On the Impact of the Types Conversion in Java onto the Coupling Measurement

Miguel Lopez[1], Alain Abran[3] , Grégory Seront[1] ,Naji Habra[2]

[1] *CETIC asbl*
Rue Clément Ader, 8
B-6041 Gosselies, Belgium
malm@cetic.be, gs@cetic.be

[2] Faculty of Computer Science
Namur University – FUNDP
Namur, Belgium
nha@info.fundp.ac.be

[3] Ecole de Technologie Supérieure
Montréal,Québec;
Canada
aabran@ele.etsmtl.ca

Formatted: English (U.S.)

Abstract

Software measurement represents an important topic heavily discussed within the software engineering community. Since thirty years, software measurement has become an important domain where interesting debates have occurred.

Internal measurements of software do not necessitate any execution. Since these measurements are automated, it is commonly accepted that during such measurements errors cannot occur. Indeed, such measurements have no random or probabilistic aspect.

The current paper aims at showing that other sources of error or uncertainty exist in the software measurement. Sources of uncertainty can appear before the measurement itself, that is, at the measurement design level. Indeed, mistakes related to the design of measurement can occur, and therefore affect the measurement results when executing the measures.

The current paper extends the notion of uncertainty to the measurement design level, and highlights the impact of the design uncertainty onto the measurement result.

Formatted: English (U.S.)

Keywords: uncertainty, measurement design, typecasting

1. Introduction

Software measurement represents an important topic heavily discussed within the software engineering community. Since thirty years, software measurement has become an important domain where interesting debates have occurred.

During last decades, a type of measurement played an important role in the development and the progress within the software measurement field, that is, internal software measurement.

Measurement of internal characteristics of software is defined as measurement of software without executing it

According to (ISO 9126), *the internal metrics may be applied to a non-executable software product during its development stages (such as request for proposal, requirements definition, design specification or source code). Internal metrics provide the users with the ability to measure the*

quality of the intermediate deliverables and thereby predict the quality of the final product.
Such measurements have met a great success, since they can be automated, and therefore are considered as low-cost means to assess the software quality.

Examples of internal of measurement are McCabe cyclomatic number or fan-out coupling measurement.

Internal measurements of software are often source code-based methods. Indeed, the measuring instrument analyses the source code in order to extract the countable elements related to a given internal measurement. For instance, the McCabe cyclomatic complexity number counts the number of decision nodes, that is, the tokens *if, for, while, switch* for Java.

Source code analysis is a deterministic phenomenon. Deterministic refers to events that have no random or probabilistic aspects but proceed in a fixed predictable fashion. In other words, a given element (such as an *if* token) within the measurement of the McCabe number of a given source code will always be detected by the measuring instrument. Repeating (without changing the measurement conditions) or reproducing this measurement will not affect the measurement result in the sense that the *if* token will always be detected.

Since the source code analysis is a deterministic operation, the internal measurement is never related to any notion of errors or uncertainty. Indeed, no doubts can be associated with the results of internal measurement. Errors due to incorrect source code analysis are considered as bugs in the source code analyzer, but not as an error of the measurement.
Nevertheless, other sources of uncertainty exist in the metrology literature. These sources of uncertainty are not only found in the measuring tool itself, that is, the source code analyzer.

The object of the current paper is to show that even thought source code analysis is a deterministic process; internal measurements can be affected by uncertainty through other sources of errors. We take here as an example the measurement of coupling. Coupling is a measurement which is often used by the software measurement community. The current paper treats the impact of the type conversion in Java for the coupling measurement.

2. Coupling Measurement

Coupling measurement has become an increasingly popular research area (Briand 1999) . A number of proposals about coupling attribute definitions and coupling measures can be found in the literature. However, as there is no consensus on coupling related terminology and formalisms, many measures are expressed in an ambiguous manner (Briand 1999): for instance, the counting rules are embedded within the definitions of the coupling measures, without explicit design rationale for the selected numerical assignment rules selected. .
Moreover, some coupling mechanisms (type conversion) are not taken into account, and of course impact the measurement result.

An important step in the design of a measurement method is the definition of the attribute to be measured that is here, the coupling concept.
As previously said, many definitions are available in the state of the art. Three of them are given in the following paragraphs.
In (Pressman 1999), coupling is defined as a measure of interconnection among modules in a software structure. In (Yourdon 1978), coupling is the degree of interdependence between modules. In (Chidamber 1994), a definition of coupling applied to the OO paradigm is given: two objects are coupled if and only if at least one of them acts upon the other, X is said to act upon Y if the history of Y is affected by X, where history is defined as the chronologically ordered states that a thing traverses in a time.
An important concept is shared by these three definitions (and also by most of the definitions), that is, connection.
Briand *et al*. highlight the different types of connections involved in coupling. Based on three frameworks of coupling measurement (Eder 1994), (Hitz 1995), (Briand 1999) , nine connection mechanisms between classes have been described.

1. Methods share data (public attributes etc.)
2. Method references attribute
3. Method of a class A invokes method of another class B
4. Method receives pointer to method
5. Class is type of a class' attribute (aggregation)
6. Class is type of a method's parameter or return type

7. Class is type of a method's local variable
8. Class is type of a parameter of a method invoked from within another method
9. Class is ancestor of another class

Many connection mechanisms deal with the type of the class (i.e. mechanisms 5 to 9). It could be of interest to probe further these type-based mechanisms. In other words, can the type of a given class be modified? Are there means in Java that allow types modification?

Unfortunately, such type modifications can occur, and are called types conversion. Indeed, it is possible to convert the type of a given class into another type within a specific context. Fortunately, there are strong limitations on type conversion. However, the conversion of types can affect the above connections mechanisms, and therefore can affect coupling.

The following sections develop the notion of type as well as the type conversions that exist in the Java programming language.

3. Brief History of Typing

Types represent an important issue in programming languages. Indeed, types limit the values that a variable can hold or that an expression can produce, limit the operations supported on those values, and determine the meaning of the operations.
Languages whose types are checked at compile-time are called *strongly typed*. Examples of such languages are Java, Ada, C++.
By contrast, languages where all checking is deferred to run time are called *weakly typed*. Examples of weakly typed languages are PHP and Python.
During the short history of Computer Science, the types within the programming languages have evolved and the following paragraphs provide an overview of this evolution.

According to (Cardelli 1985), *in early programming languages, computation was identified with numerical computation and values could be viewed as having a single arithmetic type. However, as early as 1954, Fortran found it convenient to distinguish between integers and floating-point numbers, in part because differences in hardware representation made integer computation more economical and in part because the use of integers for iteration and array computation was logically different from the use of floating point numbers for numerical computation.*

Thereafter, Algol 60 (Cardelli 1985) included an explicit concept of type and associated requirements for compile time type checking. Algol 60 block-structure requirements allowed not only the type but also the scope (visibility) of variables to be checked at compile time.

Since Algol 60, programming languages evolve in order to implement better type checking and inference mechanisms. For instance, Simula is the first object-oriented language to include a concept of type which includes classes. Indeed, Simula types, as in most object-oriented languages, are actually all classes, and vice-versa.

In the early 1970's (Cardelli 1985) , Ken Thompson and Dennis Ritchie developed the C programming language for use on the UNIX operating system. C has a type system similar to that of other ALGOL descendants such as Pascal. There are types for integers of various sizes, both signed and unsigned, floating-point numbers, characters, enumerated types (enums), and records (structs).

In the early 1980's, ADA became an ANSI standard (ANSI/MIL-STD 1815). Ada has only a very small set of predefined types, while new types are defined according to domain specific needs. Thus Ada is more like PL/I where data types are described in terms of what is needed, and not chosen from a predefined set. Type checking mechanisms implemented in Ada are known as very efficient mechanisms in the sense that *objects* of different types in Ada are incompatible, i.e. may not be assigned or mixed, even if they have matching value ranges.

In mid-1990's, another well-known programming language was developed by Sun, that is, Java. Java holds a types checking mechanism similar to that of Ada types checking, but nevertheless the Java type checking prevent dangerous types conversion. For instance, let us look at the two following class definitions:

```
class PayCheck {
int Salary;
int MonthsOfWork;
}
```

```
class EngineControl {
int Gear;
int Speed;
}
```

At a representation level, both classes are compatible in the sense that an instance object will hold the same memory space and that each attribute will be decoded the same way *at the machine level*. Nevertheless, it is obvious that if we try to assign for some obscure reason an instance of PayCheck to an instance of EngineControl, we are heading toward some serious problems. Fortunately, this type of assignation is forbidden in Java.

The brief history of the evolution of types in programming languages previously explained shows an important trend in order to better check and ensure types. Indeed, it is often accepted that languages with strong types checking mechanisms like Ada or Java allow making safer and more understandable software.

However, using so strongly typed languages decreases the programmer productivity due to a lack of flexibility within the types. For instance, in Java we have the method “clone” which is a member of the class Object from which all classes descend. This method as we can guess from its name clones an object to create a new one. Since this method belongs to the class “Object”, its return type is of type “Object”. Let us look at the following code sample:

```
…
Rectangle r = new Rectangle;
Rectangle c;

Object o = r.clone(); // Clone rectangle
C = (Rectangle) o; // Cast it back to Rectangle type
```

Without type casting, we could not cast back the object created by “clone” to its original type. This will prevent us to create methods as generic as “clone”, and thus lead to a decrease in productivity.

In this context, some mechanisms are therefore available within the language in order to overcome some of the loss of flexibility.

4. Types Conversion in Java

In Java, as in many other languages, the type conversion and promotion are mechanisms that allow certain flexibility with types.

According to (Sun Java, 2004), the types of the Java programming language are divided into two categories: primitive types and reference types. The primitive types are the boolean type and the numeric types.
The numeric types are the integral types byte, short, int, long, and char, and the floating-point types float and double.

The reference types are class types, interface types, and array types. There is also a special null type. An object is a dynamically created instance of a class type or a dynamically created array.

Primitive type conversions are also possible in Java. However, connections with primitive types are not considered in (Briand 1999). So, within the scope of this paper, it is assumed that primitive type conversions do not affect coupling. It is important to notice that this hypothesis must be tested in order to clarify the influence of primitive type conversions onto coupling.

According to (Sun Java, 2004); a lot of of references types conversions exist in the Java programming language. To gain some insights into this issue, only casting conversion is investigated here; in other words, it is assumed that casting conversion represents a mechanism that can strongly affect coupling.

Casting conversion is applied to the operand of a cast operator: the type of the operand expression must be converted to the type explicitly named by the cast operator.

The following highlights the syntax of the casting conversion. Let class *X* be a subclass of class *Y*.

```
//instanciation of a new class X
X x = new X();

//instanciation of a new class Y
Y y = new Y();

//assignement of Ywith a casting
//conversion of object x to type Y.
Y y = (Y) x;
```

The casting conversion is legal, that is, compiling does not return any error, and no error will be produced at run-time. This type conversion is legal because X is a subclass of Y.

The detailed rules for compile-time correctness checking of a casting conversion of a value of compile-time reference type S (source) to a compile-time reference type T (target) are as follows:

- If S is a class type:
    - If T is a class, then S and T must be related classes – that is, S and T must be the same class, or S a subclass of T, or T a subclass of S; otherwise a compile-time error occurs.
    - If T is an interface type:
        - If S is not a final class, then the cast is always correct at compile time (because even if S does not implement T, a subclass of S might).
        - If S is a final class, then S must implement T, or a compile-time error occurs.

In (Sun Java, 2004), other casting conversion cases are described. However, only casting conversions related to class types are relevant for the coupling, since current frameworks only consider connections with classes but not interfaces nor primitive types (Eder 1994), (Hitz 1995), (Briand 1999).

Moreover, current coupling frameworks do not take into account the impact of the casting conversion onto coupling, hence, current measuring instruments do not implement counting rules that consider the casting conversion.

For instance, the example below shows a type conversion which is not specified in the previous framework.

```
class A
{
…
        Void method() {
                ((B).a).methodB();
        }
}
```

Code Sample 1

It is assumed that object a can be converted into type of class B.
Briand *et al.* do not explicitly identify this case, and therefore the coupling relationship between classes A and B will not be counted. Hence, a certain level of uncertainty can be introduced within the measurement result.

In such a context, neglecting the type conversion of Code Sample 1 can lead to measurement errors. And, in that sense, the metrology can help clarifying the kind of errors due to this negligence and handling them.

5. Useful Metrology Concepts

According to (NIST 1994), *In general, the result of a measurement is only an approximation or estimate of the value of the specific quantity subject to measurement, that is, the measurand, and thus the result is complete only when accompanied by a quantitative statement of its uncertainty.*
However, such errors or uncertainty are considered as inexistent in the internal measurement, since internal measurement is considered as a deterministic.

Previous section shows that a specific kind of uncertainty is introduced in the measurement results due to a misconception in the measurement design, that is, the lack of a case of connexions between classes.

So, what is the type of such uncertainty? Can the metrology field highlight this uncertainty?

Some answers can be found in the approach given in the Guide to the Expression of Uncertainty in Measurement (GUM 1995). This guide represents the current international view of how to express uncertainty in measurement.

Obviously, neglecting the type conversion illustrated by Code Sample 1 represents a source of uncertainty. But, what does uncertainty mean? Based on the (GUM 1995), uncertainty (of measurement) is defined as a *parameter associated with the result of a measurement, that characterizes the dispersion of the values that could reasonably be attributed to the measurand*.

A note completes the above definition by arguing that *Uncertainty of measurement comprises, in general, many components. Some of these components may be evaluated from the statistical distribution of the results of a series of measurements and can be characterised by experimental standard deviations. The other components, which can also be characterised by standard deviations, are evaluated from assumed probability distributions based on experience or other information.*

Within this note, an important notion is mentioned, that is, components.

In practice the uncertainty on the result may arise from many possible sources, including examples such as incomplete definition, sampling.

To estimate the overall uncertainty, it may be necessary to take each source of uncertainty and treat it separately to obtain the contribution from that source. Each of the separate contributions to uncertainty is referred to as an uncertainty component.

According to (NIST 1994, GUM 1995), the uncertainty of the result of a measurement generally consists of several components which may be grouped into two categories according to the method used to estimate their numerical values:

A. those which are evaluated by statistical methods,
B. those which are evaluated by other means.

It is important to note that there is not always a simple correspondence between the classification of uncertainty components into categories A and B and the commonly used classification of uncertainty components as "random" and "systematic."

A note within the GUM (GUM 1995) highlights the distinction between error and uncertainty. Indeed, the result of a measurement after correction can unknowably be very close to the exact value of the measurand, and thus have negligible error, even though it may have a large uncertainty.

To remind, an error is defined as the difference between a true value, x, and a measure value, xi. And, the uncertainty is an estimate of the true value as a possible range of errors. So, even if a measurement value is corrected, and therefore with a negligible error, the uncertainty can remain important.

Random error typically arises from unpredictable variations of influence quantities. These random effects give rise to variations in repeated observations of the measurand. The random error of an analytical result cannot be compensated for, but it can usually be reduced by increasing the number of observations.

Systematic error is defined as a component of error which, in the course of a number of analyses of the same measurand, remains constant or varies in a predictable way. It is independent of the number of measurements made and cannot therefore be reduced by increasing the number of analyses under constant measurement conditions.

In regard with these definitions, the error due to the missed identification of the type conversion illustrated by Code Sample 1 cannot be considered as a systematic error nor a random error.

On the one hand, the error due to missed type conversion within the coupling measurement does not vary in a predictable way.

On the other hand, this error is not compensated by increasing the number of observations, and therefore it cannot be considered as a random error.

So, how can the uncertainty due to missed type conversion be qualified?

According to (VIM 1993) , for a material measure, the variation due to an influence quantity is the difference between the values of the supplied quantity when the influence quantity assumes two different values.

This variation due to an influence quantity, would then have an influence on the repeatability and replicability of the measurement results, since various measurers could miss the identification of the variation (that is the special condition previously mentioned when measuring coupling).

So, a variation within the measurand can be missed by the measuring instrument, and therefore the instrument can show an erroneous measurement result. However, the classical typology of error (systematic and random) is not complete enough to qualify the error due to a missed type conversion within a coupling measurement.

Therefore, the metrology provides other means to qualify and estimate the uncertainty related to a measurement result, that is, the types A and B evaluation of uncertainty

This approach suggested in (GUM 1995) helps evaluating the uncertainty that cannot be classified within one of the classical error types (systematic or

random).
Actually, types A and B of uncertainty evaluation represent a different means to handle uncertainty of measurement.

A Type A evaluation of standard uncertainty may be based on any valid statistical method for treating data. Examples are calculating the standard deviation of the mean of a series of independent observations. When expressed as a standard deviation, an uncertainty component is known as a standard uncertainty.
So, the evaluation of uncertainty by the statistical analysis of series of observations is termed a Type A evaluation of uncertainty.

A Type B evaluation of standard uncertainty is usually based on scientific judgment using all the relevant information available, which may include:

- previous measurement data
- experience with, or general knowledge of, the behavior and property of relevant materials and instruments
- manufacturer's specifications
- data provided in calibration and other reports
- uncertainties assigned to reference data taken from handbooks

So, the metrology provides some means to estimate the uncertainty related to a given measurement method, which exhibits non classical errors (like the missed type conversion).An important open issue is now to evaluate the uncertainty due to the type conversion in the coupling measurement. The evaluation can be done by a statistical method (type A) or other means (type B). However, this work will be done in a future work, since this is out of the scope of the current paper.

6. Conclusion

Internal measurement can be characterized as a deterministic phenomenon in the sense that the operation itself of measurement is deterministic.

Thus, there is no source of uncertainty that affect the application of an internal measurement, that is, the parsing of the source code. And, if uncertainty exists, it is often considered that its source is a bug in the measuring instrument, which is, actually, a software tool.

However, a measurement method is not limited to the application phase. Indeed, the measurement is a broader process, which includes the definition of the measurement, the application to the measurement, and the exploitation of the results (Abran 1999). So, from this higher point of view, sources of uncertainty can be present at different phases of the whole measurement process.

Discussions about errors in internal measurements are often centered on the application of the measurement method, that is, the source code parsing. And, in this phase, no sources of uncertainty have been identified by the current work.

In this paper, we have identified a source of uncertainty that operates at another level. Indeed. Misconception of the measurement can generate a certain degree of uncertainty. This source of uncertainty can occur at the design phase (the definition of the measurement method).

According to this, the paper shows how an unaccounted for type conversion can affect the measurement of coupling and introduce uncertainty in the measurement result.
This uncertainty arises from phenomenon or elements unaccounted for during the design of the measurement.

So, even if the internal measurement process itself is a deterministic phenomenon, the measurement result can be affected by uncertainty and thus by errors.

This kind of error does not seem to fit exactly to the typology of uncertainty sources proposed in the theory of metrology (GUM 1995). However, the same theory teaches us how to measure this uncertainty experimentally. Maybe the "classical" metrology needs to be adapted to take into account the specificities of internal measurement.

7. Acknowledgement

This research project is supported by the European Union (ERFD) and the Walloon Region

(DGTRE) under the terms defined in the Convention n° EP1A12030000072-130008.

8. References

(Abran 1999) Abran,A., Jacquet, J.P., "A Structured Analysis of the New ISO Standard on Functional Size Measurement-Definition of Concepts", Fourth IEEE International Symposium and Forum on Software Engineering Standards, 1999, pp. 230-241.

(Briand 1999), Briand, L., Daly, J., Wust, J., "A Unified Framework for Coupling Measurement in Object-Oriented Systems", IEEE Transactions on Software Engineering, Volume 25 , Issue 1 (January 1999), IEEE Press, USA

(Cardelli 1985), Cardelli, L., Wagner, P.,"On Understanding Types, Data Abstraction, and Polymorphism", Computing Surveys, Vol 17 n. 4, pp 471-522, December 1985

(Chidamber 1994) Chidamber, S. R., AND Kemerer, C. F. "A Metric Suite for Object-Oriented Design" IEEE Transactions on Software Engineering 20, 6 (June 1994), 476–493.

(Eder 1994) Eder, J., Kappel, G., Schrefl, M., "Coupling and Cohesion in Object-Oriented Systems",Technical Report, University of Klagenfurt, 1994.

(GUM 1995) ISO, "Guide to the Expression of Uncertainty in Measurement". International Organization for Standardization, Printed in Switzerland, ISBN 92-67-10188-9, First Edition, 1993. Corrected and reprinted 1995

(Hitz 1995) Hitz, M., Montazeri, B., "Measuring Coupling and Cohesion in Object-Oriented systems", inProc. Int. Symposium on Applied Corporate Computing, Monterrey, Mexico, October 1995.

(ISO 9126) ISO, "ISO/IEC 9126 - Software Engineering- Product Quality Part 3 - Internal Metrics., 1999.

(NIST 1994), Barry, T., Kuyatt, C. "Guidelines for Evaluating and Expressing the Uncertainty of NIST Measurement Results", NIST Technical Note 1297 1994 Edition

(Pressman 1994), Pressman, R. S. "Software Engineering: A Practitioner's Approach", McGraw-Hill, 1994.

(Sun Java 2004), Java Language Specification, 2nd Edition, 2004, http://java.sun.com/docs/books/jls/second_edition/html/jIX.fm.html

(VIM 1993), ISO, International Vocabulary of Basic and General Terms in Metrology, International Organization for Standardization - ISO, Geneva, 1993

(Yourdon 1978), Yourdon, E., AND Constantine, L. L. "Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design", 2 ed. Yourdon Press, New York, 1978.