Early metrics for Object Oriented Designs

Boris Baldassari, Chantal Robach
LCIS, INP Grenoble
50, rue Barthélémy de Laffemas
26000 Valence, France
boris.baldassari, chantal.robach@esisar.inpg.fr

Lydie du Bousquet
LSR, ENSIMAG
681, rue de la passerelle
38402 Saint Martin d'Hres — France
lydie.du-bousquet@imag.fr

Josiane Brosse
Thalès-Avionics
25 rue Jules Védrines
26000 Valence — France
josiane.brosse@thales-avionics.com

Abstract

To produce high quality object-oriented systems, a strong emphasis on the development process is necessary. This implies two implicit and complementary goals. First, to ensure a full control over the whole process, enabling accurate costs and delays estimation, resources efficient management, and a better overall understanding. Second, to improve quality all along the system lifecycle at development and maintenance stage. On the client side, a steady control over the development process implies a better detection and elimination of faults, raising the fiability and usability of the system.

*This paper introduces a realistic example of metrics integration in the development process of object-oriented software. By addressing early stages of the design (*ie. *class diagram), we can anticipate design-level errors or warnings thus enabling a reduction of immediate and further costs and problems. Metrics used are issued from state of the art object-oriented research, and integrated in the widespread unified process of software development.*

1. Introduction

Although some advances have been made in software engineering over last years [2, 8, 23], large software projects are still difficult to manage, often involving supplementary costs and delays. As developed in [18, 21], now that we are able to build important software edifices[1], we still have to learn how to build them well, in order to achieve better productivity and better quality.

Both these goals heavily depend on the development process, and the control developers have on it. More specifically, design and testing issues are the most cost and time-effective: while modifying a bad implementation is still allowed at the testing stage, design problems may be prohibitly cost- and delays-expensive, and may remain until the maintenance stage. These issues are even more critical in object-oriented paradigm, since this approach has been introduced years later.

In this paper, we present a method to anticipate such problems using object-oriented metrics at the design step. The originality of the approach is to propose testability and design targeted metrics application at steps where information is scarce (class diagrams), and to make them consistent altogether. By addressing early stages of the development process, we are able to ensure quality and testability at the design level.

First, we will briefly present in section 2 context basis with a short presentation of software quality engineering and object-oriented paradigm. In section 3, we examine the unified process of software development, and state where improvement attempts would be efficient. Section 4 presents metrics defined for object-oriented designs, and the theoretical and pratical aspects of the measures.

We propose a reduced set of metrics, an adaptation of them and an example of their integration in the unified development process in section 5.

Finally, we conclude in section 6.

[1] Parallels with the architecture science are common and relevant – see [21]

2. Context basis

2.1. Elements of Software Quality

The software quality engineering notion covers many aspects, all along the software lifecycle. On the developer's side, quality is the way to costs and delays accurate estimation, easier testing, better maintenability. One may argue that quality should be quantifiable prior to improvement [25]. Indeed, software quality notions are often associated with software metrics, and therefore dependent upon it.

2.2. The Object-Oriented Model

Object-Oriented approach has been introduced 35 years ago, with languages such as Simula (1967) and SmallTalk (1980) [18]. Since, engineers and researchers have been able to design and build complex systems, and gain experience and maturity. The maturity of the concept, however, is not currently achieved and, as stated in [21] and [18], will not be until we gain enough experience. However, such an area comes lately with the introduction of reflexion groups and increase of specialized knowledge in software industries.

Although there are common aspects between classic and object-oriented paradigms, authors have underlined differences from many points of view: development process [4, 8], metrics [6, 23], testing [2] or design [10]. Amongst others, the introduction of new features (data abstraction, encapsulation, inheritance, polymorphism, self-recursion, depending on languages) induces new bug hazards, new fault model, and moreover new design, analysis, test and metrics methodolgies.

Object-oriented paradigm focuses on architecture rather than on a sequential execution, thus raising the importance of the former. [5, 14] and [6], amongst others, address such issues and propose metrics targeted at architecture.

3. Object-Oriented sofware development

Developing the Object-oriented maturity implies the definition of a formalism for system modeling, data exchanges and process control. This is accomplished by the Unified Modeling Language (UML) [11, 13], which furnishes a way to visually represent, formalize and exchange system architectures and artefacts. Although some ambiguity have been found in definitions, UML is widely adopted by the object-oriented community and thus considered as a *de facto* standard.

There are many ways to build an object-oriented system. The software development process has been greatly improved over the years, providing developpers with guidelines and references: OOSE [12], Booch [3], OMT [20], Use-Case writing [7, 12].

3.1. Presentation of the Unified Process

The Unified Process of software development [11] is a capture of best practices [8], taking advantages from three of the main used development processes (OOSE [12], OMT [20], Booch [3]). This paternity gives it a long-breath experience and maturity. Technically, it is an iterative and incremental, use-case driven, architecture-centric development process, targeted at object-oriented systems [4, 8]. Each iteration is broken into four parts, as shown in figure 1.

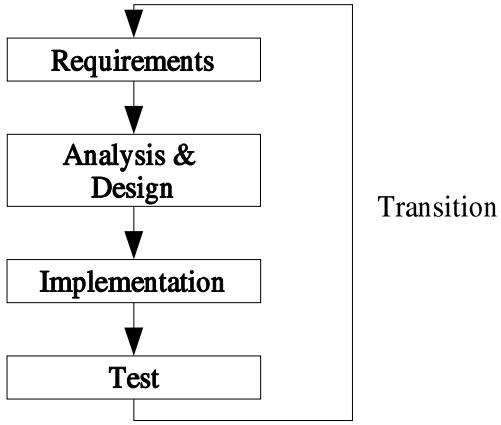

Figure 1. Steps of the Unified Process

- The **requirements** step describes *what* the system should do and allows developers and customers to agree on this description. To achieve this, functionnalities and constraints are elicited, organized and documented; actors and use cases, as well, are identified and commented.
- The **analysis & design** step deals with *how* the system will be implemented, given the consistent view obtained at the previous step. This results in an *analysis model*, which represents an abstraction of the software without any technical constraint, and a *design model*, which acts as a 'blueprint' of how the source code is structured and written.
- The **implementation** phase uses the defined design model to implement classes and objects in terms of components. Unit testing is also accomplished during this step.
- The purpose of the **testing** stage is to verify interactions between objects, proper integration of all compo-

nants, correct implementation of all requirements, and defects detection.

This development process has been widely adopted by Object-Oriented developers, and the majority of software development platform proposals include a development process based on the Unified one [9, 8]. Many, if not all, large object-oriented software projets use it as a guideline.

Our solution, for the two reasons given above (*ie.* maturity and industrial-wide use), has been dedicated to the unified process.

3.2. Early stages attempts

The unified process, by the formalism it introduces, makes clear distinctions between the different phases (*eg.* the design model is mandatory before the implementation phase). Formalism is very important for software engineering, since it reduces human errors and clarify the project's advance.

Any improvement attempt has to take place as soon as possible [19]: if modifications are needed at the implementation stage, changes costs may be prohibitively high. Besides, some changes involving structure alterations just can't be made without a complete overhaul. On the other hand, some minimal information is required to appraise software quality, and the more accurate these informations are, the more accurate metrics are too.

A compromise has to be found between earliness and completeness; the majority of metrics suites actually used address the later, often needing at least some code. Although this is undoubtedly a good approach, we specifically want to address analysis and design issues sooner in the process, thus reducing costs and delays.

4. Software quality metrics

4.1. Background

Considering a large scope, software engineering and quality issues are as old as 40 years; classical paradigms have carefully been studied, and many metrics have been proposed.

Software metrics address several aspects of quality: from algorithmic complexity to fault-proneness and testability (*ie.* the facility to conduct tests on the software).

Amongst the classics, McCabe [15] proposes an algorithmic complexity measure based on an graph analysis of the program structure. Although this has become a reference for further studies, limits of the concept have been early reached and critized [22]. Indeed, this measure is confined to pure algorithmic complexity, which make it usefull for weighting of some metrics — *eg.* for WMC in Chidamber and Kemerer metrics suite.

Voas has proposed in [24] a metric targeted specifically at testability: the PIE (Propagation, Infection, Execution) approach. This metric quantifies the probability of a fault to appear, to be executed and to be seen. The interesting point here is the focus on faults and their consequences.

Other measures from the classical (*ie.* sequential) paradigm are based on similar principles; most of the metrics targeted at classical programs do not address architecture aspects, which are essential in object-oriented paradigm – and the one to be available at a early stage.

Amongst object-oriented software metrics, many address the implementation stage, with measure items such as the number of lines in the code, the number of attributes [5, 6] or the complexity of methods [15, 17]. Although these metrics are recognized as relevant and actualy correlated to fault-proneness, they require an advanced state of development — *ie.* the implementation stage.

4.2. Class level information

One of the first suites of Object-Oriented design measures was proposed by Chidamber and Kemerer [6]. This work, widely considered as the foundation of Object-Oriented metrics, proposes a strong mathematical base — which was an aspect often criticized in other proposed suites. These metrics are:

1. *Weighted Methods per Class* (WMC): the weighted sum of all methods in a class.
2. *Depth of Inheritance Tree* (DIT): maximum length from the class to the root in the inheritance tree.
3. *Number of Children* (NOC): number of directly inherited classes.
4. *Coupling Between Object classes* (CBO): count of the number of other classes coupled to the considered class.
5. *Response For a Class* (RFC): number of methods that can potentially be invoked by a message received by an object of the considered class.
6. *Lack of Cohesion in Methods* (LCOM): number of methods using the same set of attributes minus the number of methods using a different set of attributes.

Subramanyam [23], ten years later, discusses the relevance of these metrics confronted to the industrial and research feedback. A majority of them is considered to be actually correlated to fault-prononess; although the size factor, which is under-estimated in Chidamber and Kemerer work, appears to have a great impact. Applied on large systems

(*ie.* more than 500.000 lines of code), only coupling and inheritance metrics are still relevant.

From these observations, Subramanyam proposes a reduced set of metrics consisting of WMC (*Weighted Methods per Classes*), CBO (*Coupling Between Objects*), DIT (*Depth of Inheritance Tree*) and a size measure, not included in Chidamber proposal.

4.3. Package level information

R. Martin proposes in [14] a set of metrics to measure the quality of an object-oriented design in terms of interdependences between subsystems. From an UML perspective, the author examines relations between packages, and their implication on the design quality.

A highly interdependent design (*ie.* with several relations between packages) tends to be rigid, and therefore difficult to reuse, modify and maintain. On the other hand, packages have to collaborate in order to deliver services — otherwise, they're useless. So there are good dependencies, and bad dependencies, given their quantities, the stability of packages involved, etc.

R. Martin proposes four metrics:

- C_a (Afferent Coupling) is the number of classes outside the category (*ie.* package) that depend upon classes within the category. It represents the use of the considered category by others.
- C_e (Efferent Coupling) is the number of classes inside the category that depend upon classes outside the category. The C_e metric represents the dependency of the considered category upon the outside.
- I (Instability) is calculated with the C_a and C_e metrics, and stands in the [0,1] range. $I = 0$ indicates a maximally stable category, and $I = 1$ indicates a maximally instable category.

$$I = \frac{C_e}{C_a + C_a}$$

- Stable categories should be highly abstract so that it can be extended; the A (Abstractness) metrics, ranged in [0,1], provides such a measure.

$$A = \frac{\#\ of\ abstract\ classes\ in\ the\ category}{total\ \#\ of\ classes\ in\ category}$$

Categories with a high C_a are not to be changed, since this will result in a cascade of failed dependencies. Categories with a high C_e will be harder to test, since we will have to put in place all dependencies before.

However, having a very stable or instable category is not always good or bad: this depends on the nature of the category, its role and the fonctionnalities it offers. A low-level category will be intensively used by others, and there's nothing bad with it; all we have to ensure is the adequation between the role and the fact. To fulfill this goal, R. Martin provides a gabarit, whose abstractness is 'balanced' with stability.

This suite of metrics has the merit to address package-related issues, since most, if not all, of other metrics rather address class or algorithmic issues.

4.4. Other studies

There are a few other studies providing object-oriented metrics; amongst these, we can cite Abreu [5], Subramanyam [23], Morris [16]. Although these are interesting approaches, the concepts remain identical to earlier studies, with some algorithmic complexity and class-level analysis. Given the little amount of information we own at conception stage, they provide no supplementary element.

5. Metrics selection and integration

5.1. Selection of metrics

Following [1] and [19], we focus on Analysis and Design stage, in order to adress early architectural issues. There is a strong requirement to fulfil: given the early stage of the measure, we do have a fairly low amount of information. Typically, we are restricted to general structure informations, such as those provided on a simple and concise class diagram. We consider only the following minimal requirements:

- Classes and packages are defined, but neither methods definition nor any attribute is needed. Thus, metrics involving number of lines of code, or algorithmic complexity are not available.
- Relations (*ie.* associations, compositions, aggregations and inheritance) are known, but neither stereotypes, nor adornments (*ie.* cardinality, orientation, etc.) are needed.

Amongst the presented metrics, we have selected three class-related and four package-related metrics. Considering information is scarce, we have to exploit it to the maximum, in order to address all levels of available information (class and package).

At the class level, these metrics are the *CBO*, the *DIT* and the *NOC* metrics. They all are considered to be size-independent and confirmed by industrial experience [23], and available at an early stage of design.

At the package level, selected metrics are the C_a, the C_e and the I metrics. We also add a size metrics given by

the number of classes in a package. Industrial experience, gained through several object-oriented projects at Thalès Avionics, has proven the relevance of these metrics in the way provided by [14] and [23].

5.2. Implementation

We have used, for metrics calculation, an existing software named UML Checker, developed by Thalès Avionics for internal purposes.

UML Checker takes as an input a Rhapsody™ or a Rose™ project file, and compute several metrics from [5, 6, 14]. The output is a classic XML file, which we transform into a HTML file for the user's convenience.

Presentation of resulting metrics is of central importance, since we have to make it relevant to the developer in an efficient manner. Amongst metrics provided by standard development environments, many cannot be exploited since they do not provide a consistent and useable view. We favoured a better navigability though results using hyperlinks, thus enabling a better overall view of the architecture and its measures.

5.3. Results and validation

This metric suite has been applied on a industrial object-oriented project at Thalès Avionics, developed with Ada95. This project is medium-sized with 358 classes and 55 packages, and constitutes a typical object-oriented development project.

Using this suite of metrics, we have been able to predict and identify, with the development team of the project:

- Heavily inherited elements, *ie.* with a high DIT value for the considered class and several high NOC amongst its ancestors. Such large and high inheritance trees become dangerous when they are associated with an important complexity (CBO).
- Connectivity-related problems, such as over-associated classes and packages, which lead to design misunderstandings and testing problems. Connectivity issues are revealed by high values of CBO, C_e and/or C_a. For classes, CBO is closely connected to inheritance metrics; at package level, C_e is a direct measure of testing difficulty.
- Nature of elements, and semantic errors. For example, classes with a high number of children are base-objects (*eg.* the Object object in Java). Such classes are heavily instanciated and derived, but there's nothing bad about it, as long as these, and only these, are known and identified. If a direct correspondance between semantic (*ie.* what classes are supposed to do) and facts (*ie.* what metrics reveal) cannot be established, the involved parts of the design are faulty.

6. Conclusion

Our goal in this paper was to introduce, rather than a specific measure usage, a realistic exemple of metrics understanding and integration in an industrial context. By selecting only relevant information, we keep a steady control on the system and its development process while complexity grows.

Selected metrics and statistics usage provide us with an overall view of the whole system, thus enabling us to focus particularly on problematic parts. We focus on architectural and design issues, because of their importance later in the development process, and the costs and delays they can generate if they are faulty. Quality attributes considered are mainly fault-proneness and complexity.

Software quality engineering requires, indeed, such theoretical and pragmatic work, but also a wide quality improvement process in software companies, and the development of fault models adapted to each industrial context, which has yet to be integrated in metrics suites in an efficient manner.

References

[1] B. Baudry, Y. Le Traon, and G. Suny. Testability analysis of a uml class diagram. In *8th International Software Metrics Symposium (Metrics 2002)*, IRISA, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France, jun 2002.

[2] R. V. Binder. *Testing Object-Oriented Systems – Models, Patterns and Tools*. The Addison-Wesley object technology series. Addison-Wesley, Oct. 1999.

[3] G. Booch. *Object-Oriented Design with Applications*. Redwood City, CA: Benjamin/Cummings, 1991.

[4] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, Feb 1999.

[5] F. Brito e Abreu. Toward the Design Quality Evaluation of Object-Oriented Software Systems. In *Proceedings of the 5th International Conference on Software Quality*, Austin, Texas, USA, Oct 1995.

[6] S. Chidamber and C. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[7] A. Cockburn. *Writing Effective Use Case*. Addison-Wesley Longman Publishing Co., Inc., Oct 2000.

[8] R. S. Corporation. Rational Unified Process: best practices for software development teams, 2001. http://www.rational.com/media/whitepapers/.

[9] D. H. Eran Gery and E. Palachi. Rhapsody: A Complete Life-Cycle Model-Based Development System. Technical report, I-Logix, 2001.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[11] O. M. Group. Unified modeling language. Specification v1.3, Object Management Group, June 1999. http://www.omg.org/technology/uml/.

[12] I. Jacobson, M. Christerson, P. Jonsson, and G. ´Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison–Wesley, 1992.

[13] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process – Second edition*. Prentice Hall, Jul 2001.

[14] R. Martin. OO Design Quality Metrics : an Analysis of Dependencies. Intranet Thales-avionics, 1994.

[15] T. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976.

[16] K. Morris. Metrics for Object-Oriented Software Development Environments. Master's thesis, M.I.T. Sloan School of Management, 1989.

[17] B. A. Nejmeh. Npath: a measure of execution path complexity and its applications. *Communications of the ACM*, 31(2):188–200, Feb 1988.

[18] L. B. S. Raccoon. Fifty years of progress in software engineering. *ACM SIGSOFT Software Engineering Notes*, 22(1):88–104, 1997.

[19] D. Richardson and A. L. Wolf. Software Testing at the Architectural Level. In *Proc. of the Second Inter. Software Architecture Workskip*, pages 68–71, San Fransisco, California, USA, October 1996. ACM.

[20] J. R. Rumbaugh, M. R. Blaha, W. Lorensen, F. Eddy, and W. Premerlani. *Object-Oriented Modeling and Design*. Prentice-Hall, Oct 1991.

[21] M. Shaw. Prospects for an Engineering Discipline of Software. *IEEE Software*, 7(6):15–24, Nov. 1990.

[22] M. Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2):30–36, Mar 1988.

[23] R. Subramanyam and M. Krishnan. Empirical Analysis of CK metrics for Object-Oriented Design Complexity : Implications for Software Defects. *IEEE Transactions on Software Engineering*, 29(4):297–310, Apr 2003.

[24] J. M. Voas. PIE: A Dynamic Failure-Based Technique. *IEEE Transactions on Software Engineering*, 18(8):717–727, Aug 1992.

[25] E. Weyuker. Evaluating Software Complexity Measures. *IEEE Transactions on Software Engineering*, 14:1357–1365, 1988.