Evolution Metrics

Tom Mens*
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels, Belgium
tom.mens@vub.ac.be

Serge Demeyer
Lab on Re-Engineering
Universiteit Antwerpen
Universiteitsplein 1, 2610 Wilrijk, Belgium
serge.demeyer@uia.ua.ac.be

ABSTRACT

Since the famous statement "What is not measurable make measurable" of Galileo Galilei (1564 – 1642) it has been a major goal in science to quantify observations as a way to understand and control the underlying causes. With the growing awareness that evolution is a key aspect of software, an increasing number of computer scientists is investigating how metrics can be applied to evolving software artifacts. This paper provides a classification of the various approaches that use metrics to understand and control the software evolution process, gives concrete examples for each of the approaches, and identifies topics that require further research. As such, we expect that this paper will stimulate this emerging research area.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.8 [**Software Engineering**]: Metrics; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

Keywords

software evolution, evolution metrics

1. INTRODUCTION

Improving software quality, performance and productivity is a key objective for any organisation that develops software. Quantitative measurements – and software metrics in particular – can help with this, since they provide a formal means to estimate software quality and complexity. The aim of this paper is to explore how and where software metrics can be used during the software evolution process.

Better tool support for evolution is essential, since numerous scientific studies of large-scale software systems have shown that the bulk of the total software-development cost is devoted to software maintenance [6, 15, 26]. This is mainly due to the fact that software systems need to evolve continuously to cope with the ever-changing software requirements.

Metrics have a number of interesting characteristics for providing evolution support: they are simple, precise, general and scalable to large-size software systems. In this paper we provide an overview of the kinds of evolution support that can be provided with software metrics. Therefore, we make a distinction between the use of software metrics *before* the evolution has occurred (i.e., *predictive*), and *after* the evolution has occurred (i.e., *retrospective*). This terminology is adopted from Gall et al. [14].

2. PREDICTIVE ANALYSIS

Before evolution, software metrics can be used to analyse the software, with the aim (a) to assess which parts *need* to be evolved (evolution-critical), (b) which parts are *likely* to be evolved (evolution-prone), and (c) which parts can *suffer* from evolution (evolution-sensitive). For each of these cases, some existing research is identified, but further investigation and experimental validation of the results remains crucial.

2.1 Evolution-critical parts

Evolution-critical parts are parts of the software that *need* to be evolved due to a lack of quality (i.e., software parts that violate good design principles, are badly structured, contain errors, are incomplete, etc). Evolving these parts of the software is necessary to improve the software quality and structure, or to reverse the effects of software aging. In these situations, *refactoring* [30] is often appropriate.

Metrics have long been studied as a way **to assess the quality of large software systems** [13] and have been applied to object-oriented systems as well [5, 9, 24, 27, 28, 29]. However, a simple measurement is not sufficient to assess such a complex thing as software quality [16], not to mention the reliability of the results [8].

Simon et al. [34] have proposed to use metrics **to identify which refactorings should be applied and where**. The approach is demonstrated for four typical refactorings: *move method*, *move attribute*, *extract class* and *inline class*. They make use of a generic distance-based cohesion metric $dist(x, y) = 1 - \frac{|p(x) \cap p(y)|}{|p(x) \cup p(y)|}$ where $p(x)$ is some property of x, such as the set of superclasses of x, the set of attributes of x, the set of abstract methods of x, etc... Using this metric, parts of the software can be visually clustered to easily identify anomalies that suggest a particular kind of refactoring.

Metrics have also been used **to recognise duplicated code** which can later be removed by applying the appropriate refactorings [21, 22]. Code duplication is a typical phenomenon when systems evolve, hence the relevance of the topic.

In a recent experiment we have used metrics **to detect incom-**

*Tom Mens is a postdoctoral fellow of the Fund for Scientific Research - Flanders (Belgium).

plete code. More specifically, we identified *abstract leaf classes* in an object-oriented framework. This is an indication of bad design, since one of the object-oriented design heuristics proposed by Riel [33] is that *all abstract classes must be base classes*. Stated otherwise, a leaf class should never contain any abstract methods. The reason for this is that when abstract leaf classes are instantiated they can give rise to a run-time error when one of their abstract methods is invoked. If $NMUA(c)$ is the metric that calculates the number of abstract methods that are understood by a class c (i.e., defined in the class or one of its ancestors), and the metric $NDC(c)$ calculates the number of direct children of c, then c is an abstract leaf class if $NDC(c) = 0$ and $NMUA(c) > 0$. In one application we tested, we found that 8 of the 32 leaf classes still contained an unimplemented abstract method. The original developer of the application agreed that this was a conceptual error, and it was removed in the next version of the application.

Metrics can also be used **to detect obsolete code**, i.e., code that is not used anymore. Situations like this are likely candidates to be removed in future versions of the software, to make the code more understandable and maintainable. Note that it is not always easy to find out whether or not a certain part of the software is still being used. This problem is even worse in object-oriented software due to dynamic notions such as late binding of self references. Therefore, so-called *dynamic metrics* that make use of run-time information extracted during different executions of the software could be helpful.

2.2 Evolution-prone parts

Evolution-prone parts are parts of the software that *are likely* to be evolved (even if nothing is wrong with their structure or quality). This can be because these unstable parts correspond to highly volatile software requirements.

One possible way to detect this is by investigating release histories of the software, and identifying which parts of the software have most frequently changed in the past. To detect these situations, one needs a version management system to gain access to previous versions of the software [14].

Metrics can be used here, for example, to give a measure of the number of times a change has been made to a certain class (or module, component, ...). Each change could be weigthed based on the timestamp of the version to which it was applied. The more changes that have been made in the (recent) past, the likelier it is that this part of the software is still unstable and likely to evolve.

To cope with the scalability issue, typical examples of this approach *visualise* the measurements. For example, Ball and Eick annotate code views with colours showing code age [1], and Jazayeri et al. use a three-dimensional visual representation for examining a systems software release history [18]. Lanza [23] combines software visualisation and software metrics as a simple and effective way to recover the evolution of object-oriented software systems.

2.3 Evolution-sensitive parts

Evolution-sensitive parts are parts of the software that can cause problems upon evolution, or where the estimated effort of managing the impact of changes is very high. This is typically the case for those parts of the software that are tightly interwoven. Whenever something is changed in these parts, it may have a high impact on many other parts.

Obviously, highly coupled parts of the software are very sensitive to changes because they typically consist of software entities that are strongly connected with one another. Therefore, *coupling metrics* (such as CBO: coupling between object classes [5]; CF: coupling factor [4]; RFC: response set for a class [5]) can be used to detect these situations and to give a measure of evolution-sensitive parts.

Kabaili et al. [19] recently investigated whether *cohesion metrics* could also be used as changeability indicators, and concluded that this is not the case, at least not for the common cohesion metrics LCC (loose class cohesion [2]) and LCOM (lack of cohesion in methods [5, 25, 17]). Therefore, as long as more suitable cohesion metrics are not defined, they cannot be trusted as changeability indicators.

A concrete suggestion for a cohesion metric that might assess the evolution-sensitivity of an object-oriented inheritance hierarchy would be a base class (i.e., a non-leaf class) that performs many self sends, while its subclasses perform many super sends. Due to the intricate interaction between self and super sends, replacing the common base class by a new version is likely to lead to problems in the subclasses that are not always easy to detect (cf. fragile base class problem [35]). To detect this situation, we can use and combine metrics that count the number of self sends and super sends in a class.

More research remains necessary to find out whether other metrics than cohesion and coupling can be used to detect evolution-sensitive parts of the software.

3. RETROSPECTIVE ANALYSIS

After evolution one can analyse previous releases of the software, e.g., to find out whether its structure or quality has improved. Alternatively, one can study the evolution process, e.g., to understand what has been changed and how, or to detect those places where the most substantial or intrusive changes appear. In both cases, software metrics can be used to a certain extent.

3.1 Analysing the software

By comparing the current version of the software with the previous version(s) one can try to assess whether the goals of evolution have been achieved. For example, if the evolution was a restructuring, we should assess whether the new version is better structured than the old one. We could also use metrics to detect the kind of evolution that took place.

Gall et al. [14] used coupling metrics based on a retrospective empirical analysis of multiple releases of a large telecommunication switching system. These results were used to estimate more accurately further maintenance activities.

Demeyer et al. [8] evaluated a number of existing size and inheritance metrics on three releases of a medium-sized object-oriented framework (VisualWorks / Smalltalk). From the framework documentation one can deduce that the transition from the first release (1.0) to the second release (2.0) was mostly restructuring, while the transition from the second (2.0) to the third release (2.5) was mainly extension. This restructuring and extension was confirmed by the measurements. During the *restructuring* phase, a substantial number of classes changed their hierarchy nesting level (i.e. the number of superclasses) and the number of methods defined. This implies that most of the changes were in the middle of a class hierarchy which is indeed typical for a major restructuring. Yet, during the *extension* phase none of the classes changed their hierarchy nesting level, but a significant amount increased or decreased the number of children. Thus, all changes were made to the leaves of the inheritance hierarchy which is indeed typical for extensions. Consequently, the $1.0 \rightarrow 2.0$ restructuring did improve the inheritance structure since the subsequent $2.0 \rightarrow 2.5$ transition really exploited the inheritance hierarchy.

In a recent experiment, we used metrics to compare two versions of SOUL [36], a logic language that is implemented on top of Visu-

alWorks / Smalltalk. The evolution from release 2.2 to release 2.3 was mainly an extension, since 6 new classes were introduced, and only 1 class was removed. Apart from this, many existing classes were extended, which could be detected by an increase in methods from the old release to the new one. While most of these extensions involved only one or two methods, there were 4 classes that introduced more than 10 new methods. Finally, one class was refactored, which could be detected by a change in its hierarchy nesting level combined with a decrease of its number of defined methods. All this information can provide significant help to identify candidate classes to look at when trying to get a better understanding of the evolution that took place.

While the above experiments indicate that metrics can be used to detect the kind of evolution that took place, more work is needed to find out which metrics are most appropriate for this purpose.

3.2 Analysing the process

By analysing which changes have been applied to obtain a new version of the software, it is possible to reconstruct the evolution process (i.e., the *where* and *how*), and from there deduce the underlying design rationale (i.e., the *why*). Metrics are quite applicable here because there is so much data to analyse.

As an example, Demeyer et al. [10] measure successive versions of a software system in order to discover which refactorings have been applied from one version of the software to the next. Based on three small to medium-sized case-studies, they conclude that it is possible to reverse engineer where, how —and sometimes even why— an implementation has drifted from its original design.

Ramil and Lehman [31] applied metrics to the *long-term* evolution (more than one decade) of a system, to determine whether productivity has significantly changed over this period. More specifically, they discuss the *detection of change points*, i.e., points in time that highlight sudden changes in evolutionary behaviour (in this case, productivity changes). In [32], metrics were used to assess *cost estimation* on the evolution of the same software system.

4. CONCLUSION AND FUTURE WORK

In this paper, we presented a classification of the various approaches that use metrics to understand, predict, plan and control software evolution. The classification consisted of two main categories: *predictive analysis* before the evolution (in order to detect evolution-critical, evolution-sensitive and evolution-prone parts of the software) and *retrospective analysis* after the evolution has taken place (subdivided in software and process). For each of the categories, concrete examples and references to the literature were provided. Based on this overview, we have identified a number of topics for further research.

Coupling/cohesion metrics. Coupling and cohesion are used to measure a system's structural complexity, and can be used to assess design quality and to guide improvement efforts [37]. Numerous metrics have been proposed to quantify coupling and cohesion ([2, 4, 5, 17, 25]). Unfortunately, there is strong disagreement in the literature about what constitute good coupling and cohesion metrics [12, 3], hence we are reluctant to use them as indicators for evolution-critical and evolution-sensitive system parts. We feel that coupling and cohesion are too coarse criteria and that they should be complemented with finer-grained factors (within a method, a class, a module; via variables, attributes, invocations, inheritance, ...). Then it will be easier to assess the trade-off involved in any design activity, which would make it possible to see whether a system is evolving in the right direction.

Scalability issues. Scalability is always an issue in software engineering and in that respect metrics look particularly appealing. However, analysing evolving software all too easily results in an exponential explosion of the number of measurements to be interpreted. *Visualisation* seems an interesting path to explore as pictures have the intrinsic ability to help the human eye (and brain !) to focus on the most relevant issues at a glance. However, the key question is which visualisation to use for interpreting which measurements. Given the number of possible combinations, this is a vast area to explore.

Empirical research and realistic case-studies. In the context of software evolution there is a need for more empirical research [13, 20]. This is even more the case when dealing with evolution metrics, where we can only provide significant results if the approach has been tested on sufficiently large sets of representative examples. Unfortunately, much of the current-day research makes an initial evaluation of a given technique based on a case study that relies on a single small toy-example chosen to favour the technique under study. This has a serious impact on the credibility of the results, especially concerning the generalisability. When each researcher picks his own toy-example it becomes very difficult to compare results. Therefore we should agree on a set of case-studies which together are representative for the kinds of problems we want to solve. Given the amount of open-source projects available today, it should be feasible to agree on such a benchmark [11].

Long term evolution. It is also important to look at the long term evolution of software, which might give a different perspective on the nature of the software system under consideration.

Detecting different kinds of evolution. Initial experiments have indicated that metrics can detect different kinds of evolution, such as restructuring and extension. Nevertheless, it remains an open question which other kinds of software evolution can be identified by metrics, and which metrics are most appropriate for finding a particular kind of software evolution.

Change-based configuration management A final issue that needs further work is tools that maintain precise logs concerning the changes that have been applied in order to achieve the next version of the software. Currently, most configuration management systems are *state-based*: they only record the course-grained intermediate states, not the precise changes as they have been applied [7]. As such, one must use heuristics to infer the changes from the intermediate states, but then it is not possible to assess the precision of the heuristics. *Change-based* configuration management tools do not have this shortcoming, but are not very widely used.

Measuring software quality. An important but very difficult research topic remains how metrics can be used to measure the quality of software (with respect to a certain goal such as, e.g., reusability) and to measure whether the quality has improved or degraded between two releases of the software.

Process issues. Another essential, but very difficult topic is how to measure or predict changes in the evolution process. This includes issues such as programmer productivity, cost estimation and effort estimation [31, 32].

5. ACKNOWLEDGMENTS

This research is carried out as part of the International Research Network on *Foundations of Software Evolution*, involving nine research institutes from five different European countries (Belgium, Germany, Portugal, Switzerland, Austria). The network is financed by the Fund for Scientific Research - Flanders (Belgium).

We thank the anonymous referees for their useful comments and suggestions.

6. REFERENCES

[1] T. Ball and S. G. Eick. Software visualization in the large. *IEEE Computer*, 29(4), April 1996.

[2] J. M. Bieman and B.-K. Kang. Cohesion and reuse in an object-oriented system. In *Proc. Symp. Software Reusability*, pages 259–262, April 1995.

[3] L. C. Briand, J. Daly, and al. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.

[4] F. Brito e Abreu, M. Goulao, and R. Esteves. Toward the design quality evaluation of object-oriented software systems. In *Proc. 5th Int'l Conf. Software Quality*, pages 44–57, October 1995.

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

[6] D. Coleman, D. Ash, B. Lowther, and P. Oman. Using metrics to evaluate software system maintainability. *IEEE Computer*, pages 44–49, August 1994.

[7] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2), June 1998.

[8] S. Demeyer and S. Ducasse. Metrics: Do they really help? In *Proc. Languages et Modèles à Objets*, pages 69–82. Hermes Science Publications, 1999.

[9] S. Demeyer, S. Ducasse, and M. Lanza. A hybrid reverse engineering approach combining metrics and program visualization. In *Proc. Working Conf. Reverse Engineering (WCRE '99)*. IEEE Computer Society Press, October 1999.

[10] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proc. Int. Conf. OOPSLA 2000*. ACM Press, 2000.

[11] S. Demeyer, T. Mens, and M. Wermelinger. Towards a software evolution benchmark. In *Proc. Int. Workshop on Principles of Software Evolution*, September 2001.

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

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

[14] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *International Conference on Software Maintenance (ICSM '98)*. IEEE Computer Society Press, November 1998.

[15] T. Guimaraes. Managing application program maintenance expenditure. *Comm. ACM*, 26(10):739–746, 1983.

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

[17] M. Hitz and B. Montazeri. Measuring coupling and cohesion in object-oriented systems. In *Proc. Int. Symp. Applied Corporate Computing*, pages 25–27, October 1995.

[18] M. Jazayeri, C. Riva, and H. Gall. Visualizing software release histories: The use of color and third dimension. In H. Yang and L. White, editors, *Proc. Int'l Conf. Software Maintenance (ICSM '99)*. IEEE Computer Society, 1999.

[19] H. Kabaili, R. K. Keller, and F. Lustman. Cohesion as changeability indicator in object-oriented systems. In P. Sousa and J. Ebert, editors, *Proc. 5th European Conf. Software Maintenance and Reengineering*, pages 39–46. IEEE Computer Society Press, 2001.

[20] C. Kemerer and S. Slaughter. An empirical approach to studying software evolution. *IEEE Trans. Software Engineering*, 25(4):493–509, July/August 1999.

[21] K. Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *Proc. Working Conf. Reverse Engineering (WCRE'97)*, pages 44 – 54. IEEE Computer Society Press, 1997.

[22] B. Laguë, D. Proulx, E. M. Merlo, J. Mayrand, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proc. Int'l Conf. Software Maintenance (ICSM'97)*. IEEE Computer Society Press, 1997.

[23] M. Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proc. Int'l Workshop on Principles of Software Evolution (IWPSE2001)*, 2001.

[24] C. Lewerentz and F. Simon. A product metrics tool integrated into a software development environment. In S. Demeyer and J. Bosch, editors, *Object-Oriented Technology (ECOOP'98 Workshop Reader)*, LNCS 1543, pages 256 – 257. Springer-Verlag, 1998.

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

[26] B. P. Lientz and E. B. Swanson. *Software maintenance management: a study of the maintenance of computer application software in 487 data processing organizations*. Addison-Wesley, 1980.

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

[28] R. Marinescu. Using object-oriented metrics for automatic design flaws in large scale systems. In S. Demeyer and J. Bosch, editors, *Object-Oriented Technology (ECOOP'98 Workshop Reader)*, LNCS 1543, pages 252–253. Springer-Verlag, 1998.

[29] P. Nesi. Managing OO projects better. *IEEE Software*, July 1988.

[30] W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[31] J. F. Ramil and M. M. Lehman. Metrics of software evolution as effort predictors - a case study. In *Proc. Int. Conf. Software Maintenance*, pages 163–172, October 2000.

[32] J. F. Ramil and M. M. Lehman. Defining and applying metrics in the context of continuing software evolution. In *???*, 2001.

[33] A. J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley Publishing Company, April 1996.

[34] F. Simon, F. Steinbrückner, and C. Lewerentz. Metrics based refactoring. In *Proc. European Conf. Software Maintenance and Reengineering*, pages 30–38. IEEE Computer Society Press, 2001.

[35] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press / Addisson-Wesley, 1998.

[36] R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proc. Int'l Conf. TOOLS USA'98*, pages 112–124. IEEE Computer Society Press, 1998.

[37] E. Yourdon and L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice Hall, 1979.