

The identification of anomalous code measures with conditioned interval metrics

Carlos López¹, Esperanza Manso², Yania Crespo²,

¹ Área de Lenguajes y Sistemas Informáticos, Universidad de Burgos
EPS Edif. C. C/Francisco de Vitoria S/N 09006 Burgos, Spain.
clopezno@ubu.es

² Departamento de Informática, Universidad de Valladolid
ETS de Ingeniería Informática. Campus Miguel Delibes 47011 Valladolid, Spain.
{manso,yania}@infor.uva.es

Abstract. Anomalous measurements are identified in the software measurement process using valid metrics intervals. In the particular case of code measurements, the same intervals are used independently of the nature of the problem solved by the entity being measured. Our proposal is to condition the measurement intervals according to the nature of the problem solved by the said code entity. By ‘nature’ we understand that which is expressed through standard UML classifier stereotypes. This paper identifies the requirements needed for a code measurement support tool to be able to take on this new perspective. Using these requirements as a basis, some existing tools are reviewed and the difficulty of applying this proposal with its current functionality is recognized. To this end, we present the adaptation of one of the reviewed tools (RefactorIt) and, in addition, the measurement process is applied to ten real projects, obtaining some initial intervals conditioned by the nature of the code entities.

Key words: Code metrics, Use intervals, Code measurement tools, Measurement process.

1 Introduction

Since the 1990s, software metrics and their associated measurement process, have attracted great interest in the software engineering community as a means of quantifying and controlling software quality [1] [2] [3]. According to [4], measuring is part of a process (see Figure 1) which consists of obtaining a numerical value for an attribute of a software product or process.

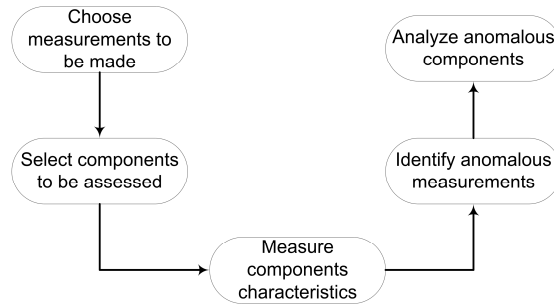


Figure 1 Measurement process defined by Sommerville [4].

In this process, the detection of anomalous entities is based on the identification of their anomalous measurements. The identification is performed in a pragmatic way by checking whether a particular measurement is within the range of recommended values. In general, the metrics used are those proposed by a quality model such as the standard ISO 9126 [5]; these metrics measure software products with different levels of abstraction, from analysis to code.

More precisely, the process can be applied to the code since it is a product in constant evolution and in need of constant maintenance [6]. This measurement process, as applied to the code, makes the evaluation of its quality easier. The evaluation of a code by means of metrics is not new. In fact, in the existing literature, there are a great many definitions of metrics, grouped according to different criteria, depending on the author. For instance, in the object-orientated paradigm, some well known sets of metrics on different code entities are:

- On classes: Chidamber and Kemerer [7], Lorenz and Kid [8].
- On subsystems: Robert Martin [9], Brito and Abreu [10].
- On methods: McCabe [11].
- Others mentioned by Piattini [12].

In the literature, there are also, however, many unfavourable criticisms concerning the application of metrics [13]. One of them is that the intervals used to identify anomalous measurements, obtained through empirical experiments, are restricted to the measuring context, thus limiting their use in other contexts. Even recommended intervals, taken from past measurements in the same context, cannot be used for code entities from different categories. This paper, then, aims to add to the knowledge on dependency that the context of the code entity may have in identifying anomalous measurement intervals. Upon this premise, and in the field of object orientation, a selection of categories can be based on the use of some UML classifier stereotypes. For instance, the analysis class stereotypes [14, 15] are: entity, control and boundary. In addition, the classification boundary criterium is, in turn, divided into: user interfaces, system interfaces and device interfaces. Other interesting stereotypes in the classifiers are those obtained as a result of some tasks performed by the development process, such as those related to exceptions, tests and utilities. This causes a subdivision of tasks within the measuring process which results in obtaining different use intervals for each stereotype considered. In short, the nature of the code entity is extracted from the following UML stereotypes: *exception*, *boundary* (*system*, *user* and *device interface*), *entity*, *control*, *test* and *utility*. Figure 2 shows the adaptation of

the classic measurement process when the new tasks, within the rectangle, are incorporated.

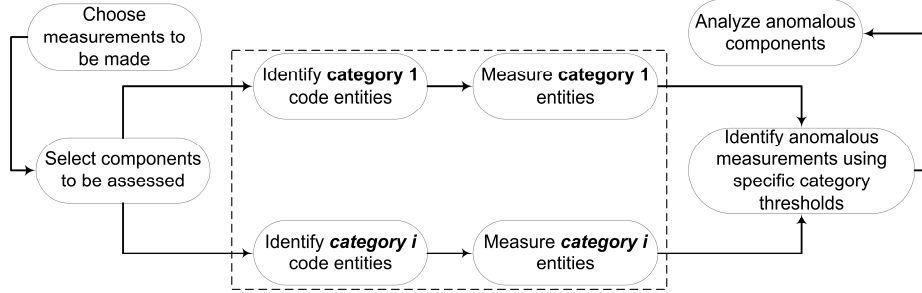


Figure 2 Measurement and modification process.

The rest of the paper is structured as follows: Section 2 introduces a preliminary evaluation of the functionality of code measurement tools and sets out the lack of any identification of anomalous measurements through conditioned use intervals. Section 3 describes the measurement process carried out using the *RefactorIt* tool [16] and the requirements for a code measurements tool to be able to take on this new perspective are analyzed. Section 4 proposes some conditioned use intervals obtained from measuring the code entities of ten real projects and applying statistical measurements to them. Finally, the conclusions and future lines of work are set out in Section 5.

2 Evaluating code tools

The measurement process needs tools to automatically perform the calculation of the values of the metrics for a particular code entity. For an easier preliminary understanding of the current functionalities of the code tools in **Table 1**, the result of the evaluation of a set of tools is shown with respect to the following characteristics:

- C1. Programming language on which the work is done.
- C2. Input: binary or source files (binary/source/both).
- C3. Number of metrics calculated (C31 Chidamber and Kemerer, C32 Lorenz and Kid, C33 Robert Martin)
- C4. Format for exporting results (html/txt/xml/xls).
- C5. Graphic indicators or grouping and filtering techniques to analyze results (Yes/No).
- C6. Configuration of metrics profiles.
- C7. Automatic classification of code entities.
- C8. Evaluation of multiple use intervals in the same evaluation.

The selection of the tools aims to evaluate a representative sample of tools available, according to the characteristics considered. The aim is to obtain a non-empty intersection of all the characteristics of the tools with all its possible values.

This will guarantee that it has an example of each of possible values of the characteristics.

Table 1 Code tools.

Tools	C1	C2	C3	C31	C32	C33	C4	C5	C6	C7	C8
Dependency Finder	java	binary	33	1	1	0	html,txt,xml	No	Yes	No	No
RefactorIt	java	sources	25	5	2	5	html,txt xml	Yes	Yes	No	No
JDdepend	java	binary	9	0	0	5	html,txt,xml	No	No	No	No
Eclipse Metrics - v1.3.6	java	sources	25	4	6	5	xml	No	Yes	No	No
NDepend	.NET	both	66	6	2	5	html, txt, xml, xls	Yes	Yes	No	No
SourceMonitor	java, C#, C++, VB	sources	14	0	0	0	txt, xml,	Yes	Yes	No	No

Although the definitions of many of the code metrics do not depend on the programming language, in practice, many tools or components only work on a single programming language (see column C1 of Table 1).

As for column C3, one of the criticisms made about experimentation with code metrics is the lack of any standardization in the definition of the metrics. This means that the measurement obtained for a particular code entity may vary according to whether it is calculated using one tool or other. This makes the comparison of values obtained using different tools impossible. Columns C31, C32, C33 show that none of the metric tools calculates all metrics of the authors considered.

The characteristic C6, profiles of metrics, refers to the tool's capacity to enable the user to configure the measurement intervals to detect anomalies. The majority of tools offer this characteristic. The last two characteristics, C7 and C8, are necessary for adapting the tools with the measurement process proposed in Figure 2. As can be seen in Table 1, none of the tools considered offer them.

The incorporation of the new tasks proposed in the process involves incorporating two new requirements in the tools: on the one hand, classifying the code entities in categories and, on the other hand, use intervals of metrics associated to each category considered. The first includes the definition of an open classification and computer aided classification mechanisms of the code entities. The tool will enable code entities to be evaluated using the recommended values for each category. The second involves a data gathering process to allow use intervals to be obtained.

3 Adaptation of the *RefactorIt* tool

The adaptation of the new measurement process requires tools to support these new activities. In this case, we have chosen to extend the RefactorIt tool [16]. From the initial set are discarded they do not measure Java code. Additionally they are discarded which are not open source and therefore can not be adapted. Moreover, it is

important the number of metrics that implement each set considered. These criteria leave RefactorIT and Eclipse Metrics as two possible candidates. The final selection criterion is based on the functionality offered RefactorIT on the metric profile management and user interface it offers.

RefactoIt is an *open source* tool that is used to inspect the *Java* code using code metrics and semantic rules. It has several forms of distribution: as a desktop tool or a *plugin of Eclipse*. In addition, it provides a catalog of refactorings which assist in the maintenance process.

The tool's basic measurement process is currently automated. **Figure 3** shows a screen with the result of the evaluation of an Eclipse project called “*RefactorItLab*” which is documented in [17]. The numbered rectangles show the parts of the graphic interface which serve as input for the basic activities of the measurement process. Furthermore, the unnumbered rectangles show the result of an evaluation with respect to the WMC (*Weighted Methods per Class*) metric and the identification of an anomalous measurement, in the *Network* class, with respect to the tool's recommended use interval [1-50].

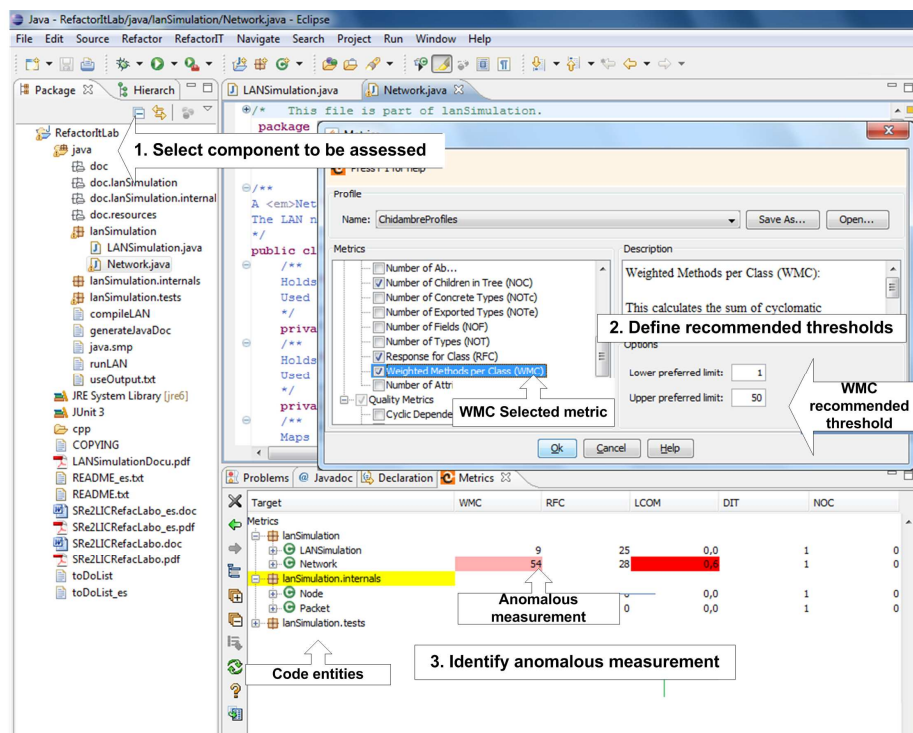


Figure 3 Measurement process with RefactorIt.

The following subsections analyze the new requirements that need to be incorporated in a tool for its adaptation to the new proposed measurement method and the particular adaptation of the *RefactorIt* tool, which will be referred to as

RefactorItUBU. Two videos showing the automatization of the new tasks for the measurement process can be found in [18].

3.1. Open classification of code entities

The initial hypothesis is that *the code entities may need different use intervals to detect anomalies depending on certain classifications*. A priori, the tools must provide some kind of mechanism, either automatic or manual, which allows the inspector to classify the entities in the categories that make up the classification.

Although there are classifications which may be considered standard, it is preferable that the tools should allow the inspector to define his/her own classifications. In this paper, we initially use a classification of code entities whose categories are based on the nature of the entity, expressed with standard stereotypes on UML classifiers: e_1 *exception*, e_2 *interface*, e_3 *entity*, e_4 *control*, e_5 *test*, e_6 *utility*.

The new functionality, which corresponds to the creation of a classification, has been added to the tool, defining a configuration file from which the different categories under consideration are extracted (*/refactorit_ubu/estereotipos.csv*). This classification will be used in two later activities: one, when the use interval of each metric is defined, and the other, when the measurement of the component is carried out. The following figures show, concretely, the functionality added to RefactorItUBU, supposing the following classification of categories for the file content: *Unknown*, *Exception*, *Interface*, *Control*, *Entity*, *Test* and *Utility*. **Figure 4** shows the new definition of use intervals for each metric, and in the lower righthand corner a panel is added which is labelled with each of the stereotypes and the use intervals recommended for each one. As happened with the tool's original functionality, these intervals have to be introduced by the user and can be stored in *profile* files [19]. Finally, **Figure 5** shows the evaluation of entities which allow the user to define the stereotype of each entity. From the inspector's point of view, it is interesting to point out that if the classification is not closed, a category called "*unknown*" should be considered.

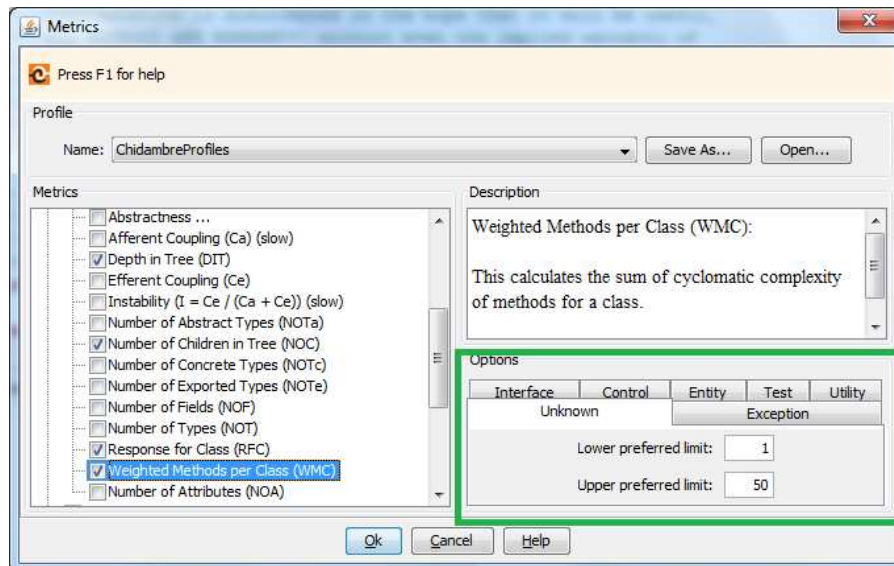


Figure 4 *RefactorItUBU* Definition of use intervals for each metric and stereotype considered.

Target	STR	WMC	RFC	DIT	NOC	LCOM
Metrics	Unknown					
lanSimulation	Unknown					
lanSimulation	Unknown	9	25	1	0	0,0
Network	Unknown	54	28	1	0	0,6
lanSimulation.internals	Unknown					
Node	Unknown	2	0	1	0	0,0
Packet	Unknown	2	0	1	0	0,0
lanSimulation.tests	Test					
LANTests	Test	38	39	3	0	0,0
LANTests\$PreconditionViolationTestCase	Test	2	4	4	0	0,0

Figure 5 *RefactorItUBU*: Inspection of code entities.

3.2. Classification of entities in the categories considered

When working with real systems, the number of code entities to be classified is very large. It is desirable that the inspector should have assistance to carry out this new activity efficiently. What is more, we start from the hypothesis that *the classification may be subjective and it is the inspector's responsibility to take the final decision on how to classify the code entities*. In this sense, the tool could provide two new functionalities:

- classification by entity groupings
- automatic classification

The application's architecture (layers, components) means that the code entities possess some logical groupings that must be identified by the inspector. In addition, this logical organization corresponds to the physical structure through the code

entities themselves. The physical grouping structures are: on the one hand, the packages, which contain packages and classes; and on the other hand, the classes, which contain methods. The application of a category on a grouping structure is propagated to the rest of the components. Thus, an application with a logical grouping marked by a three-layered architecture could be classified by indicating the category of the three packages that contain the superior levels of the architecture. Each change of category requires a new evaluation of the entity with the interval of the new stereotype chosen.

A code inspector would, in addition, want functionality with automatic classification methods that could later be adjusted by him/her. In this sense, we should mention, as an entity identification technique, the different name conventions used by software architects and programmers. For instance, the code entities whose name contains the literal strings “interface”, “gui”, “form”, etc. usually belong to the category e_2 *graphic interface*. The knowledge, based on name conventions, needed to identify entities may be generic with respect to design conventions or the programming language, or it may be specific knowledge of the project requirements. For instance, if there is a layer called “metric” in a metrics calculation project, in a first inspection, it could belong to the categories e_3 *entity* or e_4 *controllers*. The conventions of the libraries and the programming language itself may help to identify entities without any doubt, which is the case of the JUnit test and the *Java* exceptions. Table 2 shows a summary of the identification criteria presented which depend on the different categories of code entities considered: e_1 *exception*, e_2 *interface*, e_3 *entity*, e_4 *control*, e_5 *test*, e_6 *utility*.

The extension to *RefactorIt* performed here includes these functionalities. The change of category on a code entity grouping is propagated on the other entities it contains. In order to carry out this change from the use interface (see **Figure 5**), the new category has to be chosen from the pull-down list.

The automatic classification algorithm reads the conventions of names associated with each category considered from a configuration file. Thus, the inspector can customize the algorithm, introducing specific conventions from the context of the application being inspected. The specification will be made up of a quadruple <package convention, stereotype package, class convention, stereotype class>. For instance, the application of the name convention proposed by Junit is indicated by the following quadruple <“test”, “Test”, “Test”, “Test”>, which means that the packages containing the literal string “test” will be classified in the category of *Test*, and the classes containing the literal string “Test” will be classified in the category of *Test*. The exceptions are not grouped in packages, so to identify exceptions within the test package, the quadruple <“test”, “Test”, “Exception”, “Exception”> has to be added. Another example is the quadruple <“ui”, “Interface”, “listener”, “Control”>, in which all the entities in the package containing the literal string “ui” are classified in the category *Interface*, except those classes that contain the literal string “listener”, which are classified as *Control*. This algorithm is executed each time the inspector, through the user interface, requests a component to be measured. Once the classification of the code entities has been applied, conditioned intervals of recommended values, conditioned for each category, are used.

Table 2 Criteria for identifying stereotypes.

	e₁	e₂	e₃	e₄	e₅	e₆
Grouping	No	Yes	Yes	Yes	Yes	Yes
Generic Knowledge	Yes	Yes	Yes	Yes	Yes	Yes
Specific knowledge	No	No	Yes	Yes	No	No
Name conventions	exception	interface gui forms ui report swing visual awt	core model entity	control facade manager handler action callback provider	test debug dummy	utility properties log preference template options

4 Use intervals for entity types

In order to obtain a use interval for each category (e_1 *exception*, e_2 *interface*, e_3 *entity*, e_4 *control*, e_5 *test*, e_6 *utility*), measurements have been taken on ten plugins of the IDE of Eclipse, identifying the entities and selecting, for each measurement, the following percentile statistics: 25 (first quartile Q1) and 75 (third quartile Q3). The incorporation of this information into the tool is done by means of the functionality related to the specification of the use interval of a particular metric. If the tool has no multi-interval evaluation, the new measurement process to identify anomalous measurements may be excessively tedious, due to the need to repeat it as many times as there are use intervals under consideration.

4.1. Project selection

To select the projects to be measured and analyzed, the study has focused solely on plugins for the Eclipse tool, obtained through the open code software repository *SourceForge* (<http://sourceforge.net/>). Using the information provided by the said repository, several project selection criteria have been followed:

- Percentage of activity, measured using the information concerning the activity of continuous modifications and the recent activity. The criterium established for %Activity is $> 85\%$.
- Popularity, measured using the number of downloads by the users. The criterium established for N° of Downloads is > 7000 .
- State of development of the application, measured using the following ordinal scale: 1 Planning, 2 PreBeta, 3 Alpha, 4 Beta, 5 Production/Stable, 6 Maturity, 7 Inactive. The criterium established for state is ≥ 3

Another characteristic considered in the selection of study projects is that related to the type of programming language used in the implementation, which will be the same for all of them: Java.

Finally, it would seem to be of interest to consider the characteristic associated with the project size. The number of code entities for each category considered, depending on the nature of the problem, will be taken as a reference. The criterium is that the minimum number of entities should be greater than 400.

Table 3 Information concerning the set of projects measured.

Eclipse Plugins	Number of entities							Sourceforge information		
	e ₁	e ₂	e ₃	e ₄	e ₅	e ₆	unclassified	%Activity	N° Downloads	State
esFtp	1	90	38	68	8	86	52	94.19	45878	4
AVR	12	376	125	306	362	789	310	98.68	1464780	5.6
Jedit	25	1671	1657	1548	12	1719	681	99.62	5371954	5.6
EclEmma	1	257	288	78	186	35	150	99.93	1488061	5
AzSMRC	45	511	655	272	9	759	758	99.00	59327	4.5
EclipseME	39	575	951	535	281	1215	446	97.63	731177	5
ELBE	26	1788	1561	1216	138	380	288	85.95	30172	7
OpenReports	17	0	568	1074	2	159	383	96.80	235708	4.5
EclipseCorba	7	145	148	232	143	205	334	94.86	23062	3
LabelDecorator	7	0	116	34	52	177	74	85.00	7004	5
Total Code Lines	909	111818	85911	83876	14238	14797				

Table 3 shows the selected projects with the information concerning size and that provided by the open code software repository. The column “unclassified” shows the number of entities which could not be included in any of the categories considered. In addition, in order to get an overall idea of the size of the experiment, the last row of the table shows the total number of code lines analyzed of each type of entity considered.

4.2. Selection of metrics

The tool selected to obtain the measurements is *RefactorIt* [16]. **Table 4** shows the metrics provided by the tool and the recommended use intervals on some of them (columns MinValue and MaxValue). In addition, the column called scope shows which type of code entity the metric is associated with: package (P), class (C), method (M) or all the previous categories together (T).

Independently of the particular conventions on the code metrics, each time the code of a software system is analyzed, information concerning its size and complexity is required. Some works express the size of a system in terms of lines of code, number of classes and even quantity of megabytes of the source code. These numbers are only values of some basic metric. Unfortunately, after obtaining a set of individual values, there are still problems in characterizing the system or entity evaluated. The characterization through the metrics must serve to reflect the goodness of the main design aspects such as: size (SIZ), documentation (DOC), coupling (COU), inheritance (INH), structural complexity (COM), abstraction (ABS), cohesion (COH) or design principles (DP). Another column has been included in **Table 4**, called characteristic, which shows this subjective classification. In any case, these metrics

are considered of interest because of their relationship with different aspects of the quality of the software products [20-24].

Table 4 Set of metrics defined in the RefactorIt tool.

Description	Identifier	Min Value	Max Value	Scope	Characteristic
Cyclomatic Complexity	V(G)	1	10	M	COM
Density of Comments	DC	0.2	0.4	T	DOC
Executable Statements	EXEC	0	20	T	SIZ
Number of Parameters	NP	0	4	M	SIZ
Total Lines of Code	LOC	5	1000	T	SIZ
Abstractness	A	0.0	0.5	P	ABS
Afferent Coupling	Ca	0	500	P	COU
Depth in Tree	DIT	0	5	C	INH
Efferent Coupling	Ce	0	20	P	COU
Instability	I	0.7	1.0	P	COU
Number of Abstract Types	NOTa	0	20	P	ABS
Number of Children	NOC	0	10	C	INH
Number of Concrete Types	NOTc	0	80	P	ABS
Number of Exported Types	NOTe	3	50	P	COU
Number of Fields	NOF	0	1	C	SIZ
Number of Types	NOT	0	80	P	SIZ
Response for Class	RFC	0	50	C	COM
Weighted Methods per Class	WMC	1	50	C	COM
Number of Attributes	NOA	0	5	C	SIZ
Cyclic Dependencies	CYC	0	1	P	DP
Dependency Inversion Principle	DIP	0.3	1.0	C	DP
Direct Cyclic Dependencies	DCYC	0	1	P	DP
Distance from the Main Sequence	D	0.0	0.1	P	DP
Encapsulation Principle	EP	0	0.6	P	DP
Lack of Cohesion of Methods	LCOM	0.0	0.2	C	COH
Limited Size Principle	LSP	0	10	P	DP
Modularization Quality	MQ	0	1000	P	DP
Number of Tramps	NT	0	1	M	O

4.3. History of use interval metrics

From the data obtained in the experiment are calculated intervals of use for metrics considered. The lower and upper thresholds are calculated from Q1 and Q3, respectively, as indicated at the beginning of section 4. It starts from the premise that thresholds are calculated on the same domain applications, in this case Eclipse plugins, and are guidelines which do not ensure the existence anomaly on entity. The idea to calculate limits based on Q1 and Q3 have been used in previous studies that define rules for detecting design defects [25]. It also corroborates the improvement of the new measurement process as it is observed that the behavior of the metric is

different when considering the stereotypes and also the intervals obtained are more accurate.

The following tables, Table 5, Table 6 and Table 7, show the use intervals recommended depending on the classification considered: e_1 *exception*, e_2 *interface*, e_3 *entity*, e_4 *control*, e_5 *test*, and e_6 *utility*. The data from the study case corresponding to all the values of the metrics can be obtained from [18].

These data are from a case study in the area of software engineering, as a phase prior to the empirical validation based on experiments [26]. Having clarified this fact, we now summarize the results observed on applying the new measurement process proposed in this paper. In Table 6, the column showing the structural complexity metric WMC [7] is highlighted, as this indicates the limits of the use intervals for each of the stereotypes considered: e_1 [2,4], e_2 [4,16], e_3 [5,25], e_4 [3,16], e_5 [2,8], e_6 [4,22]. Table 4 shows [1, 50] the recommended use interval, with respect to the tool, for each metric. Two aspects of this information can be stressed: on the one hand, the interval proposed by the tool is very wide and includes all the other intervals. On the other hand, the intervals vary depending on the stereotypes considered: the classes of exception (e_1) and test (e_5) are less complex than the classes of controllers (e_4) and utility (e_6). Analogously, this analysis can be done for the rest of the measurements in the tables.

Table 5 Metrics of packages: Recommended intervals according to the nature of the problem.

		EXEC	Ca	Ce	I	A	D	NOTc	CYC	EP
Exception e_1	Q1	---	---	---	---	---	---	---	---	---
	Q3	---	---	---	---	---	---	---	---	---
Interface e_2	Q1	15.00	0.0	5.00	0.60	0.00	0.00	4	0	0.071
	Q3	134.00	7.0	18.00	1.000	0.07	0.36	18	3.0	0.80
Entity e_3	Q1	11.75	2.0	3.00	0.25	0.00	0.10	2	0	0.52
	Q3	194.50	35.5	14.00	0.68	0.26	0.50	12	4.5	1
Control e_4	Q1	7.50	0.0	2.00	0.31	0.00	0.00	1	0	0.00
	Q3	93.00	8.5	11.00	1.00	0.29	0.25	11	1.5	1
Test e_5	Q1	2.00	0.0	2.00	0.93	0.000	0.00	1	0	0.00
	Q3	28.00	1.0	7.75	1.0	0.39	0.34	7	0	0.19
Utility e_6	Q1	12.75	0.0	2.00	0.45	0.00	0.00	2	0	0.000
	Q3	127.25	6.0	14.00	1.00	0.17	0.40	13	1.0	0.81

Table 6 Metrics of classes: Recommended intervals according to the nature of the problem.

		DC	LOC	EXEC	WMC	DIT	RFC	LCOM	NOA
Exception e_1	Q1	0.00	10.25	0	1.75	3	1.75	0	0
	Q3	0.14	29.25	1.75	4.00	4	3.25	0.00	2
Interface e_2	Q1	0.00	35.00	2	4.00	1	2.00	0	1
	Q3	0.20	179.25	14.00	16.00	2	19.00	0.95	7
Entity e_3	Q1	0.00	40.00	2	5.00	1	2.00	0	1
	Q3	0.27	169.50	17.00	25.00	2	21.00	0.96	6
Control e_4	Q1	0.00	18.00	1	3.00	1	2.00	0	0
	Q3	0.21	115.25	12.00	16.25	2	17.00	0.75	3
Test e_5	Q1	0.00	19.00	0	2.00	1	2.00	0	0
	Q3	0.28	105.00	8.00	8.25	2	9.00	0.81	2
Utility e_6	Q1	0.02	33.00	2	4.00	1	3.00	0	1
	Q3	0.31	202.75	23.00	22.00	2	23.00	0.92	6

Table 7 Metrics of methods: Recommended intervals according to the nature of the problem.

		LOC	EXEC	NP	V(G)	NT
Exception e_1	Q1	1	0	0	1	0
	Q3	8	1	2	2	0.75
Interface e_2	Q1	1	0	0	1	0
	Q3	14	2	1	2	0
Entity e_3	Q1	1	0	0	1	0
	Q3	9	2	1	2	0
Control e_4	Q1	1	0	0	1	0
	Q3	11	2	1	3	0
Test e_5	Q1	1	0	0	1	0
	Q3	10	1	1	1	0
Utility e_6	Q1	1	0	0	1	0
	Q3	14	3	1	3	0

5 Conclusions and Future Lines of Work

In this paper, we have carried out a case study to prove our measurement process proposal. In this proposal, the measurement process [4] is modified by incorporating the inspector's/evaluator's knowledge of the code entity classification depending on its nature: e_1 *exception*, e_2 *interface*, e_3 *entity*, e_4 *control*, e_5 *test*, e_6 *utility*. As a result of the said code entity classification, a use interval has been proposed for each metric and category of the classification. In addition, the modifications of the process and the relationship with the code measurement tools have been analyzed. In particular, the result of adapting the new code entity measurement process to the *RecfactorIt* tool is presented.

This paper presents a collection of recommended intervals to identify anomalies, obtained as a result of measuring the code entities of a set of real projects. Our conclusion is that it is necessary to pursue research into the field opened up by this case study, and to this end, we make the following proposals for future work:

- It is necessary to replicate the case study with other sets of projects defined using the external factors and functionality. The aim of this is to refine and compare the proposed use intervals. In addition, it would be possible to evaluate whether our measurement process proposal depends on the context of the application to be measured or not.
- It is necessary to validate the proposed measurement process, and to this end we aim to study whether the use intervals of anomalous measurements are more accurate in the new scenario.
- Furthermore, we believe that, in order to validate our measurement process proposal, it would be interesting to incorporate new sets of metrics which would be useful from the point of view of software engineers [24].
- One of the problems found in practice has been the labelling of code entities, as they sometimes do not belong to only one stereotype. Thus, we propose elaborating a new fuzzy classification of code entities, in which each code

entity would have a tuple of weights corresponding to the degree of belonging to each stereotype.

- The improvements achieved in this work depend on the new task of classification of entities according to the stereotypes code considered. In this sense, it is necessary to carry out experiments that help to validate the consistency of classification by experts.

Acknowledgements. This paper has been funded by the Spanish ‘Ministerio de Ciencia e Innovación’ through the research project ROADMAP (TIN2008-05675).

References

1. Fenton, N.E. and S.L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*. 2nd ed. 1998: Course Technology. 656.
2. IEEE, C.S., *Guide to the Software Engineering Body of Knowledge: 2004 Edition - SWEBOK*. 2005.
3. Pressman, R.S., *Software Engineering: A Practitioner's Approach*. 6^a ed. 2005: McGraw-Hill. 958 p.
4. Sommerville, I., *Software Engineering*. 8th ed. 2005: Addison-Wesley. 864.
5. ISO/IEC, *Software engineering -- Product quality Part 1: Quality model*. 2001.
6. Marín, B., N. Condori-Fernández, and O. Pastor, *Calidad en modelos conceptuales. Un análisis multidimensional de modelos cuantitativos basados en ISO 9126*, in *Revista de Procesos y Métricas*. 2007.
7. Chidamber, S.R. and C.F. Kemerer, *A Metrics Suite for Object Oriented Design*. IEEE Transactions on Software Engineering, 1994. **20**: p. 476-493.
8. Lorenz, M. and J. Kidd, *Object-oriented software metrics: a practical guide*, ed. I. Prentice-Hall. 1994, Upper Saddle River, NJ, USA.
9. Martin, R., *OO Design Quality Metrics. An Analysis of Dependencies*. 1994.
10. Brito e Abreu, F. and R. Carapuça, *Candidate metrics for object-oriented software within a taxonomy framework*. Journal of Systems and Software, 1994. **26**(1): p. 10.
11. McCabe, T., *A Complexity Measure*. IEEE Transactions on Software Engineering, 1976. **2**: p. 308-320.
12. Piattini, M.G., *Calidad en el desarrollo y mantenimiento del software*, ed. F.O. García. 2002: Ra-Ma. 310 p. ; 24 cm.
13. Marinescu, R., *Measurement and quality in Object-Oriented Design*, in *Automatics and Computer Science*. 2002, Timișoara: Timișoara.
14. Arlow, J. and I. Neustadt eds. *Uml 2 And The Unified Process: Practical Object-oriented Analysis And Design*. 2005, Addison-Wesley Object Technology Series.
15. Jacobson, I., G. Booch, and J. Rumbaugh, *The Unified Software Development Process*. 1999: Addison-Wesley.
16. Aqris-Software, *RefactorIt*. 2001.
17. Demeyer, S., S.e. Ducasse, and O. Nierstrasz, *Object-Oriented Reengineering Patterns*. 2002: Morgan Kaufmann and DPunkt.
18. Carlos, L. *QAOOSE2010 Measurements and metric tool video*. 2010 [cited; Available from: <http://pisuerga.inf.ubu.es/clopez/QAOOSE2010/>]
19. Crespo, Y., et al., *Object-Oriented Design Knowledge: Principles, Heuristics and Best Practices*. 2006, Idea Group Publishing. p. 193-249.
20. Dromey, R.G., *A Model for Software Product Quality*, in *Transactions on Software Engineering*. 1995. p. 146-162.

21. Dromey, R.G., *Cornering the Chimera*. 1996. p. 33-43.
22. Briand, L.C., J. Wüst, and H. Lounis, *Replicated Case Studies for Investigating Quality Factors in Object-Oriented Designs*. Empirical Software Engineering, 2001. **6**(1): p. 11-58.
23. Moody, D.L., *Theoretical and practical issues in evaluating the quality of conceptual models: current state and future directions*. Data & Knowledge Engineering, 2005. **55**(3): p. 243-276.
24. Manso, E., *Estudio empírico para la validación de indicadores de la reusabilidad de diagramas de clases UML*. 2009, Valladolid.
25. Marinescu, R. *Detection Strategies: Metrics-Based Rules for Detecting Design Flaws*. in *Proc. ICSM 2004*. 2004.
26. Juristo, N. and A. Moreno, *Basics of Software Engineering Experimentation*. 2001: Kluwer Academic Publishers.