it is the fact that there are some units of the same dimension that do not relate to each other directly, but only through a contextual or temporal relationship. Examples of these are the "currencies", which are used to measure "amount of wealth", where their relationship is not constant but it rather varies according to the market where they are quoted, a date, a financial operation, etc.

Thanks to this abstraction, a group of measurements of wealth can be valued in another wealth unit, using financial scenarios to convert them. Figure 10 shows how to value a financial balance that contains dollars and euros in another currency like the Argentinean peso, where "1 dollar" is equal to "3 pesos" and "1 dollar" is equal to "1.4 euros".

```
newYorkExchangeMarketScenario
   quoteFor: 1 dollar is: 3 pesos.
newYorkExchangeMarketScenario
   quoteFor: 1 dollar is: 1.4 euros.
accountBalance := 100 dollars + 200 euros.
accountBalanceInPesos :=
   newYorkExchangeMarketScenario
      convert: accountBalance to: peso.
accountBalanceInPesos := 1140 pesos.
```

Figure 10: Converting a MeasurementBag to a Measurement

3.6 Multiplication and Division of Measurements

To multiply or divide a measurement by a number implies to multiply or divide the amount of the measurement with the corresponding number.

To multiply measurements we decided to create measurements whose amount is the result of the multiplication of the amounts of the factors, and the unit is a compound unit instance of **MultipliedUnit,** whose factors are the units of the multiplied measurements.

When dividing measurements, the system creates a measurement whose amount is the result of dividing the amounts of each measurement and the unit is an instance of **DividedUnit** whose numerator is the unit of the measurement to divide and denominator is the unit of the dividend measurement.

When multiplying or dividing measurements, the model takes care of simplifying the units when necessary. In the same way, if a measurement bag is multiplied or divided by a measurement, the model applies the distributive property of the arithmetic operator directly.

For example, a velocity measurement of "45 miles per hour" is represented with an instance of **Measurement** whose amount is 45, and its unit is instance of **DividedUnit,** being its numerator the unit "mile" and its denominator the unit "hour". If we wanted to know the final velocity of applying an acceleration of 600 miles/ hour^2 (that is 600 miles/(1 hour*1 hour)) during 30 seconds, we should only evaluate the uniform rectilinear movement physical formula, and as one can see in Figure 11 the system would give us 50 miles per hour as result.

```
finalVelocity := (45 miles/1 hour) +
              ((600 miles/(1 hour * 1 hour)) *
               30 seconds).
finalVelocity:= (45 miles/hour) +
              ((600 miles/(1 hour * 1 hour)) *
               (1/120) hour).
finalVelocity:= (45 miles/hour) +
              (600 miles/1 hour * 1/120 hour).
finalVelocity:= (45 miles/hour) +
              (600/120 miles/hour).
finalVelocity:= 45 miles/hour + 5 miles/hour.
finalVelocity:= 50 miles/hour.
```

Figure 11: Getting the final velocity

3.7 The Zero Problem

One of the biggest problems we faced was how to treat measurements that represent the absence of what they are measuring, that is, "nothing".

It is natural to say that "0 meters" is equal to "0 centimeters", they both mean "nothing" in the distance dimension, but what should happen when comparing "0 meters" with "0 dollars", should the system respond that they are equal or different?, even more, what should happen if "0 meters" is compared with the number 0?

At first, our intuition leads us to believe that "0 meters" should not be equal to "0 dollars" (apparently they are measurements in different dimensions), and therefore, also not equal to the number 0. However, when we analyze the problem in detail, we reach to the conclusion that in all these cases we are dealing with different representations of "nothing", with the absence of what is being measured or counting, and one representation of "nothing" cannot be different from another one. For this reason, "0 meters" and "0 dollars", both representing "nothing" at the same time, must be equal, and they must be equal to 0 because it also represents "nothing".

This seems an anti-natural design decision, but we confirmed its correctness after studying, in greater detail, the meaning of measurement, that according to the NIST (National Institute of Standards and Technology) is defined as "The *value* of a quantity[1] is its magnitude expressed as the product of a number and a unit, and the number multiplying the unit is the *numerical value* of the quantity expressed in that unit." [13]. Therefore, if the value of the measurement's amount is 0, the result of multiplying it by its unit will always be 0.

We must also mention that 0 is not always the amount that represents the absence of what it is being measured. For example, in the case of temperature measurements, "0 Celsius degrees" is not equal to "0 Kelvin degrees", being "0 Kelvin degrees" the absence of temperature. This intriguing characteristic of 0 is also explained in [1] and [12].

3.8 Measurement Comparison

When measurements of the same dimension are compared, the system has to convert them to a common unit before comparing their amounts. On the other hand, when comparing measurements not automatically convertible between each other, like "100 dollars < 100 euros", the same solution cannot be adopted.

Contrary to the behavior chosen in the case of adding and subtracting non-convertible measurements, when comparing non automatically-convertible measurements we raise an exception indicating the impossibility of comparing them under these circumstances. The cause of the exception is that the measurement on the right side of the comparison cannot be converted to the unit of the measurement on the left side of it. However this exception has the characteristic of being "resumable", allowing the programmer to provide a suitable conversion for the measurements, letting the system continue with the normal flow of execution.

4. IMPLEMENTATION

To achieve the goals pursued, it was fundamental to extend the arithmetic operations offered by the programming language, making it possible to operate with numbers and measurements, that is, to treat them polymorphically regarding the arithmetic protocol.

Achieving these goals using statically typed languages implies that the measurements should have a common interface with the numbers (if the language offers interfaces) or to subclass a common class making it possible to define variables of that type to avoid type errors when compiling. Because of our experience with statically typed languages, we knew from the very beginning that this transparency objective would be impossible to achieve without modifying the number model offered by the programming language.

This can not be done with the current Java implementation because numbers are not objects, they are data types and their behavior is hardcoded in the virtual machine. On the other hand, it is not possible to define messages for arithmetic operators like +, *, etc., having to create a brand new arithmetic protocol loosing the desired transparency. The class wrappers, like Integer or Long, could be used to avoid the data type restriction, but none of them implement arithmetic messages, restricting us again to implement this model transparently.

Although .Net offers the possibility of overloading operators, it is not possible to extend the number's behavior (for example, to make them respond messages such as #amount and #unit), making it impossible to treat them polymorphically with measurements.

We should mention that we did not explore the possibility of implementing this model with parameterize classes. We did not spend time thinking about this solution because the implementation we achieved using Smalltalk was simple and straightforward.

Using a dynamically typed programming language like Smalltalk to implement this model allowed us to avoid the limitations found with statically typed languages. Not only there are no restriction on the arithmetic operation result's type, but Smalltalk is also prepared to collaborate with other type of objects when evaluating arithmetic expressions.

Due to the dynamic typing Smalltalk's characteristic, the only requirement for objects that collaborate among them is to answer to the messages they receive; that is why it is feasible to make a number collaborate with a measurement, as long as the measurements responds to messages like +, -, * and /.

[1] Quantity refers to what we call measurement

Figure 12 shows the message * (multiplication) implementation for the SmallInteger class in Smalltalk. It's worth noting that, if the primitive used to multiply the number fails, it delegates to the object received as parameter the responsibility to solve that message. In the case of multiplying a number by a measurement, the primitive will fail because the Smalltalk VM does not know how to multiply a number by a measurement and it will send the message * to the measurement with the corresponding number as parameter. It is now the **Measurement**'s method the one that knows how to carry out this operation, and using the double dispatch technique (see [8] and [11]) it finally returns the result of multiplying a measurement by a scalar.

```
* aNumber
  "Try to multiply using the VM"
  <primitive: VMprSmallIntegerMultiply>

  "If aNumber is an instance of an unexpected
class, delegate the responsibility of multiply
this number to that object"
  self primitiveErrorCode = PrimErrInvalidClass
    ifTrue: [^aNumber * self].

  ^self primitiveFailed
```

Figure 12: Implementation of * in SmallInteger

The implementation of this model showed us the importance of the programming language being used. If the programming language adheres to the stipulated principles in [10], extending its behavior will be simple and straightforward, without needing to modify base classes or lower lever components like the compiler or VM. The transparency and uniformity objectives were completely achieved with Smalltalk, without modifying a single line of code, adding a single keyword nor modifying its syntax. The behavior of basic objects like numbers was extended showing the beauty and simplicity of Smalltalk's design, reinforcing the need for open and extensible language implementations, that allow programmers to add any type of new behavior not considered during the language design phase.

5. RESULTS

5.1 Concrete Benefits

We have obtained many benefits from using this model to develop financial software. All of them come from the fact of treating uniformly "different ways of measuring or representing the same thing". In the next paragraph we highlight some concrete benefits we obtained while solving recurrent financial software problems.

5.1.1 Quotes

Financial instruments are quoted in different markets and in different currencies. It is normal to mathematically operate quotes for the same instrument expressed in different currencies, without noticing that they were incorrectly mixed.

Using measurements to model quotes allows the programmer to write code without validating that he or she is using the correct currencies, and without having to make explicit conversions when needed. The measurements model takes care of these validations and conversions automatically.

5.1.2 Nominal and Residuals quantities

When trading bonds, the trade's quantity can be expressed in nominals or residuals. A quantity expressed as nominal refers to the "concrete amount of bonds". A quantity expressed in residuals indicates the amount with respect to the amortization percentage that the bond issuer still has to pay.

Therefore, it is normal to talk about "1 NOMINAL BOND" or "0.5 RESIDUAL BOND", where both quantities represent the same concrete amount of BONDs, they are equivalent. For this example, the issuer still has to pay a 50% of the capital the buyer has lent him.

Modeling nominal quantities and residual quantities as related and automatically convertible units let the programmer avoid the verifications and conversions he or she should write when dealing with bond quantities, delegating that task to the measurements model.

5.1.3 Clean Price/Dirty Price

The same way a bond's quantity can be expressed as nominal or residual, a bond's price can be expressed as clean (that means, without including the accrued interests) or dirty (that is, including the accrued interests). The combination of these possibilities gives us four totally different ways to quote a bond's price.

Using measurements to represent bond prices where their units are related and automatically converted, relieves the programmer from this complex situation, being the objects the ones that take care of converting themselves.

5.1.4 Other Models based on Measurements

We have designed and implemented other models using measurements. One of them is a completely new set of classes that model the time domain (such as years, dates, months, etc.) in a simpler and more powerful way than the Smalltalk implementation.

For example, representing a date as a measure of time at year 0, for instance, simplified the comparison and arithmetic operations that could be performed on dates, like going forward "3 days" or going backwards "2 weeks".

We feel we achieved a simple and uniform design, due in great amount, to the use of measurements, to such an extent that we completely replaced the Smalltalk classes provided for this problem, like Date, Time, etc., with ours. Discussing the benefits of this model would imply another report. With this model it is easy to create financial magnitudes taking into account their dynamic dimension (that is, relative to time).

5.2 Lessons Learned

5.2.1 Immutable Objects

Having implemented the measurements model using immutable objects when appropriate, simplified its design and implementation. Not only they do provide the benefits mentioned in [2] but they also avoid non-contemplated consistency problems that could appear during an object's life cycle.

Immutable objects that are valid from the time they are created assure the programmer that she is not dealing with invalid ones, because an object is not instantiated if its preconditions do not hold.

5.2.2 Development Technique

We cannot end this paper without mentioning the advantages we got due to the use of the "Test Driven Development" technique. (See [3] and [4]).

Each observation we made on the problem domain about how to mathematically operate with measurements was programmed as tests that we took as the starting point to implement and create a better model.

It is necessary to highlight again the facilities that a dynamically typed and late binding programming language offers when using this technique. It is because of the dynamically typed characteristic of the language that we could make our model evolve smoothly due to not having to rename or refactor the variables type. The late binding characteristic allowed us to "program on the fly", that is, directly within the debugger, a characteristic still restricted in languages like Java or C#.

5.3 Related Work

Griggs has an implementation of measurements for VisualWorks [14], but the model needs special classes for temperature measurements, it does not allow measurement bags and it is mainly oriented to physical measurements.

Allen et.al. presented [1], where they proposed units to be part of the type system, allowing statically typed languages to check unit errors. Although they generate an Abelian group allowing algebraic operations on it, it does not support measurement bags and it uses an extension to Java called MixGen, where generic types are first-class.

Kennedy, [12] based his Ph.D. thesis on this issue, but he also proposes the units to be part of the type system in statically typed languages. There is no implementation of his work in object oriented languages.

In [5], Martin Fowler analyzed this problem but we needed and implemented a more general model.

There was a proposal to add units to the Java Programming Language (JSR 108, See [9]), but in contrast to our job, they just model units not measurements, loosing what we think is the most powerful feature of our model, to do arithmetic operations with measurements. Also, this work only deals with physical units.

In [4], Kent Beck proposed the use of Money and MoneyBag to solve the multi-currency problem. We go one step further generalizing all financial quantities to measurements, providing a uniform solution to financial systems.

6. CONCLUSION

To evaluate arithmetic operations not only with numbers but also with measurements allows the programmer to delegate to the system a cumbersome and error prone task, lowering the error rate produced by the execution of arithmetic operations with numbers that represent quantities of different units.

In financial systems, where one of the main objectives is to account (to measure) for how many of any possible financial instrument one owns, having a model with these characteristics is fundamental because treating measurements as first class objects simplifies the implementation of this kind of systems.

After completing this model, we realized that what we did was just follow a well know Object-Oriented design practice which is to always model Reality. That means that if a measurement exists in reality, the model must have an object to represent it, not just a number implicitly playing the role of a measurement. If Reality has a measure of distance, there must be an object to represent it, so if that measure is divided by a measurement of time, a new measurement is created to represent the real entity named "velocity", and not just a number. We need the system, we need its objects to know "the velocity" and not just a number implicitly playing the role of velocity, whose meaning can only be interpreted by a human.

We believe that a model such as this should be part of any development environment, because as we could experience, it solves a set of common and recurrent problems.

6.1 Future Work

There are some enhancements that can be made to this model, such as:

1) To reify the measurement dimensions. Currently, there are no classes to represent measurement dimensions such as "distance", "force", "capacity", "temperature", etc.

2) We have not implemented any kind of functionality to allow well know composed units to be named, such as Joule (equivalent to m^2*Kg*s^{-2}) or Pascal (equivalent to $m^{-1}*Kg*s^{-2}$).

3) To have units understand some messages to facilitate the creation of exponential units, such as "meter pow: 2" to represent m^2 (square meter).

4) To restrict the amount of some measurements to be a valid one, for example, measurements whose amounts must be a whole number.

7. ACKNOWLEDGMENTS

We would like to thank the Mercap's Software Development Team, for their comments and use of the measurement model. Thanks to the Object Group of the Exact Sciences School at the University of Buenos Aires. Also, we like to thank Michael Maximilien of the IBM Almaden Research Center for his review and friendship.

8. REFERENCES

[1] Allen, E., Chase, D., Luchangco, V., Maessen, J. and Steele, G. *Object-Oriented Units of Measurement.* Technical Report, OOPSLA 2004.

[2] Baümer, D., Riehle, D., Siberski, W., Lilienthal, C., Mergert, D.,Sylla, K. and Züllighoven, H. *Values In Objects Systems.* UBILAB Thecnical Report, 1998-10-10, Zurich, Switzerland

[3] Beck, K. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, MA, 1999

[4] Beck, K. *Test Driven Development: By Example*. Addison-Wesley, Reading, MA, 2002

[5] Fowler, M. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, Reading, MA, 1996

[6] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of ReusableObject-Oriented Software*. Addison-Wesley, 1995.

[7] Goldberg, A. and Robson, D. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, 1983.

[8] Hebel, K., Johnson, R.. *Arithmetic and Double Dispatching in Smalltalk-80.* University of Illinois at Urbana-Champaign.

[9] Java Extension proposal JSR 108, Source Code at http://jsr-108.sourceforge.net/javadoc/javax/units/Unit.html Extension proposal at http://www.jcp.org/en/jsr/detail?id=108

[10] Ingalls, D. *Design Principles Behind Smalltalk*. BYTE Magazine, August 1981. The McGraw-Hill Companies, Inc., New York, NY.

[11] Ingalls, D. *A simple technique for handling multiple polymorphism.* ACM SIGPLAN Notices, 21(11):347--349, Nov. 1986

[12] Kennedy, Andrew J. *Programming Languages and Dimensions.* PhD Thesis, University of Cambridge. Published as Technical Report No. 391, University of Cambridge Computer Laboratory, April 1996

[13] Taylor, Barry. Guide for the Use of the International System of Units. *National Institute of Standards and Technology,* Reading, Aplil 1995

[14] Travis Griggs, VisualWorks public repository, http://www.glorp.org/publicRepository/Measurements.html, 2004

[15] Woolf, B. *The Null Object Pattern.* in Pattern Languages of Program Design (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, Massachusetts, Addison-Wesley, 1997.