Platform Library Complexity

Anestis A. Toptsis
Dept. of Computer Science and Engineering
York University
Toronto, Canada
anestis@cse.yorku.ca

Jensen Emilrazan
Dept. of Computer Science and Engineering
York University
Toronto, Canada
cs243048@cse.yorku.ca

***Abstract*— We present a suite of metrics that can be used to evaluate the complexity of a library of a software platform. We then use those metrics to evaluate parts of two major platforms – .NET and Java. Our findings indicate that each of the two platforms has its own strengths and weaknesses, and none of the two is clearly superior to the other.**

Keywords- Software Metrics, Platform Complexity.

I. INTRODUCTION AND BACKGROUND

When developing software (s/w), a main issue is to measure the complexity of the resulting s/w product. This is useful especially for the maintenance phase of the s/w lifecycle. The maintenance phase of the s/w lifecycle is the most long-lived part of every s/w product. Obviously, if the s/w is easy to understand, it will be easier to maintain it during the maintenance phase. Therefore, the notion of *complexity* of s/w has been established. In short, *s/w complexity* is the degree of how difficult a program is to comprehend and work with. Toward this end, many software *metrics* have been developed. ways to measure (and predict) s/w complexity. A metric is basically a number that indicates how complex a s/w artifact may be. For example, if s/w artifacts S1 and S2 have metrics m1 and m2 and m1 > m2 then S1 is more complex than S2. Similar to program complexity, metrics have been devised for the structural complexity of other software artifacts, such as XML schemas and UML models. This type of metrics is typically referred to as *structure metrics*.

Overall, software metrics is a 35+ years old research area. Some of the earliest works are [1], [2], [3], [4], [5], [6], [7], [8]. Work in software metrics continued through the 1980s, including attempts to axiomatize the area of software complexity measures [9], and works on meta-metrics (i.e., attempts to evaluate the effectiveness of existing software metrics) such as [10] and [11], and the introduction of metrics inspired by information theory [12]. The increasing popularity of object-oriented (OO) programming and XML in the 1990s and 2000s led to new works on metrics, aiming to measure the complexity of representative entities of the OO paradigm, [13], [14], [15]. All the above works deal with measuring the complexity of the end-product, i.e., the complexity of a program developed in a structured or in an OO programming language, or the complexity of the design of a software artifact developed in UML or XML. Interestingly, there seems to be a complete lack of reported works that measure the complexity of the tools that are used to develop such end-products. In this work we address a different aspect of complexity. Namely, we attempt to evaluate the *complexity of the tools* used to create the software artifacts, rather than the complexity of the artifacts themselves. Although the complexity of a software artifact may determine the degree of its usability, and this is clearly important, we argue that the usability of the tool that is used to build the artifact is also important – and may be even more so.

In this vein, we present our work from evaluating the complexity of two major software development platforms – the .NET platform [16] and the Java platform [17]. It is probably broadly agreed that these two platforms are used to create most of the software artifacts developed during the last decade, as well as the ones that are currently under development. The rest of the paper is organized as follows.

Section II presents our criteria (metrics) for evaluating platform complexity. Section III presents the results of our evaluation. Section IV is the conclusion.

II. PLATFORM METRICS

The two platforms that are being analyzed are the .NET (Framework 3.5) and Java 2 (Standard Edition v1.4.2). Since the .NET framework is composed of many different programming languages we choose to focus on the integral C# components. This provides a fair and equal assessment when compared with the Java platform. Hereafter, the term library is used to refer to packages (of the Java platform) and namespaces (of the .NET platform).

The *IO*, *GUI*, *Network*, *Regular Expressions*, and *Reflection* libraries are chosen for our comparisons since they represent integral components of both the .NET and Java libraries as well as they provide similar functionality in both platforms.

Library Overview.

- *IO* – "Provides classes for system input and output through data streams, serialization and the file system." [18]
- *GUI* – "Provides classes for rich user interface features." [18]
- *Network* – "Provides classes for implementing network applications." [18]
- *Regular Expressions* – "Provides classes for matching character sequences against patterns specified by regular expressions." [18]
- *Reflection* – "Provides classes and interfaces for obtaining reflective information about classes and objects." [18]

The following measures are adopted for our suite:

- Platform overview.
- Entropy.
- Ambiguity.
- Type metrics.
- API Documentation Properties.

In what follows, we provide a description and a discussion of each of the above metrics.

Platform Overview.

This measure provides an extensive look at the libraries for the .NET and Java platforms. The following analysis provides an overview of how detailed one library is when compared with another library. The following components are analyzed:

- *Classes* – the total number of classes for a specified library.
- *Methods* – the total number of methods for a specified library.
- *Methods per Class* – the average methods per class for a specified library.

Note, the Platform Overview is a two pronged metric. On the one hand, more classes and more methods indicate a complicated library, harder to grasp, harder to manage, and a higher cognitive load on the end-programmer. On the other hand, a rich library provides additional features to the end-programmer and has the possibility of making programming much easier.

Entropy.

The entropy metric aims to give a viable measure of how complex a particular software product, or in this case a platform/library, is to use and maintain. Entropy has been used on multiple occasions in the past as a metric for a variety of software artifacts. Example works are [11] and [12]. The main idea behind this metric is that the information content portrayed by a given platform/library does indeed provide a valid assessment of how complex a platform/library truly is. Generally speaking, the lower the entropy, the less complex a given platform/library is. The entropy metric is calculated as follows:

$$H = -\sum_{i=1}^{N} P_i \cdot \lg P_i \text{ , where}$$

$$P_i = P(C_i) = \frac{\text{number of nodes of class } C_i}{\text{total number of nodes in the graph}}$$

N = total number of classes C_i, i.e., total number of P_i s.

C_i: Equivalency class. This is the set of all nodes such that any two nodes N_i and N_j of a graph G (constructed to resemble class inheritance relationships of the corresponding library; the nodes of G correspond to classes and two nodes are connected by an edge if they are related by a class-subclass relationship in their library) have indegree(N_i) = indegree(N_j) *and* outdegree(N_i) = outdegree(N_j).

Ambiguity.

The ambiguity metric aims to give a sense of how unclear a particular library is. The level of ambiguity is determined based on the number of overloaded methods in a class per unit library. The higher the level of ambiguity, the less clear a particular library is. For the purposes of this assessment we have chosen to analyze the same set of libraries once again. Moreover, the ambiguity metric presents us with how complex a particular library is. The following components are analyzed.

- *Class Ambiguity* - the average number of overloaded methods per class.
- *Method Ambiguity* – the average number of times a method is overloaded per method per class.
- *Net Ambiguity* – the average net ambiguity rating; if a method in a class is overloaded a rating of one is given to that particular class, otherwise a rating of zero is given. The larger the net ambiguity the more ambiguous a particular library is deemed to be.

Type.

The type metric aims to give an overview of the cognitive load placed on the end-programmer. Types, either primitive or extended, are critical components of a programming language. From a cognitive standpoint, one could argue that with a greater number of unique types comes a greater level of difficultly to comprehend a specified platform. Namely, one would have to understand the implications behind the presence of many unique types. Additionally, the end-programmer would have to become very familiar with the various types and be able to apply them in the various contexts. The type metric also aims to inform how useful a particular type is across a specified class/library. For assessment purposes we deem a type to be useful if it has a higher degree of reusability across a class/library. Namely, if a type is reused quite often throughout a class/library its usefulness is established. Furthermore, the frequent use of a type places a lower cognitive load on the end-programmers

as they get used to dealing with that type. The following components are analyzed.

- *Unique Types*: Sum of unique types across a library.
- *Unique Types per Class*: Sum of unique types / total number of classes in a library.
- *Reusability*: Sum of the number of times each unique type is used across a library / total number of unique types across a library.

API Documentation Properties.

The manner in which a platform is documented is vital to its end-user. The documentation behind a platform indicates how simple or complicated a platform is to use. Stepping aside from purely quantitative results, we examine the qualitative aspects of the documentation for the .NET and Java platforms. Namely, we examine the navigation, the depth of the documentation involved, and the physical outlook of the documentations. The reason for choosing this as a metric lies once again on the cognitive load the end-programmer is faced with. The following components of the documentation per class are analyzed.

- *Method Signatures* – the number of clicks it takes to view the signatures of methods.
- *Overloaded Methods* – the number of clicks it takes to view the number of overloaded methods present.
- *Overloaded Method Signatures* – the number of clicks it takes to view the signatures of the overloaded methods and the number of times a method is overloaded.
- *Exceptions* – the number of clicks it takes to view the exceptions of a class in view mode.
- *Exceptions Subheading* – Examines whether exceptions are distinguished by a subheading.
- *Collapsible Subheadings* – Examines whether the class subheadings can be collapsed.
- *Method Overview* – Examines how detailed the documentation is for each method and whether examples are provided for them.
- *Primitive Types/Modifiers Identified* – Examines whether primitive types / modifiers are clearly distinguished.
- *Search Capability* – Examines whether the platform provides querying capabilities for methods and classes.

III. RESULTS

In this section we present the results from evaluating the two platforms using the metrics and libraries described in Section II.

Figure 1 shows the results from the first metric – Platform Overview. N and J in the left column of Figure 1 refer to the .NET and Java Platforms, respectively. L1 through L5 in the 2nd column refer to the five Libraries examined, as specified in the previous section – specifically, L1 refers to the IO libraries (System.IO of .NET and java.io of Java), L2 refers to the GUI libraries (System.Windows.Forms of .NET and javax.swing of Java), L3 refers to the Network libraries (System.Net of .NET and java.net of Java), L4 refers to the Regular Expressions libraries (System.Text.RegularExpressions of .NET and java.util.regex of Java), and L5 refers to the Reflection libraries (System.Reflection of .NET and java.lang.reflect of Java). The same notation is used in Figures 2, 3, and 4 as well.

The columns C, M, and M/C in Figure 1 stand for Classes, Methods, and Methods per Class, respectively.

P	L	C	M	M/C
N	L1	35	662	18.914
J	L1	66	519	7.864
N	L2	377	19963	52.952
J	L2	167	2602	15.581
N	L3	58	705	12.155
J	L3	38	389	10.237
N	L4	10	118	11.8
J	L4	3	30	10
N	L5	49	614	12.531
J	L5	10	89	8.9

Figure 1. Platform Overview.

From Figure 1 we readily observe that except the IO library in the Java platform, the .NET platform has more classes and methods than the Java platform. This suggests that the .NET platform provides more functionality to the end-programmer. However, we can also argue that the .NET platform is more complicated to understand and manage. The IO libraries for the Java and .NET platforms are quite comparable. The GUI library for the .NET platform is much more complicated than that of the Java platform. However, it provides more functionality and allows for effortless GUI programming. In the other three libraries we observe a similar result.

Figure 2 shows the results for the Entropy metric. The columns EqCl and Ent stand for Equivalency Classes and Entropy, respectively. These are quantities that are related to the entropy calculations as described in the previous section.

P	L	Nodes	EqCl	Ent
N	L1	59	8	1.48
J	L1	115	10	1.142
N	L2	435	16	1.21
J	L2	330	11	1.642
N	L3	79	7	1.447
J	L3	52	7	1.551
N	L4	26	4	.85
J	L4	8	3	1.406
N	L5	72	8	1.361
J	L5	16	5	1.717

Figure 2. Entropy.

From Figure 2 we readily observe that aside from the IO library in the .NET platform, the Java platform has a relatively higher measure of entropy. This seems to suggest that the Java platform's GUI, Network, Regular Expressions, and Reflection libraries are relatively more complex than the corresponding libraries of the .NET platform. The higher number of equivalency classes found in the .NET platform appears to contribute to this factor. Not surprisingly enough, we also take note that the .NET platform's core libraries have a greater number of nodes than that of the Java platform.

Figure 3 shows the results of the Ambiguity metric. Columns CA, MA, and NA stand for Class Ambiguity, Method Ambiguity, and Net Ambiguity, respectively.

P	L	CA	MA	NA
N	L1	2.2	2.68	.771
J	L1	.864	1.946	.636
N	L2	5.416	1.39	.554
J	L2	.677	.579	0.234
N	L3	1.793	1.05	0.483
J	L3	0.5	0.868	0.421
N	L4	0.7	0.471	0.1
J	L4	2.333	1.333	0.667
N	L5	1.714	0.906	0.408
J	L5	0.2	0.4	0.2

Figure 3. Ambiguity

In Figure 3 we see that aside from the Regular Expressions library, the .NET platform is far more ambiguous than the Java platform. This suggests that in the .NET platform the IO, GUI, Network, and Reflection libraries are far more complex than the corresponding libraries of the Java platform. Moreover, we see that the average number of times a method is overloaded is much higher in those libraries. Hence we could argue that the majority of the core .NET libraries are difficult to understand since the net ambiguity of the aforementioned libraries is closer to one. On the other hand one could argue that the .NET platform provides more flexible classes with respect to its numerous unique types.

Figure 4 shows the results of the Types metric. Columns UT, UT/C, and R stand for Unique Types, Unique Types per Class, and Reusability, respectively.

P	L	UT	UT/C	R
N	L1	41	16.182	0.123
J	L1	26	3.727	0.143
N	L2	288	18.211	0.056
J	L2	191	4.772	0.025
N	L3	55	9.649	0.141
J	L3	27	2.842	0.105
N	L4	16	8.857	0.473
J	L4	5	3	0.6
N	L5	35	7.896	0.206
J	L5	13	3.3	0.254

Figure 4. Types.

In Figure 4 we see that the .NET platform has nearly double the amount of unique types in each of the five libraries. Additionally we notice that the average unique types per class exhibit the same properties in the .NET platform. This suggests that the .NET platform requires the end-programmer to become familiar with more unique types. This places a greater cognitive load on the end-programmer and further makes the platform more complex to comprehend. However, another important component is the reusability factor of the unique types. We observe that the unique types in the IO, GUI, and Network libraries for the .NET platform are far more readily reusable than those of the Java platform. This on the other hand indicates that the .NET platform's unique types for the said libraries are more readily retained in the end-programmer's memory bank, hence being much easier to work with when compared against the Java types.

Figure 5 shows the results of the API Documentation metric. The entries in column DocView (Document View) stand for the following: MS stands for Method Signatures; OM stands for Overloaded Methods; OMS stands for Overloaded Method Signatures; E stands for Exceptions; ES stands for Exceptions Subheadings; CS stands for Collapsible Subheadings; MO stands for Method Overview; PT/MI stands for Primitive Types/Modifiers Identified; SC stands for Search Capability.

P	DocView	Result
N	MS	1 Click/method
J	MS	0 clicks/method
N	OM	1 Click/method
J	OM	0 clicks/method
N	OMS	1 Click/method
J	OMS	0 clicks/method
N	E	1 Click/method
J	E	1 Click /method
N	ES	Yes
J	ES	No
N	CS	Yes
J	CS	No
N	MO	Detailed
J	MO	Minimalistic
N	PT/MI	Yes (dark blue)
J	PT/MI	No
N	SC	Yes
J	SC	No

Figure 5. API

The results in Figure 5 demonstrate that some aspects of the Java platform's documentation excel against that of the .NET framework while it underachieves in other areas. Let us examine the navigation of the .NET platform's documentation in comparison to that of the Java platform. In the .NET platform, under class view, method signatures are not readily available. The end-programmer must click on the method itself to view its signature. In the Java platform all the methods and their respective signatures are visible in the class documentation view. Hence it is easier to view methods and their respective signatures in the Java platform as it requires no user clicks. Analogously, under the .NET platform the end-programmer would have to click the method of interest just to view its overloaded methods and its signatures; in the Java platform this is readily visible without the need for navigating to external pages. Viewing method exceptions in both the Java and .NET platforms require one click per method; no advantages are present here. However, the .NET platform clearly identifies Exceptions with a subheading in bold whilst the Java platform fails to do so; the subheading makes it easier on the end-programmer to quickly identify the exceptions for a given class/method. Analogously the .NET platform provides collapsible subheadings; this allows the end-programmer to narrow in on the sections of interest and makes it extremely easy to focus on the relevant information throughout the documentation. The Java platform's documentation does not support collapsible subheadings and hence makes it harder to navigate throughout the documentation. Another advantage of the .NET platform's documentation is that it provides a detailed overview of the methods with accompanying examples for the most part. This is a very appealing component of the documentation as it provides the end-programmer with detailed applications of the methods and clarifies any of the misconceptions present. As such, it makes the .NET platform much easier to follow and grasp when compared to the Java platform. Finally, the .NET platform's documentation excels in two other ways; namely, it identifies primitive types/identifiers using a dark blue font color and provides the ability to search for methods / classes within the documentation database. Color-coded primitive types/identifiers allow the end-programmer to readily distinguish essential components of the programming language and thus allows for easy interpretation of the textual base. Similarly, a convenient query field aids the end-programmers in the .NET platform to quickly address their needs and thus proving to be an effective feature lacking in the Java platform. All in all, the trade offs are numerous, the Java documentation has its own benefits and so does the .NET platform.

IV. CONCLUSION

We present a suite of metrics that can be used to evaluate the complexity of a software platform. We also use those metrics to evaluate two major platforms – .NET and Java. Our findings for each of our metrics are summarized as follows.

- *Platform Overview* – both platforms have their roughly equal share of strengths and weaknesses.
- *Entropy* – the .NET platform is less complex than the Java platform.
- *Ambiguity* – the Java platform is less ambiguous than the.NET platform.
- *Type* – the Java platform requires less cognitive effort than the .NET platform when working with types.
- *API Documentation Properties* – both platforms have their roughly equal share of strengths and weaknesses.

According to the above, the Java platform "wins" in two metrics, the .NET platform "wins" in one metric, and the two platforms "tie" in two metrics. Considering that the impact of each metric as compared to the other metrics is not evaluated, it would be fair to say that each of the two platforms has strengths and weaknesses without any of the two being clearly superior to the other.

It would be interesting, although difficult, to evaluate/measure the cognitive load when using certain components/areas of each platform. Also, and perhaps even more difficult, would be to develop methods to evaluate the importance of each metric as compared to the other metrics.

Last but not least, it is an important research direction to investigate the role and impact of the two platforms in terms of risk in the software development and maintenance process. As stated in [19], "*Software development involves plenty of risks, and errors exist in software modules represent a major kind of risk. Software defect prediction techniques and tools that identify software errors play a crucial role in software risk management*." We believe that it will be interesting and possibly very fruitful research to investigate how methods used in [19] can be applied in comparing complexities of entire platforms such as the ones we examine here.

REFERENCES

[1] J. Elshoff, "An Analysis of Some Commercial PL/I Programs", *IEEE Transactions on Software Engineering*, Vol. SE-2, June 1976, pp. 113-120.

[2] J. Elshoff, "Measuring Commercial PL/I Programs Using Halstead's Criteria", *ACM SIGPLAN Notices*, May 1976, pp. 38-46.

[3] T. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, Vol. SE-2, Dec. 1976, pp. 308-320 .

[4] S.H. Zweben, "A study of the physical structure of algorithms", *IEEE Transactions on Software Engineering,* Vol. SE-3, May 1977, pp. 250-258.

[5] J.L. Elshoff, "A study of the structural composition of PL/I programs", *ACM SIGPLAN Notices*, vol. 13, June 1978, pp.29-37.

[6] A. Fitzsimmons and T. Love, "A Review and Evaluation of Software Science," Computing Surveys, Vol. 10, No. 1, March. 1978, pp. 3-18.

[7] S.H. Zweben, and M.H. Halstead, "The Frequency Distribution of Operators in PL/I Programs", *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 2, March 1979, pp. 91-95.

[8] W. Harrison, K. Magel, R. Kluczny, and A. DeKock, "Applying Software Complexity Metrics to Program Maintenance", *IEEE Computer*, September 1982, pp. 65-79.

[9] R.E. Prather, "An Axiomatic Theory of Software Complexity Measure", *Computer Journal*, Vol. 27, No. 4, 1984, pp. 340-347.

[10] E.J. Weyuker, "Evaluating Software Complexity Measures", *IEEE Transactions on Software Engineering,* Vol. 14, No. 9, September 1988, pp. 1357-1365.

[11] J.S. Davis, and R.J. LeBlanc, "A Study of the Applicability of Complexity Measures", *IEEE Transactions on Software Engineering,* Vol. 14, No. 9, September 1988, pp. 1366-1372.

[12] W. Harrison, "An Entropy-Based Measure for Software Complexity", *IEEE Transactions on Software Engineering*, Vol. 18, No. 11, November 1992, pp. 1025-1029.

[13] J. Visser, "Structure Metrics for XML Schema", http://xata.fe.up.pt/2006/papers/51.pdf (last accessed March 9, 2010).

[14] H. Kim, and C. Boldyreff, "Developing Software Metrics Applicable to UML Models", 6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.101.3430 (last accessed March 9, 2010); also available at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.101.3430&rep=rep1&type=pdf (last accessed March 9, 2010).

[15] C. Marinescu, R. Marinescu, P.F. Mihancea, D. Ratiu, and R. Wettel, "iPlasma: An Integrated Platform for Quality Assessment of Object-Oriented Design", http://www.inf.usi.ch/phd/wettel/download.php?f=iPlasma-tooldemo.pdf. (last accessed March 9, 2010).

[16] .NET Platform, http://www.microsoft.com/NET/ (last accessed March 9, 2010).

[17] Java Platform, http://java.sun.com/javase/ (last accessed March 9, 2010).

[18] Java API. http://java.sun.com/j2se/1.5.0/docs/api/, (last accessed March 9, 2010).

[19] Y. Peng, G. Kou, G. Wang, H. Wang, F.I.S. Ko, "Empirical Evaluation Of Classifiers For Software Risk Management", *International Journal of Information Technology and Decision Making*, vol. 8, no 4, 2009, pp. 749-767.