# Value with Quality

Bernhard Buergin
UBS, Bahnhofstrasse 45, 8021 Zurich, Switzerland
Bernhard.Buergin@ubs.ch, +41/236-7393, Fax +41/236-7040

## Abstract

In a context where software performs numeric calculations on the basis of incomplete or contradictory data, the end user might want to see not only the calculated results, but also information about the quality of the results. The Value with Quality pattern bundles a value together with its quality, and automates the propagation of quality to dependent results.

## Terminology

In this paper we use the term *quality indication* to denote an individual piece of information about the quality of a piece of data, such as "interpolated value" or "approximate price". A data value can have zero, one, or several quality indications associated with it. Data which has no quality indications at all is called *clean*; data which has at least one indication is called *tainted*. A value which is capable of having quality indications associated with it is called a *quality value*.

## Example

For certain types of bonds, the bond's yield can be calculated using the formula:

$$Yield = \left(\frac{R}{P}\right)^{1/L} - 1$$

Where $R$ is the redemption price, $P$ is the current price, and $L$ is the remaining lifetime of the bond. (Familiarity with these terms is not necessary to understand the example.)

Consider a program that calculates the yield of a bond based on this formula. Assume that the variables in the formula, such as $R$, are also calculated on the basis of more basic data. Let us also assume that during a particular calculation of a yield, the following situations were encountered:

- While calculating $R$, some input figures were not available in the desired currency. Another currency had to be used and converted to the desired one using the appropriate exchange rate.

- While calculating $P$, the latest price of the bond was not available so a price had to be estimated (extrapolated).

- While calculating $L$, the maturity date (end date) of the bond was calculated in two ways (based on data from two different databases). Normally the two dates should agree; in this case, there was a discrepancy and the date from the more reliable source was used.

Although there are problems here that compromise the quality of the calculated result, it may be a requirement that a value be calculated nevertheless on a "best efforts" basis. However, we want the final result (the calculated yield value) to "know" that these problems arose during the course of its calculation. This might mean, for example, that when the calculated yield is written to a report or

stored to a database, it be accompanied by a list of proper quality indications. In the current example, the output for the calculated yield might be as shown:

Yield = 4.25% (MissingRedemption, CurrencyConverted, ContradictoryMaturity)

In other words, the quality indications which affected the calculation somewhere along the way are reflected in the output of the final result.

# Context

A calculated value may be the result of a large number of atomic arithmetic operations (addition, exponentiation, etc.) that took place in many different methods. In the real world, the available input data for a calculation is often imperfect because it may be incomplete, contradictory, unreliable, or untrusted.

# Forces

**Imperfect data can occur anywhere**: Imperfect data could have been detected in any one of the calculating methods; in particular, there may be a large number of computational steps between the detection of imperfect data and the production of the final result which depends on that data.

**Abortion of calculation is not acceptable**: Simply throwing an exception after encountering imperfect data is often unacceptable, as the calculation should continue and a result be produced, albeit an imperfect one.

**Understandability of code should not suffer**: Normal calculation code should not be encumbered with checks about the data quality or alternate flows of control for abnormal situations. Ideally, if imperfect data is encountered, then any subsequent result that depends on this data should *automatically* be marked as being tainted, i.e., quality indications should propagate automatically through a calculation. In other words, if *any* of the raw data that flowed into the calculation had suboptimal quality, the results should reflect that, as well as an indication as to *what* in the input data was suboptimal.

# Applicability

This pattern can be used when calculations are performed on data of varying quality and the following conditions hold:

- The results of calculations should include information about the quality of the data that flowed into them; "the result is only as good as the worst of the inputs".

- If bad or questionable data is encountered, this should not abort the calculation in progress; the result should be calculated anyway but have quality indications associated with it.

- Quality Indications are cumulative. This pattern is most useful when a result can have several quality indications. In applications where encountering suboptimal data causes that calculation to abort (perhaps with a single quality indication), this pattern is overkill. See however "Exceptions and Quality Indications" in the "Consequences" section.

This pattern may not be applicable under certain space and time constraints. The C++ solution presented in the appendix adds a `long` to every number, doubling (or more) the space they occupy. Performance also takes a hit because every simple operation (such as addition) is now carried out by an overloaded method. In our implementation ("Key Figures"; see "Known Uses") the performance hit

was negligible: according to our profiler, the most often used operators (* and + for class QDouble) only accounted for 0.16% of total execution time in a typical run.

# Solution

For each data type that can have quality indications associated with it, define a class that acts like the built in type, but carries along a data member which holds the quality indications. Redefine the relevant mathematical and boolean operators such that when a result is produced by an atomic operation (+, -, etc.), the result acquires quality based on the quality indications of the input variables. In many cases this means the resultant quality is simply the union of the input qualities. However, other ways of combining qualities are possible. The Interval Arithmetic example under "Known Uses" is an example.

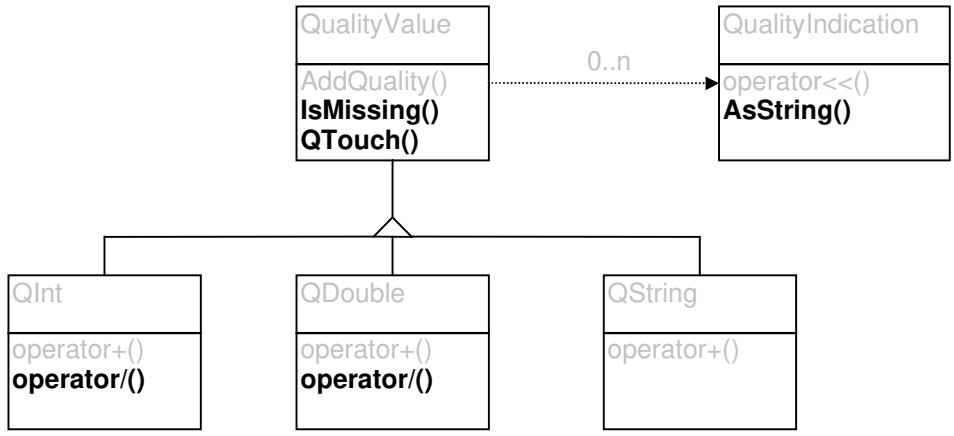
# Defining Criteria

A solution to the above forces is considered to be an instance of "Value with Quality" if it meets the following criteria:

- **Bundling:** a primitive value is encapsulated (bundled) together with its quality into a user defined type such that the value and its quality can be treated as a unit
- **Redefinition:** primitive operations (such as addition or concatenation) are redefined (e.g., overloaded in C++) to handle quality as well as value
- **Propagation**: A value's quality is automatically propagated to other values which were affected by it in primitive operations

These are the criteria which were used to identify "Known Uses".

# Structure




The QualityValue class defines the abstraction of a value that can have quality. The QualityValue class has a set of 0 to n QualityIndications. Whenever imperfect quality is encountered during a calculation, the corresponding quality indicator can be added to the value with the AddQuality

method. If any of the quality indications is the special QualityIndication "Missing", the boolean method IsMissing returns true. The method Qtouch is discussed under "Consequences".

In C++, the concrete types which can have quality can be implemented either with inheritance (shown in the structure above) or with templates. For the sake of variation, the sample code shows an implementation with templates. The "Consequences" section discusses the two options.

Note that a "string" value could also be a QualityValue (as in the diagram). This is because the result of a calculation could be a string, rather than a number. An example would be a calculation whose output is the name of the stock exchange index (e.g., Dow Jones) which had the highest standard deviation last year. A result of such a calculation might be, for example, "Dow Jones (InterpolatedValue)". The quality indication of "InterpolatedValue" does not mean that the string "Dow Jones" is an interpolated value (it makes no sense to interpolate strings); as always, the quality indication means that somewhere during the course of the calculation of the given result, a value had to be interpolated.

# Sample Code (C++)

The following code shows how a user of a QDouble quality type might use this type in C++. More details about how the QDouble class might actually be implemented are shown in the appendix.

This code would be used to calculate the gross price of a bond:

```
SomeClass::GrossPrice()
{
  QDouble Price = GetPriceFromDB(today);
  // DB-Layer sets NULL values to missing
  if (Price.IsMissing())
  {
                Price = GetPriceFromDB(yesterday);
                Price.AddQuality(Q_OLD_VALUE);
  }
  QDouble PremiumAdjust = GetPremiumAdjustFromDB();
  QDouble Premium = GetPremiumFromDB();
  // Divide -- no need to check for division by zero
  QDouble ActualPremium = Premium / PremiumAdjust;
  if (! ActualPremium.IsMissing())
  {
                Price = Price + ActualPremium;
  }
  return Price;
}
```

Later in the course of the calculation, a duration is calculated using the above price:

```
cout << Bond::Duration(GrossPrice(), CouponHeight, Fraction, NumCoupons);
```

If, during the calculation of GrossPrice, the Q_OLD_VALUE indication was set, this will appear in the output of the duration value. How it appears depends on the implementation of operator<<.

# Consequences

**Special value Q_MISSING**. It is useful to define one quality indicator which indicates missing (or unusable) data, and an associated boolean method IsMissing(). Situations such as division by zero render their results useless for further calculation; in these cases, in addition to setting the specific

quality indication (such as division by zero), the variable is also set to missing. If the application has an interface to a database system it can set values to missing if they are NULL on the database.

**Representation of the actual quality code**s. In some cases a numeric representation for each quality code is sufficient. When outputting quality values, the associated quality numbers are output as well. In other cases, the user might want to make a separate class for the actual quality codes, so they can hold additional information (e.g., a self-describing string).

**Automatic quality propagation and the need for QTouch()**. The philosophy behind this pattern is that if any data of dubious quality affect a calculated result, that result should reflect that quality. In most cases this is automatically achieved through the overloaded operators. In other words, if C = A + B, and A has a quality indicator, then C will obtain it too (as will any subsequent number that "comes in contact with" C). Consider, however, the following case:

```
if (A < B)
{
        x = C + D;
}
```

Assume that A is tainted but that C and D are clean. Then x will also end up clean, even though the conditions under which x was calculated were of dubious quality[1]! This is wrong, as x should clearly inherit any quality indications A or B might have. To deal with this, we define a method QTouch() by which one variable can "touch" another and thus confer its quality indications to the other variable. The above code would be changed to:

```
if (A < B)
{
        x = C + D;
        x.QTouch(A);
        x.QTouch(B);
}
```

Unfortunately this solution brings back some of the complexity we were trying to hide in the quality values class, but our experience was that it is manageable. There is one additional point to make in this connection, namely that there are cases where a tainted variable influences a value but should not be reflected in that value:

```
if (A.IsMissing())
{
        x = y;
        // don't do x.QTouch(A) here!
}
```

In the above example, the variable A affects the value of x, but we don't want A's quality indications to be propagated to x.

**Dealing with math errors**: in addition to catching division by zero, the overloaded operators can deal with other abnormal math situations such as overflow or a domain error (such as taking the square root of a negative number in real-valued math). In C++, for example, this can be done by checking the global variable errno (from <errno.h>) and checking for violations such as EDOM or EOVERFLOW, and setting the appropriate quality indicators.

**Exceptions and Quality Indications**: when data is encountered during a calculation whose quality is so bad that it makes no sense to continue the calculation, the logical action is to terminate that calculation. Using the present pattern it is not absolutely necessary to do this; it is possible to let the

[1]. The alternative choice of making all booleans evaluate to false if any participant in the expression was tainted was found to be too restrictive to be practical. For example, if A were an estimated value of 4, above, and B was a clean value of 5, we would still want (A < B) to evaluate to true.

calculation take its course and let the quality indications take care of the result. However, this wastes computational effort. A reasonable approach is to throw an exception of class ExceptionWithQualityIndication which is then caught by the top-level loop of the calculation engine. This top level module then creates an empty result containing as its sole quality indication the indication which was in the thrown exception.

# Consequences (C++ Related)

**Separation of the quality classes from the actual quality codes**. Each project will have its own quality indications (although some, such as Q_MISSING and Q_MATH_ERROR might always be present). The concrete quality indications used in a project can be defined in a separate file which is then #included by the quality class code. (For simplicity of exposition, we used simple symbolic constants in this paper; in reality, the codes' names could be protected using a namespace or an enum within a class scope.)

**Concrete types implemented with templates or inheritance.** As mentioned in the "Structure" section, in C++ the concrete types which can have quality can be defined from the QualityValue abstraction either by using templates or inheritance. With inheritance, the overloaded operators need to be defined for each concrete subclass of QualityValue (they cannot be defined in QualityValue because the underlying types are different for each subclass). Therefore, using templates can reduce source code. However, if the subtypes are not all identical (as in the diagram in the "Structure" section, where the QString class has no division operator), or the implementations differ between subclasses, then inheritance is probably a simpler option.

# Known Uses

**Key Figures**: the example illustrated in "Sample Code" is from a medium sized (100k LOC) C++ application at UBS called "Key Figures" whose purpose is to calculate risk and return characteristics of financial instruments. The concrete QualityValue types were defined by inheriting from QualityValue (rather than by using templates) because fairly diverse concrete types were defined (including a Qstring and a QDate).

**Satellite Control System**: A use of Value with Quality in a satellite control system was communicated to the author. In this use, telemetered values have, at any point in time, a value and a set of quality indications (boolean statuses) such as NO_TELEMETRY, KNOWN_INVALID, UNRELIABLE, VOLATILE, etc. Values are propagated during arithmetic operations, in the manner described in this paper. Values that are presented to the end user have their quality marked by color coding in the user interface.

**Interval Arithmetic**: Abelson and Sussman give an example of the pattern's use in Lisp in [SICP], pp 93 ff. Their "Interval Arithmetic Package" allows calculations to be performed on inexact quantities. The tolerances of the input numbers propagate up to the final result in a worst-case fashion. For example, if a number between 5 and 6 is added to a number between 7 and 8, the result is a number between 12 and 14. Abelson and Sussman implement the package by defining an abstract "Interval" object (representing a number with lower and upper bounds), analogous to our QualityValue class. They then define arithmetic operators (addition, multiplication, etc.) which incorporate the worst-case rules. Abelson and Sussman mention the interesting point that interval arithmetic (or QualityValue arithmetic in general) has the inherent shortcoming that equivalent algebraic expressions may lead to different results[2], and that resolving this issue is very difficult.

---

[2]. The example given in [SICP] is the formula for parallel resistance; this can be written either as $(R_1R_2/(R_1+R_2))$ or $1/(1/R_1 + 1/R_2)$. The tolerances resulting from these calculations are not equivalent. The code from the Key Figures project shown in the "Sample Implementation" has similar problems. As a simple example, if a = 2*b then the statement y = x + a/b will confer a's and b's quality to y, whereas the equivalent statement y = x + 2 will not.

**Insecure Data in Perl**: The Perl programming language [LW] incorporates "Value with Quality" as part of the language in its "taint mode", designed to help deal with security issues. Data whose origin lies outside the Perl program (command line arguments, environment variables, etc.) are considered *tainted*. Concatenating a tainted string to a clean string results in a tainted one. Propagation of quality is conservative (worst-case): "Any variable set within an expression that has previously referenced a tainted value becomes tainted itself, even if it is logically impossible for the tainted value to influence the variable" ([LW], p. 357).

# Related Patterns

Patterns have been documented which satisfy one or two of the three defining criteria of Value with Quality (bundling, redefinition, and propagation). The CHECKS pattern language [WC] bundles a basic value with supplementary information, while the Observations and Measurements patterns [MF] bundle values with their units and redefine operators to handle the units.

The CHECKS pattern language is a set of patterns for dealing with user input and the fact that programs must be able to handle invalid input. Three of the patterns are particularly relevant here: Whole Value, Exceptional Value, and Diagnostic Query. Whole Value captures values that occur in the domain, bundling the values with their units. Missing or undefined values are represented by Exceptional Value. When presenting values to the user, the supplementary information (e.g., units) must be presented to the user as well; this is handled by Diagnostic Query.

The Observations and Measurements analysis patterns ([MF], Chapter 3) also bundle values with their units in a class called Quantity. A value representing a price, for example, will consist of a number plus a currency. Operators are redefined so that they take units into account. For example, "an addition operation allows quanitites to be added together as easily as numbers but checks the units so that 34 inches are not added to 68 kilograms" ([MF], p. 37).

# References


[LW] Larry Wall, Tom Christiansen, and Randal L. Schwartz, with Stephen Potter. *Programming Perl*, Second Edition. 1996, O'Reilly & Associates, Inc.

[MF] Martin Fowler, *Analysis Patterns: Reusable Object Models*, 1997, Addison Wesley.

[WC] Ward Cunningham, "The CHECKS Pattern Language of Information Integrity", in *Pattern Languages of Program Design*, Coplien and Schmidt, eds.1995, Addison Wesley, pp. 145 -155.

[SICP] Abelson, H. and Sussman, G., with Sussman, J. *Structure and Interpretation of Computer Programs*, Second Edition. 1996, MIT Press.


# Acknowledgements


Dr. Peter Meinen and Nevo Hed helped apply this pattern to the Key Figures project. Thanks to Dirk Riehle for encouragement and valuable feedback.

# Appendix: Sample Implementation in C++

The following code illustrates a possible implementation of this pattern. For simplicity, the QualityIndications are represented as integers instead of full blown classes (i.e., they carry no self-descriptive information). The sample code uses templates rather than inheritance for the different QualityValue types.

**Class Definition**

A declaration for a simple quality value class is given below. The key member variable is _quality, which holds a bitmap of up to 32 quality indications. When an atomic operation takes place, the result acquires the bitwise OR of the quality values of the input variables. Using this template class, the necessary classes can be instantiated, e.g., for ints and doubles.

```
template<class T>
class QualityValue
{
public:
  QualityValue(T value) : _quality(0), _value(value) {}
  QualityValue() : _quality(0), _value(0) {}

  void QTouch(const QualityValue& qv) { _quality  |= qv._quality; }
  void AddQuality(long quality) { _quality |= quality; }
  bool IsMissing() const  { return _quality & Q_MISSING; }

  friend QualityValue<T> operator+(
                const QualityValue<T>& qv1,
                const QualityValue<T>& qv2);
          // ... other operators

  friend ostream& operator<<(
                ostream& os,
                const QualityValue<T>& qv);
protected:
  long _quality;
  T _value;
};
typedef QualityValue<int> QInt;
typedef QualityValue<double> QDouble;
```

**Implementation**

The implementation of operator/ (division) could be as shown below. Other operators are similar.

```
template <class T>
QualityValue<T> operator/(
                const QualityValue<T>& qv1,
                const QualityValue<T>& qv2)
{
  QualityValue<T> result;
  result._quality = qv1._quality | qv2._quality;
  if (qv2._value == 0)
  {
    // Q_xxx indications are in an #include file.
    result.AddQuality(Q_MATH_ERROR);
    result.AddQuality(Q_MISSING);
    result._value = 0;
  }
```

```
  else
  {
    result._value = qv1._value / qv2._value;
  }

  return result;
}
```

The actual quality indications used in a particular project would be defined in a separate file:

```
const long Q_MISSING = 1L;
const long Q_MATH_ERROR = 2L;

// project specific codes
const long Q_OLD_VALUE = 4L;
const long Q_INTERPOLATED_VALUE = 8L;;
const long Q_MANUAL_CORRECTION = 16L;
// etc.
```