Metrics, Do They Really Help?

Serge Demeyer and Stéphane Ducasse

Software Composition Group,
Institut für Informatik (IAM), Universität Bern
Neubrückstrasse 10, CH-3012 Bern, Switzerland
{demeyer,ducasse}@iam.unibe.ch,
http://iamwww.unibe.ch/∼scg/
Apppeared in LMO'99

RÉSUMÉ. *Rendre mature la conception d'un cadre de conception nécessite des métriques permettant de conduire un processus de développement iteratif. Sur les bases d'une étude de VisualWorks – un cadre de conception pour la construction d'interfaces graphiques –, nous concluons que les métriques basés sur la taille des entités et l'héritage ne sont pas fiables pour détecter de problèmes de conception. Cependant, ces mêmes métriques sont utiles pour mesurer la stabilisation du cadre de conception.*

ABSTRACT. *Maturing a well designed framework requires a set of software metrics to steer the iterative development process. Based on a case study of the VisualWorks/Smalltalk framework for user-interface building, we conclude that today's size and inheritance metrics are not reliable to detect problems but are useful in measuring stability. We expect that this work will contribute to the application of metrics as a project management tool.*

MOTS-CLÉS : *Métriques, Cadre de Conception, Evolution, Rétro-Conception*
KEYWORDS: *Metrics, Frameworks, Evolution, Reengineering*

1. Introduction

Today, both research and industry pay a lot of attention to object-oriented application frameworks [JF88, FS97]. A framework is then defined as an artefact that represents a design to be reused in a family of applications. Framework designers specify variations within the framework design by means of hot spots; application developers refine the framework design to the needs of their application by filling in those hot spots.

Identifying the right combination of reusable abstractions and consequently coming up with the right set of hot spots is known to be difficult and is best achieved via an iterative development processes [GR95, JGJ97]. However, project managers are often reluctant to such an iterative approach because it is quite difficult to control without a reliable set of software metrics. To support project management, a metric programme should not only help in assigning priorities to the parts of the framework that need to be redesigned first, it should also tell whether the framework design is improving. That is, a metric programme should *detect problems* and *measure stability*.

To become a reliable project management support tool, metric programmes should be backed up by empirical data and case studies. From a literature survey, we learn that quite a number of object-oriented metrics have been proposed today, but that surprisingly little empirical data on their application is available [HS96, MN96, Mar97]. Moreover, no-one mentioned the application of metrics in a context of framework development. This shouldn't come as a surprise since framework technology is still relatively young and few mature industrial frameworks exist. Moreover, to validate a metrics programme against an iterative development approach, one must have access to different releases of the same framework and compare measurements on each of them. Also, existing frameworks represent important strategic assets of companies and as such the framework source code is almost never accessible without signing non-disclosure agreements. Summarized, it is hard to do empirical studies, it is even harder to publish results and consequently empirical studies are seldom compared nor reproduced.

This paper presents a case study about a set of metrics for managing an iterative framework development process. The empirical data have been gathered from the VisualWorks/Smalltalk user-interface framework, which is one of the few industrial scale frameworks that provides full access to the different releases of its source code. After providing some details about the case study and the metrics evaluated (section 2), the paper continues with an evaluation of metrics for problem detection (section 3) and stability measurement (section 4). Before coming to a conclusion (section 6), the paper reports on related work on object-oriented metrics (section 5).

2. The Experimental Set-up

Besides being an industrial framework that provides full access to different releases of its source code, the VisualWorks user-interface framework offers some extra features which make it an excellent case for studying iterative framework development. First, it is available to anyone who is willing to purchase VisualWorks/Smalltalk, which ensures that the results in this paper are *reproducible*[1]. Second, the changes between the releases are documented, which makes it possible to *validate experimental findings*. Finally, since the first three releases (1.0, 2.0 & 2.5) of the VisualWorks frameworks together represent a fairly typical example of a framework lifecycle [GHJV95], it is *representative*.

Indeed, VW1.0 –released in 1992– was pretty 'rough', but contained already all the core ingredients for a look-and-feel independent UI-builder. After releasing it to the market, addition of extra functionality demanded some major refactoring of the framework resulting in VW2.0 [Hau95]. The subsequent VW2.5 –released in 1995-

1. To allow other researchers to reproduce our results, we clarify that we define the VisualWorks framework as the part of the Smalltalk system corresponding with the categories 'Graphics-Visual Objects', 'Interface-*', 'UI-*' (excluding examples) and 'Globalization' when present. This definition is based on the manuals and [How95].

– included extensions to the framework which did not involve a lot of framework redesign and as such represents a consolidated design. Table 1 provides an overview of the evolution between the three releases. As can be seen, the shift from release 1.0 to 2.0 involved a lot of extra classes and methods, while the shift from 2.0 to 2.5 involved some extra methods and a few extra classes. This conforms with the documented amount of alterations between 1.0 and 2.0 (a lot) and 2.0 and 2.5 (a few).

	VW 1.0	VW 2.0	VW 2.5
Total number of methods	5282	7784	8305
Total number of classes	576	716	732

Table 1 : Overview of the evolution between VisualWorks releases[2].

To gather the necessary measurements, we have developed our own metrics tool named MOOSE/metrics. MOOSE is a re-engineering environment developed as part of the FAMOOS project (see http://www.iam.unibe.ch/∼famoos/); a project whose goal is to come up with a set of re-engineering techniques and tools to support the development of object-oriented frameworks. Metrics are examined as an essential part of framework re-engineering, which explains the motivation behind this case-study. The MOOSE/metrics tool extracts classes, methods and attributes from Visual-Works/Smalltalk source code and generates some tab separated ASCII files subsequently loaded in Microsoft/Access and Microsoft/Excel for post processing.

Table 2 lists the metrics evaluated in the study, including a short description and a reference to the definition of the metric. All of the metrics are proposed by Chidamber & Kimerer [CK94] ($\diamond$) or by Lorenz & Kidd [LK94] ($\star$). However, we rule out some of the proposed metrics because they received serious critique in the literature (LCOM and RFC [CK94]), because the definition isn't clear (MCX, CCO, CCP, CRE [LK94]; LCOM [CK94, EDL98]), because the lack of static typing in Smalltalk prohibits the computation of the metric (CBO [CK94]), because the metric is too similar with another metric included in the list (NIM, NCM and PIM in [LK94] resemble WMC-NOM in [CK94]), or simply because the metric is deemed inappropriate (NAC, SIX, MUI, FFU, FOC, CLM, PCM, PRC [LK94]).

2. A class and its meta-class are counted separately.

Name/Ref.	Description
Class Size (computed for each class)	
WMC-NOM ◇	Count all methods in class. This is the WMC metric of [CK94] where the weight of each method is 1.
WMC-MSG ◇	Sum of number of message sends in all method bodies of class.
WMC-NOS ◇	Sum of number of statements in all method bodies of class.
WMC-LOC ◇	Sum of all lines of code in all method bodies of class
NIV, NCV ⋆	Number of instance variables, number of class variables.
Method Size (computed for each method)	
Mthd-MSG ⋆	Number of message sends in method body.
Mthd-NOS ⋆	Number of statements in method body.
Mthd-LOC ⋆	Lines of code in method body.
Inheritance (computed for each class)	
HNL (DIT) ⋆ and ◇	Count number of classes in superclass chain of class. In case of multiple inheritance, count number of classes in longest chain.
NOC ◇	Number of immediate children of a class.
NMO ⋆	Number of methods overridden, i.e. redefined compared to superclass.
NME ◇	Number of methods extended, i.e. redefined in subclass by invoking a method with the same name on a superclass.
NMI ⋆	Number of methods inherited, i.e. defined in superclass and inherited unmodified.
NMA ⋆	Number of methods added, i.e. defined in subclass and not in superclass.

Table 2 : Overview of the metrics applied.

3. Metrics and Problem Detection

One important aspect of an iterative framework development process is *problem detection*, i.e. the ability to detect those parts of the framework that hinder future iterations. From a project management perspective, one would like to use a metric as some kind of fever thermometer, i.e. apply it to all parts of the framework and the parts that exceed a certain threshold value would be rated ‘too complex’ and need to be redesigned first.

To evaluate the usefulness of a given metric for problem detection, we apply them on one release and checked whether the parts that are rated ‘too complex’ are indeed improved in the subsequent release. If most of those parts are actually improved, it implies that the metric is reliable to detect flawed parts in the design. If most of the parts are not improved, it implies that the metric is unreliable, since it means parts are marked too complex that do not harm the natural evolution of the framework.

Moreover, we are also interested in the effect of threshold values. Since all of the metrics flag a part ‘too complex’ when it exceeds a certain threshold value, we wonder whether the threshold value affects the quality of the metric. Therefore, we run every test with several threshold values.

3.1. *Method Size Metrics*

Lorenz & Kidd propose method size metrics to identify methods that are 'too large' and that should be improved by making them smaller. To validate such a metric, we select all methods in one release that exceed a threshold value, doing several tests with different threshold values. For each of the methods that exceeds the threshold, we compare with the subsequent release to rank this method as either (i) removed (†), (ii) adapted so that the value of the measurement drops below the value of the threshold ($\searrow$), (iii) no meaningful decrease with respect to the measurement ($\leadsto$), or (iv) an increase in the measured value ($\nearrow$). Categories (i) and (ii) correspond then with an improvement of the method, while category (iv) corresponds with a method getting worse. Table 3 counts the number of methods for each of these categories with the threshold values proposed by Lorenz & Kidd.

	VW1.0	VW2.0			
# methods	threshold	removed	decreased	unchanged	increased
Mthd-MSG	$\geq 9 : 1162$	$\dagger : 315$	$\searrow (<9) : 48$	$\leadsto : 618$	$\nearrow : 181$
Mthd-NOS	$>7 : 779$	$\dagger : 231$	$\searrow (\leq 7) : 41$	$\leadsto : 390$	$\nearrow : 117$
Mthd-LOC	$>6 : 1751$	$\dagger : 408$	$\searrow (\leq 6) : 46$	$\leadsto : 1048$	$\nearrow : 249$
	VW 2.0	VW 2.5			
Mthd-MSG	$\geq 9 : 1712$	$\dagger : 84$	$\searrow (<9) : 25$	$\leadsto : 1514$	$\nearrow : 89$
Mthd-NOS	$>7 : 1124$	$\dagger : 51$	$\searrow (\leq 7) : 23$	$\leadsto : 988$	$\nearrow : 62$
Mthd-LOC	$>6 : 2614$	$\dagger : 143$	$\searrow (\leq 6) : 16$	$\leadsto : 2325$	$\nearrow : 130$

Table 3: Validation of method size metrics for problem detection.

Interpreting these data – and this is independent of the threshold value – we see that for all metrics, less than half of the methods identified as 'too large' were actually improved in a subsequent release. Moreover, a considerable number of the methods marked too large become even larger in a subsequent release. *Consequently, we conclude that method size metrics are not reliable to identify problematic methods in the VisualWorks framework.*

3.2. *Class Size Metrics*

Lorenz & Kidd, as well as Chidamber & Kemerer propose a number of class size metrics to identify the classes that are 'too large' and should be improved by factoring behaviour into other classes. Like we do for the validation of the method size metrics, we select all classes in one release that exceed a certain threshold value, doing several tests with different threshold values. For each of the classes that exceeds the threshold value, we compare with the subsequent release to rank this class as either (i) removed (†), (ii) adapted so that the value of the measurement drops below the value of the threshold ($\nearrow$), (iii) no meaningful decrease with respect to the measurement ($\leadsto$), or (iv) an increase in the measured value ($\nearrow$). Categories (i) and (ii) correspond then with an improvement of the class, while category (iv) corresponds with a class getting worse.

Table 4 counts the number of classes for each of these categories with the threshold values as proposed by Lorenz & Kidd (NIC, NCV), or with the threshold value being 20% of the maximum measurement (all others).

Interpreting the data of Table 4 – and this is again independent of the threshold value – we see that for most of the metrics (WMC-NOM, WMC-LOC, NIV, NCV), less than one third of the classes identified as ‘too large’ are actually improved in a subsequent release.

	VW1.0	VW2.0			
# classes	threshold	removed	decreased	unchanged	increased
WMC-NOM	$\geq 27 : 48$	$\dagger : 8$	$\searrow(<27) : 3$	$\leadsto : 9$	$\nearrow : 28$
WMC-MSG	$\geq 451 : 15$	$\dagger : 5$	$\searrow(<451) : 2$	$\leadsto : 0$	$\nearrow : 8$
WMC-NOS	$\geq 246 : 18$	$\dagger : 6$	$\searrow(<246) : 2$	$\leadsto : 0$	$\nearrow : 10$
WMC-LOC	$\geq 341 : 21$	$\dagger : 6$	$\searrow(<341) : 1$	$\leadsto : 3$	$\nearrow : 11$
NIV	$\geq 3 : 84$	$\dagger : 6$	$\searrow(<3) : 1$	$\leadsto : 55$	$\nearrow : 22$
NCV	$\geq 3 : 30$	$\dagger : 11$	$\searrow(<3) : 2$	$\leadsto : 16$	$\nearrow : 1$
	VW2.0	VW2.5			
WMC-NOM	$\geq 33 : 64$	$\dagger : 2$	$\searrow(<33) : 1$	$\leadsto : 29$	$\nearrow : 32$
WMC-MSG	$\geq 565 : 15$	$\dagger : 0$	$\searrow(<565) : 0$	$\leadsto : 5$	$\nearrow : 10$
WMC-NOS	$\geq 300 : 19$	$\dagger : 0$	$\searrow(<300) : 0$	$\leadsto : 6$	$\nearrow : 13$
WMC-LOC	$\geq 423 : 26$	$\dagger : 0$	$\searrow(<423) : 0$	$\leadsto : 6$	$\nearrow : 20$
NIV	$\geq 3 : 130$	$\dagger : 7$	$\searrow(<3) : 1$	$\leadsto : 110$	$\nearrow : 12$
NCV	$\geq 3 : 36$	$\dagger : 0$	$\searrow(<3) : 0$	$\leadsto : 34$	$\nearrow : 2$

Table 4: Validation of class size metrics for problem detection.

The WMC-MSG and WMC-NOS metrics fare a little better, but still less than half of the classes are improved. Finally, for all metrics a considerable number of the classes marked too large become even larger in a subsequent release. *Consequently, we conclude that class size metrics are unreliable for detecting problematic classes in the VisualWorks framework.*

To understand the better performance of the WMC-MSG and WMC-NOS compared to WMC-NOM and WMC-LOC, we computed the correlation between each of the WMC-metrics and found a remarkably high correlation . This suggests that, for a first approximation the WMC-NOM or WMC-LOC metrics –which can do without expensive parsing technology– are sufficient. However, in Table 4 we see that for both the WMC-MSG and WMC-NOS almost no classes fall in the ‘unchanged category’ ($\leadsto$). Thus, for a more detailed analysis, the difference between the WMC metrics is relevant and makes the extra cost of computing the metric worthwhile.

3.3. *Inheritance Metrics*

Lorenz & Kidd, as well as Chidamber & Kemerer propose a number of inheritance metrics to identify the classes that have a large impact on the inheritance hierarchy and are potential candidates for further analysis and change. Like we do for the validation of the method and class size metrics, we select all classes in one release that exceed a certain threshold value, doing several tests with different threshold values. For each of the classes that exceeds the threshold value, we compare with the subsequent release to rank this class as either (i) removed ($\dagger$), (ii) adapted so that the value of the measurement drops below the value of the threshold ($\searrow$), (iii) no meaningful decrease with respect to the measurement ($\leadsto$), or (iv) an increase in the measured value ($\nearrow$). Categories (i) and (ii) correspond then with an improvement of the class, while category (iv) corresponds with a class getting worse. Table 5 counts the number of classes for each of these categories with the threshold values as proposed by Lorenz & Kidd (HNL, NMO), or with the threshold value being 20% of the maximum measurement (all others). Note that the NMI metric should be maximised, i.e. the larger the measurement, the better the class.

	VW1.0	VW2.0			
# classes	threshold	removed	decreased	unchanged	increased
HNL	$\geq 6: 170$	$\dagger: 28$	$\searrow < 6: 0$	$\leadsto: 96$	$\nearrow: 46$
NOC	$\geq 5: 36$	$\dagger: 4$	$\searrow < 5: 4$	$\leadsto: 16$	$\nearrow: 12$
NMO	$\geq 3: 199$	$\dagger: 29$	$\searrow < 3: 4$	$\leadsto: 99$	$\nearrow: 67$
NME	$\geq 2: 76$	$\dagger: 6$	$\searrow < 2: 1$	$\leadsto: 52$	$\nearrow: 17$
NMI	$\leq 35: 375$	$\dagger: 39$	$\searrow > 35: 25$	$\leadsto: 296$	$\nearrow: 15$
NMA	$\geq 25: 42$	$\dagger: 5$	$\searrow < 25: 5$	$\leadsto: 10$	$\nearrow: 22$
	VW2.0	VW2.5			
# classes	threshold	(i)	(ii)	(iii)	(iv)
HNL	$\geq 6: 220$	$\dagger: 14$	$\searrow < 6: 0$	$\leadsto: 206$	$\nearrow: 0$
NOC	$\geq 8: 10$	$\dagger: 0$	$\searrow < 8: 0$	$\leadsto: 8$	$\nearrow: 2$
NMO	$\geq 3: 263$	$\dagger: 13$	$\searrow < 3: 2$	$\leadsto: 213$	$\nearrow: 35$
NME	$\geq 4: 33$	$\dagger: 1$	$\searrow < 4: 0$	$\leadsto: 24$	$\nearrow: 8$
NMI	$\leq 46: 454$	$\dagger: 19$	$\searrow > 46: 9$	$\leadsto: 425$	$\nearrow: 1$
NMA	$\geq 31: 51$	$\dagger: 2$	$\searrow < 31: 1$	$\leadsto: 25$	$\nearrow: 23$

Table 5: Validation of inheritance metrics for problem detection.

Interpreting these data – and this is again independent of the threshold value– we see that for all of the metrics, less than half of the classes identified as ‘large impact on inheritance tree’ are actually improved in a subsequent release. *Consequently, we conclude that inheritance metrics are unreliable for detecting problematic classes in the VisualWorks framework.*

3.4. *Evaluation*

For all metrics evaluated in the study, more than half of the framework parts rated ‘too complex’ did not harm the natural evolution of the framework. Because this observation is independent of the threshold value, we conclude that method size, class size and inheritance metrics are not reliable to detect flawed parts in the framework design.

Note that the fact that the first release of the VisualWorks framework is quite well-designed, does not invalidate our results. Indeed, even in a assumed ‘good’ design, inferior parts would have been improved if they obstruct the natural framework evolution.

Further investigating these results, we analysed the distribution of the metric values. In general, most of the framework parts in the VisualWorks framework are small and there tend to be only a few large outliers. To put it in numbers, more than 80% of the framework parts occupy the lowest 20% of the distribution scale; less then 20% classify as large. In the VisualWorks case study, these larger parts represent functionality that should be used as a whole, hence need not be factored out in smaller pieces. Assuming that any framework somehow must contain some larger pieces, we forward the rule of the thumb that *"any metric that results in a 80/20 distribution for a particular framework design, should not be used for problem detection"*.

4. Metrics and Stability Measurement

A second important aspect of an iterative framework development process is *stability measurement*, i.e. the ability to assess where and how much of the design of the framework is changing and from this information deduce whether the design of the framework is improving. From a project management perspective, one would like to use a metric as a retrospection tool, i.e. check which parts of the framework have been changed and deduce which parts of the framework have been consolidated.

To evaluate the usefulness of a given metric for stability measurement, we measure the differences between two subsequent releases. For those parts of the framework that have changed the measurement, we perform a qualitative analysis by comparing the identified changes with the documentation and the source code. In general, we expect the amount of changes between VW1.0 and VW2.0 to be more than the amount of changes between VW2.0 and VW2.5.

4.1. *Method Size Metrics*

As a first assessment of the method changes with every release, we compare the total number of methods in a release with the number of methods that were (i) removed, (ii) added or (iii) retained in the subsequent release. This classification –shown in Table 6– is based entirely on naming, so we do not take into account renamed methods. Thus, a method is classified 'removed' when its owning class is removed, or when a method with the same name does not exist on the same class in the subsequent release. A method is classified 'added' when its owning class is added, or when a method with the same name does not exist in the previous release. A method is classified 'retained' when the owning class is retained in both releases and when a method with the same name is defined in both releases. Classification of classes in the same categories is defined on naming as well (see section 4.2).

VW1.0	VW2.0			VW2.0	VW2.5		
total	removed	added	retained	total	removed	added	retained
5282	1067	3569	4215	7784	337	858	7447

Table 6: Methods removed / added / retained.

Next, for all methods classified 'retained' (RM), we apply the method size metrics to check the difference in measurement, classifying them into the categories (i) decrease, (ii) equal and (iii) increase. Table 7 shows the resulting classification.

VW1.0→VW2.0				
	RM	decrease	equal	increase
Mthd-MSG	4215	361-8.6%	3643-86.4%	211-5.0%
Mthd-NOS	4215	327-7.7%	3726-88.4%	162-3.8%
Mthd-LOC	4215	448-10.6%	3576-84.8%	191-4.5%
VW2.0→VW2.5				
Mthd-MSG	7447	186-2.5%	7146-95.9%	115-1.5%
Mthd-NOS	7447	168-2.5%	7188-96.5%	91-1.2%
Mthd-LOC	7447	205-2.7%	7150-96.0%	92-1.2%

Table 7: Validation of method size metrics for stability measurement.

Interpreting the data in Table 6, we see that the amount of methods removed-added is indeed larger with VW1.0→VW2.0 than it is with VW2.0→VW2.5. Also, the ratio between the number of methods retained and the number of methods added/removed is far better with VW2.0→VW2.5, revealing that the 2.5 release is indeed stabilizing. Moreover, Table 7 tells us that most of the retained methods did not change, but that there were more changes with VW1.0→VW2.0. For the qualitative analysis, we checked the removed methods against the release notes and discovered undocumented removals. Also, we checked the retained methods with the largest decrease in measurement and discovered a number of interesting refactorings. For instance, we have detected the introduction of ”template methods” [GHJV95] where methods have been cloned and patched in the subclass. We also detected refactorings where behaviour has been delegated to new classes.

4.2. *Class Size Metrics*

As a first assessment of the class changes with every release, we compare the total number of classes in a release with the number of classes that were (i) removed, (ii) added or (iii) retained in the subsequent release. This classification –shown in Table 8– is based entirely on naming, so we do not take into account renamed classes. Thus, a class is classified ‘removed’ when a class with the same name does not exist in the subsequent release; a class is classified ‘added’ when a class with the same name does not exist in the previous release and finally a class is classified ‘retained’ when a class with the same name is defined in both releases.

VW1.0	VW2.0			VW2.0	VW2.5		
total	removed	added	retained	total	removed	added	retained
576	70	210	506	716	38	54	678

Table 8: Classes removed / added / retained.

Next, for all classes classified ‘retained’ (RC), we apply the class size metrics to check the difference in measurement, classifying them into the categories (i) decrease, (ii) equal and (iii) increase. Table 9 shows the resulting classification.

	VW1.0→VW2.0			VW2.0→VW2.5		
	decrease	equal	increase	decrease	equal	increase
WMC-NOM	184-36.4%	282-57.7%	40-7.9%	98-14.4%	569-83.9%	11-1.6%
WMC-MSG	196-38.7%	238-47.0%	72-14.2%	128-18.9%	514-75.8%	36-5.3%
WMC-NOS	206-40.7%	235-46.4%	65-12.8%	124-18.3%	523-77.1%	31-4.6%
WMC-LOC	234-46.2%	213-42.1%	59-11.7%	139-20.5%	509-75.0%	30-4.4%
NIV	39-7.7%	462-91.3%	5-1.0%	21-3.1%	654-96.5%	4-0.4%
NCV	17-3.4%	483-95.4%	6-1.2%	3-0.4%	673-99.3%	2-0.3%

Table 9: Validation of class size metrics for stability measurement.

Interpreting the data in Table 8, we see that the amount of classes removed-added is indeed larger with VW1.0→VW2.0 than it is with VW2.0→VW2.5. Also, the ratio between the number of classes retained and the number of classes added/removed is far better with VW2.0→VW2.5, revealing that the 2.5 release is indeed stabilizing. Moreover, Table 9 provides an interesting overview of the way the retained classes did change. With VW1.0→VW2.0, a small yet significant amount of instance variables and class variables have been added/removed, while VW2.0→VW2.5 these values drop to almost zero. Also looking at the changes in WMC measurements more than half of the classes have been affected by the VW1.0→VW2.0 redesign. But, the amount of classes changed in VW2.0→VW2.5 has decreased below 25%, thus the framework modification had much more local impact then. For the qualitative analysis, we checked the removed classes against the release notes and could verify all removals as true removals or renaming. Also, we checked the retained classes with the largest decrease in measurement and discovered classes where functionality has been delegated to other classes.

4.3. *Inheritance Metrics*

To assess the inheritance changes for every release, we apply the inheritance metrics on all classes classified ‘retained’: 506 for VW1.0→VW2.0 and 678 for VW2.0→VW2.5. Again, we checked the difference in measurement, classifying them into the categories (i) decrease, (ii) equal and (iii) increase. Table 10 shows the resulting classification.

	VW1.0→VW2.0			VW2.0→VW2.5		
	decrease	equal	increase	decrease	equal	increase
HNL	50-9.9%	444-87.7%	12-2.4%	2-0.3%	676-99.7%	0-0.0%
NOC	48-9.5%	432-85.4%	26-5.1%	18-2.6%	640-94.4%	20-2.9%
NMO	112-22.1%	369-72.9%	25-4.9%	42-6.2%	629-92.8%	7-1.0%
NME	54-10.7%	439-86.7%	13-2.6%	25-3.7%	652-96.2%	1-0.1%
NMI	288-57.1%	203-40.1%	15-3.0%	297-43.8%	380-56.0%	1-0.1%
NMA	150-29.6%	324-64.0%	32-6.3%	73-10.8%	592-87.3%	13-1.9%

Table 10: Validation of inheritance metrics for stability measurement.

The data in Table 10 provides some interesting insights in the way the refactoring affected the inheritance tree. With VW1.0→VW2.0, a significant amount of classes experienced a change in HNL, implying that the inheritance tree has been refactored. However, with VW2.0→VW2.5 all classes retain their HNL value, thus the inheritance structure remained intact. Nevertheless, there are changes in NOC values, indicating that the inheritance tree has been changed by adding or removing leaves. This is a positive sign, showing that the inheritance tree in VW2.0 has indeed stabilized.

The changes in NMO and NME confirm that the framework is stabilizing (i.e., less changes with VW2.0→VW2.5 than with VW1.0→VW2.0). Additionally, the changes in NMI provides an excellent understanding of the way the changes ripple through the inheritance tree. Indeed, we see that for both VW1.0→VW2.0 and VW2.0→VW2.5 almost half of the classes have changed their NMI value, thus inherit more or less methods. Yet, the change in WMC-NOM value are quite low (see Table 9), thus the methods must have been added or removed high in the inheritance tree to affect that many subclasses.

4.4. *Evaluation*

Based on the qualitative analysis (i.e., checking documentation and source-code) of the parts of the framework that did change, we conclude that (i) all documented changes are detected; (ii) some undocumented changes are detected; and (iii) changes correspond to new functionality or refactorings. *Thus, metrics are a reliable basis to identify changes and can be used to measure stability.*

The more interesting results however, is that for the method and class size metrics, decreases in measurement point to places where behaviour has been refactored. Also, interpretation of changes in inheritance values reveals a lot about the stability of the inheritance tree. *Thus, change metrics can be used to assess the quality of a design.*

5. Related Work

Very few empirical studies on object-oriented metrics are published so far, and all apply to object-oriented programs rather than to object-oriented frameworks [CK94, LH93, LK94, MN96].

We are aware of other metric programmes proposed in the literature and we suspect that some of them will be more reliable in detecting problematic parts of the framework design. Cost estimation metrics like function points and COCOMO [FP97], are of course not relevant. But, metrics dealing with coupling and cohesion – such as Lack of Cohesion in Methods (LCOM), Coupling between Object Classes (CBO) [CK94, EDL98], Data Abstraction Coupling (DAC), Message Passing Coupling (MPC) [LH93] – seem quite promising. Yet, we did not include these in the case study for a number of reasons. First, some definitions are far from precise and not well-formulated (LCOM, DAC, MPC) – although some rectifications have been published in the literature [CS95, HM95, HM96]. Second, as Smalltalk is not statically typed, it is impossible to accurately compute the CBO metric based on source-code alone (although Chidamber & Kimerer claim they did but without describing how they inferred the type [CK94]). Third, metric definitions like LCOM or CBO in [CK94] are subject to controversy in the literature [LH93, HM95, HM96]. In the case of the Chidamber's definitions of LCOM, our own experiments confirmed the unreliability of the metric reported in [HM96]. For instance, the impact of attribute access via accessor methods – which is considered a good practice in Smalltalk [KST96]– is not taken into account into the metric definition.

6. Conclusion

Based on a case study of a set of object-oriented software metrics against three releases of a medium sized framework (VisualWorks/Smalltalk), we conclude that today's size and inheritance metrics are not reliable for detecting concrete problems in the framework design. However, those metrics are very good for measuring the differences between two releases and as such can be used to measure stability.

Concerning the application of metrics for problem detection, we point out that the distribution of the measurements showed that 80% of the parts occupy the lowest 20% of the distribution scale. From this observation, we forward the rule of the thumb that ”any metric that results in a 80/20 distribution for a particular framework design, should not be used for problem detection”. In the near future, we plan to confirm this rule on other frameworks, but in the meantime we invite other researchers to validate or invalidate this rule.

Our work answers the need for good project management tools when developing a framework. Indeed, because of the iterative nature of framework development, reliable project management tools are indispensable. By showing the positive effects of metric programmes, we hope we will persuade software teams to incorporate them into their usual working habits.

References

[CK94] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[CS95] N. I. Churcher and M. J. Shepperd. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 21(3):263–265, Mar. 1995.

[EDL98] L. Etzkorn, C. Davis, and W. Li. A practical look at the lack of cohesion in methods metric. *Journal of Object-Oriented Programming*, 11(5):27–34, Sept. 1998.

[FP97] N. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, second edition, 1997.

[FS97] M. E. Fayad and D. C. Schmidt. Object-oriented application frameworks (special issue introduction). *Communications of the ACM*, 40(10):39–42, Oct. 1997.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[GR95] A. Goldberg and K. S. Rubin. *Succeeding With Objects: Decision Frameworks for Project Management*. Addison-Wesley, Reading, Mass., 1995.

[Hau95] J. Haungs. A technical overview of visualworks 2.0. *The Smalltalk Report*, pages 9–14, Jan. 1995.

[HM95] M. Hitz and B. Montazeri. Measure coupling and cohesion in object-oriented systems. *Proceedings of International Symposium on Applied Corporate Computing (ISAAC'95)*, Oct. 1995.

[HM96] M. Hitz and B. Montazeri. Chidamber and kemerer's metrics suite; a measurement perspective. *IEEE Transactions on Software Engineering*, 22(4):267–271, Apr. 1996.

[How95] T. Howard. *The Smalltalk Developer's Guide to VisualWorks*. SIGS Books, 1995.

[HS96] B. Henderson-Sellers. *Object-Oriented Metrics: Mesures of Complexity*. Prentice-Hall, 1996.

[JF88] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[JGJ97] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse*. Addison-Wesley/ACM Press, 1997.

[KST96] E. J. Klimas, S. Skublics, and D. A. Thomas. *Smalltalk with Styles*. Prentice-Hall, 1996.

[LH93] W. Li and S. Henry. Object oriented metrics that predict maintainability. *Journal of System Software*, 23:111–122, 1993.

[LK94] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: A Practical Approach*. Prentice-Hall, 1994. (2).

[Mar97] R. Marinescu. *The Use of Software Metrics in the Design of Object-Oriented Systems*. Diploma thesis, University Politehnica Timisoara - Fakultat fur Informatik, Oct. 1997.

[MN96] S. Moser and O. Nierstrasz. The effect of object-oriented frameworks on developer productivity. *IEEE Computer*, pages 45–51, Sept. 1996.