Proceedings of the 7th International Conference on Applied Informatics
Eger, Hungary, January 28–31, 2007. Vol. 2. pp. 293–299.

Application of OO metrics to estimate .NET project software size*

Zoltán Porkoláb[a], Norbert Pataki[a], Ádám Sipos[a], Viktória Zsók[a], Marjan Heričko[b], Aleš Živkovič[b]

[a]Department of Programming Languages and Compilers
Faculty of Informatics, Eötvös Loránd University, Budapest
e-mail: {gsd, patakino, shp, zsv}@elte.hu

[b]Faculty of Electrical Engineering and Computer Science
University of Maribor
e-mail: {ales.zivkovic, marjan.hericko}@uni-mb.si

Abstract

One of the key questions in software development is software size estimation. For systematic software size estimation, different methods are used, all of which have their roots in the Function Point Analysis (FPA) method. However, the elements and constructs of the FPA method are not directly applicable to object-oriented concepts: a mapping of object-oriented concepts to FPA elements is required. There are proposals for such mappings, but a serious calibration and validation process is required to ensure that the various parameters have been chosen in the most appropriate way. Such a validation implies the creation of effective product metrics working in environments like the industrial standard .NET platform.

Since .NET is a typical multi-language environment, a product metric capturing all languages and producing comparable and accumulable results is hard and expensive to produce. Therefore we propose to solve the problem in the .NET Common Interface Language level: thus only one metric is capable to calibrate and validate the FPA mappings.

Keywords: Function point metrics, Complexity metrics, .NET

MSC: 68N30 Mathematical aspects of software engineering (metrics)

1. Introduction

Metrics play an important role in modern software engineering. Testing, bug-fixing cover an increasing percentage of the software life cycle. In software design

*Supported by Hungarian-Slovenian Bilateral S&T International Co-operation, National Office for Research and Technology, SLO-11/05.

the most significant part of the cost is spent on the maintenance of the product. The cost of software maintenance highly correlates with the structural complexity of the code. Based on software metrics we can give recommendations and define coding conventions on the development of sound, manageable and hygienic code.

The main problem with software complexity metrics is that they produce valuable data from the existing, already written code. With good complexity measurement tools the critical parts of the software can be identified in the early stages of the development process. However, such information is rarely useful to predict the future efforts to complete the whole system. As we build larger and larger systems, stringent tools are required to estimate the whole size of the project.

Software size is a relatively primitive measure, through which other important values are calculated (e.g. effort, costs, productivity and duration). These values are particularly interesting for project managers at the beginning of a project. In practice, an intuitive approach is often used, not a methodological one. The results are often unsatisfactory. The efficiency of the intuitive estimates can be indirectly measured via a software project's success factors, especially time and budget overruns. For systematic software size estimation, different methods are used [9, 13], all of which have their roots in the Function Point Analysis (FPA) method. Albrecht [1, 6] introduced this method in 1979. Since then, it ha become the most important method for software size estimation. The method introduced a specific way of representing a software system and distinguished between data functions and transactional functions. The method was intended for all domains, although in practice, its accuracy is different within different domains. A more detailed empirical analysis of the method revealed some weaknesses, which include [2, 3, 10, 11].

In the software development process *abstractions* play a central role. An abstraction focuses on the essence of a problem and excludes the special details. Abstractions depend on many factors: user requirements, technical environment, and the key design decisions. In software technology a *paradigm* represents the directives in creating abstractions. The paradigm is the principle by which a problem can be comprehended and decomposed into manageable components. In practice, a paradigm directs us in identifying the elements in which a problem will be decomposed and projected. The paradigm sets up the rules and properties, but also offers tools for developing applications. These methods and tools are not independent of their environment in which they occur.

The last 50 years of software design has seen several programming paradigms from *automated programming* and the FORTRAN language in the mid-fifties, to *procedural programming* with structured imperative languages (ALGOL, Pascal), to the *object-oriented* paradigm with languages like Smalltalk, C++ and Java. However, it is important to understand that new paradigms cannot entirely replace the previous ones, but rather form a new structural layer on the top of them. Object-orientation is a new form of expressing relations between data and functions, however, these relations implicitly existed in the procedural paradigm.

2. Mapping FPA to OO

Function Point Analysis methods were developed in the time where the main programming paradigm was structured programming. Albeit FPA was intended as independent from programming languages and design principles, its notation is inheritable differs from the current object-oriented ones. Therefore a very careful approach is necessary when it is applied for typical present-day projects.

The difficulties appear as a result of the different approach of abstractions in the view of FPA and in modern object-oriented environments. The abstraction of FPA separates the software system into two individual but co-operating parts: one part considers the data's influence and the other part counts the impact of the functionality. That is perfectly corresponds to the classical structural view of a software systems.

However, in modern object-oriented systems, there is two important considerations which contradict the FPA approach. First, a great part of data is hidden, as it is implemented as non-public part of classes. Those data have minor or no affect to the rest of the software system, but might seriously affect the implementation of the class. On the other hand, a strict separation of data and operations (functionality) does not stand for object-oriented systems. Here data complexity and operational complexity defined on data must be computed together.

Various approaches exist to fix these problems. In [16] Živkovič et al. supposed a mapping between FPA method and .NET systems – as an example of modern object-oriented systems. However, such a mapping requires a strong validation from the product metrics side. In ideally, the predictions of the .NET to FPA mapping has to be reflected by the values produced by a product metric working on the system after its completion. The schema of such a research can be described as it is in Figure 1.

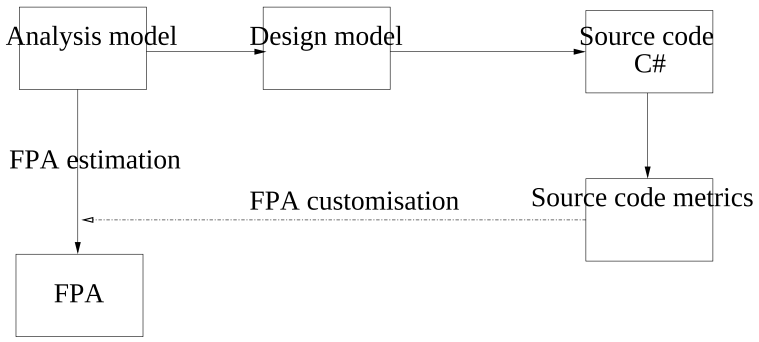

Figure 1: Original validation model of FPA mapping

Such a validating metric is not easily can be defined. In one side the direct use of object-oriented notations, like class, interface, method, inheritance, etc. must be avoided, since FPA methods are defined without object-oriented notions. In the same time, the metric should be applied for object-oriented systems developed in a special, multilanguage environment. It is typical, that in the .NET environment source code in C#, Visual Basic and other languages exist and working together to solve the problem.

3. Metric Validation

The .NET Framework is a development and execution environment that allows different programming languages and libraries seamlessly work together [4]. It has two main components: the Common Language Runtime and the .NET Framework class library. The Common Language Runtime is the foundation of the .NET Framework; this executes the program code and provides additional services for the development.

Programs written in different languages under the .NET development environment most cases are translated into Common Intermediate Language (IL). IL is a byte-code language standardized byby ECMA and ISO. IL stores the actual program code with a special instruction set and high level meta information about the program. The latter one is not necessary to run the program, but enables reflection and code analysis easier. The IL instruction set can be divided to the following major parts from the point of measuring:

- Base instructions: These are a set of basic operations, like loading and storing data on the stack, arithmetic operations, branch instructions, method calls etc. These instructions closely correspond to what would be found on a real CPU.

- Object model instructions: The object model instructions are less built-in than the base instructions in the sense that they could be built out of the base instructions and calls to the underlying operating system. These instructions are object instance creation, virtual method call, exception handling, memory allocation and type management.

With an appropriate utility, any .NET executable can be disassembled to IL (for example with ildasm) and can be decompiled to source level within moderate boundaries. Certainly, the decompiled code is only similar to the original source but the classes, methods and even the program flow structures are still recognizable. We can utilize this fact for the measurements.

Therefore it is possible to set up the measurement scenario described in Figure 2. Such a scenarion works not on the individual source files implemented in different languages but on IL assemblies. This way a more coherent and much more cost-effective validating environment can be established.

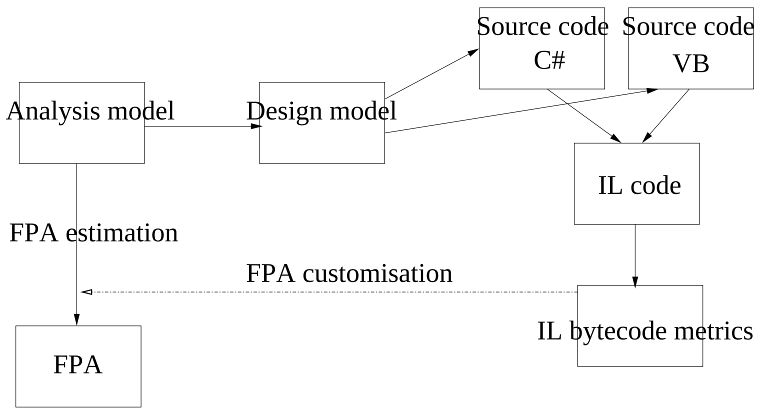

Figure 2: Validation model in a multi-language environment

4. Complexity Measurment on IL

Being rigorous is very important requirement on the measurement process: precision enables us to have interpretable absolute values that can trigger critical parts of an implementation and also enables us to have the future chance of comparison different program results. Now, let us examine the Intermediate Language as an input data for the measurement process. Clearly, it is key to have enough information encoded in the .NET executable to do precise measure offs. IL stands half way between the native code and the source code. We can prove that the IL is a better choice for measurements than the other two formats.

Native code is specific for measuring. The encoded information here is only for execution on a machine environment, so all irrelevant data are detached from the executables. Furthermore, native code relies on particular machine architecture, which leads to the similar problem with the different languages. To sum up, there is no way to have precise measurements on native executables.

Going along the opposite extreme, source code is many cases too complex; it requires a lot of effort to extract the relatively simple measurement information. Dealing with source code text requires literally a half compiler (lexical- and syntactical analysis plus some semantic analysis too) for each different programming language to build up the program structures. This would change provided the primary data format of program sources change from pure text representation to real structures; but until then we have to face with this problem.

5. Conclusion

Function Point Analysis (FPA) methods are frequently used for software size estimations. Since FPA is not defined in terms of object-orientation certain mapping procedure is required when FPA is used in most modern software environment, such as the .NET platform. For the correct parameterization the design of such a mapping is required a validating and calibrating product metric to check the accuracy of the mapping.

In this paper we propose to define such a validating metric not on the source language level as ordinary metrics do, but to utilize the IL as a common platform for .NET complexity measurement. This enables to avoid the excessive work dealing with different languages and in the same time it increases consistency between the measurements. Handling inter-language connections are also possible. We presented a way to implement such tool by evaluating the relationship between the IL encoded information and the required information to calculate some popular metrics.

In the future we continue the empirical evaluation of our .NET metrics tool. We plan to examine the results of measuring large-scale .NET projects and develop the best parameterization for FPA metrics based on the empirical results.

References

[1] Albrecht, A., Measuring Application Development Productivity, *IBM Applications Development Symposium*, (1979), 83-92.

[2] Antoniol, G., Lokan, C., Caldiera, G., Fiutem, R., A Function Point-Like Measure for Object-Oriented Software, *Empirical Software Engineering*, 4 (1999), 263–287.

[3] Antoniol, G., Fiutem, R., Lokan, C., Object-oriented function points: An empirical validation, *Empirical Software Engineering*, 8 (2003), 225–254.

[4] Standard ECMA-335 Common Language Infrastructure (CLI) 4th edition June 2006, `http://www.ecma-international.org`

[5] Heričko, M., Rozman, I., Živkovič, A., A Formal Representation of Functional Size Measurement Methods, *The Journal of System and Software*, 79 (2006), 1341–1358.

[6] IFPUG, Function Point Counting Practices Manual, Release 4.2, *International Function Point Users Group*, Princeton Junction, USA, (January 2004).

[7] ISBSG, Practical Project Estimation, A toolkit for estimating software development effort and duration. *International Software Benchmarking Standards Group*, (2001).

[8] ISO/IEC TR 14143-1. Information technology – Software measurement – Functional size measurement, Part 1: Definition of concepts, First edition, *ISO/IEC*, (1998).

[9] Jeffery, D. R., Low, G. C., Barnes, M., A Comparison of Function Point Counting Techniques, *IEEE Transactions on Software Engineering*, 19 (1993), 529–532.

[10] Lokan, C. J., An empirical study of the correlations between function point elements, *Proceedings of METRICS '99*: Sixth International Symposium on Software Metrics, (1999), 200–206.

[11] Lokan, C. J., An empirical analysis of function point adjustment factors, *Information and Software Technology*, 9 (2000), 649–659.

[12] Uemura, T., Kusumoto, S., Inoue, K., Function-point analysis using design specifications based on the Unified Modelling Language, *Journal of Software Maintenance and Evolution-Research and Practice*, 13 (2001), 223–243.

[13] Živkovič, A., Hericko, M., Kralj, T., Empirical assessment of methods for software size estimation. *Informatica (Slovenia)*, 4 (2003), 425–432.

[14] Živkovič, A., Heričko, M., Brumen, B., Beloglavec, S., Rozman, I., The Impact of Details in the Class Diagram on Software Size Estimation, *Informatica (Lithuania)*, 16 (2), (2005), 195–211.

[15] Živkovič, A., Rozman, I., Heričko, M., Automated Software Size Estimation based on Function Points using UML Models, *Information & Software Technology*, Vol. 47, Issue 13, (October 2005), 881–890.

[16] Živkovič, A., Heričko, M., Goljat, U., Porkoláb, Z., Improving Size Estimetes with .NET Product Metrics, *ERK 2006*, IEEE 15th International Electrotechnical and Computer Science Conference, September 2006, Portorož.

Zoltán Porkoláb, Norbert Pataki, Ádám Sipos, Viktória Zsók
Eötvös Loránd University, Faculty of Informatics
Dept. of Programming Languages and Compilers,
Pázmány Péter sétány 1/c., H-1117 Budapest,
Hungary

Marjan Heričko, Aleš Živkovič
University of Maribor
Faculty of Electrical Engineering and Computer Science
Smetanova 17, SI-2000 Maribor,
Slovenia