ELSEVIER

Information and Software Technology 44 (2002) 593–600

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.com/locate/infsof

Measuring software evolution at a nuclear fusion experiment site: a test case for the applicability of OO and reuse metrics in software characterization

G. Manduchi*, C. Taliercio

Associazione EURATOM-ENEA per Ricerche sulla Fusione, CONSORZIO RFX, Corso Sati Uniti 4, 35127 Padova, Italy

Received 30 November 2001; revised 22 April 2002; accepted 30 April 2002

Abstract

A set of software metrics has been used to provide empirical evidence on how code organization changes when a software product evolves. A Java application for graphical data display used in experimental physics has been used as a test case. Exploiting common patterns in the way software applications evolve is desirable as it would give designers and managers a better understanding of the software process. The analysis in which framework reuse has also been considered, highlighted a limited use of the inheritance mechanism and, despite an increase in the overall complexity, a substantial invariance of the internal application organization. This fact is explained by the increasing framework usage and integration during the product's lifetime. © 2002 Elsevier Science B.V. All rights reserved.

Keywords: Software evolution; Object-oriented metrics; Reuse metrics

1. Introduction

Software metrics have been proposed as quality indicators for software projects, based on the statement that an important component of process improvement is the ability to measure the process. The aim of such approach is to identify components in the software system which are likely to be faulty, so that the testing and verification effort can be concentrated on these modules. The simplest measure of software is the number of Lines of Code (LOC). Other metrics have been proposed in order to capture better the effective complexity of a program and to provide a tighter correlation with defect density. For example, the McCabe cyclomatic number [1] takes into account the program flow graph under the assumption that the effective complexity of a program lies in its structure rather than in a mere statement count. In order to provide a formally sound base, a set of general properties for software metrics has been proposed in Ref. [2]. An example of these general properties is that the complexity of a program P has to be lower or equal than the complexity of the concatenation of the program P and another program Q. More generally, the properties presented in Ref. [2] provide a formal method for stating how 'good' are the various metrics, i.e. how much related they are with the effective complexity of a program. These properties have been used in several subsequent works in order to provide a sound base for the proposed metrics.

More recently, a set of metrics tailored to Object-Oriented (OO) design has been introduced by Chidamber and Kemerer [3]. These metrics, known also as C & K metrics, have been widely used in complexity measurements for OO systems [4,5] and proved valid, to a certain extent, in the identification of those classes which are more error prone and therefore deserve additional verification effort. The C & K metrics aim at a characterization of the class organization. In particular, some metrics are related to the inheritance structure, while others refer to the number of methods and the coupling among different classes. If these metrics (or a subset of them) were proved to be related to defect rate, it would be possible to develop automated tools supporting software managers in the identification of critical components in a software system. It is worth noting that the full collection of the C & K metrics cannot be achieved by simple analysis of code sources such as line counting. Due to the richness of the syntax for OO languages a syntactical parser is generally required for extracting the required information.

Software metrics have been used in a different

* Corresponding author. Tel.: +39-049-8295039; fax: +39-049-8700718.

E-mail address: manduchi@igi.pd.cnr.it (G. Manduchi).

PII: S0950-5849(02)00079-4

perspective in Ref. [6]. In that work the C & K metric suite has been used to provide an empirical evaluation of the impact of a framework in the code organization. For this purpose, descriptive statistics of the C & K metrics have been used to compare the complexity of a Java application, including all the reused classes of the Java framework, with the complexity of the application classes alone. Frameworks represent in fact an important factor in modern OO systems. Today's fast moving and demanding world of information technology requires that software applications have to be designed, built and produced for less money, with greater speed and with fewer resources than ever before. Consequently, in-house development of all the required functionality is unaffordable and the system architecture needs to be defined so that it can take full advantage from existing (commercial or free) frameworks. A framework usually defines a set of ready-to-use classes for common functionality such as graphical user interfaces, I/O and networking. However, it can also define patterns of interactions for a generic environment which is then targeted by the development of the classes implementing the specific business domain. As an example, the classes of the Java Run Time Environment (JRE) greatly simplify and speed the development of Java applications, while Java Enterprise Beans (J2EE) provide a component-based framework for the design, development, assembly and deployment of enterprise applications.

It is not easy to provide a formal evaluation of the degree of integration of the framework in the application code, however reuse measurements can provide an indication. As stated before, the use of a framework consists of the integration of ready-to-use objects for well-defined operations, and the integration of application classes into the generic organization provided by the framework. While in the first case it is likely that ready-to-use objects are integrated since the earlier versions of the application (provided that the programmers have already a basic knowledge of the framework), the integration of application classes into the framework interaction patterns requires often both a good knowledge of the framework organization and a deeper insight into the application organization, which may be gained after the first releases of the software products. As an example, every Java programmer knows how to use container objects for handling vectors, lists and hash tables, but the optimal integration of the design patterns such as Observer for event management [7], or the Model View Controller used in the Swing framework requires an iterative approach, involving often some code reorganization.

The incidence of framework usage in a software system has been analyzed in Ref. [8], where it is shown how the use of framework components and of common patterns of interaction contributed to a significant gain in productivity and quality, in Ref. [9] where significant benefits from reuse in terms of reduced defect density and rework are reported, and in Ref. [10] where a revision of the C & K metrics is proposed in order to take into account the reuse of software components.

As a software project evolves, changes in its parts are likely to occur, possibly introducing a new component organization. A better understanding of the way software organization evolves during the lifetime of a product may provide useful information in the design phase for new products. Research on empirical approaches to study software evolution is relatively scarce. The seminal work in this area reports the study of 20 releases of the OS/360 operating system and states five laws of software evolution dynamics, based on a number of observations about the size and complexity growth of the system [11]. Other research works concentrated on the study of the defect data taken during the lifetime of software systems, in order to devise a methodology for allowing managers to take actions to improve process and system performance [12]. Information taken from bug reports in a large commercial system has also been used in Ref. [13] where it is argued that the conventional time series model does not exploit the richness of the coding categories. The proposed alternative approach defines first a rich set of detailed change events, and then seeks to determine whether these patterns coalesce into phases along which system progresses.

While in the above works the parameters considered in the analysis referred to the complexity of the system and the defect rate, in Ref. [14] the distribution of parameters related to the OO organization (number of lines per method and number of methods per class) is considered during several stages of development for some software systems, in order to analyze evolution patterns of objects and construct an object evolution model. C & K measurements taken at different stages of the lifetime of a commercial C++ applications have also been reported in Ref. [5], even though the main purpose of that work was the analysis of the correlation between measured C & K metrics and observed defect rate.

The main objective of this work is to investigate the utility of OO metrics for the understanding of the software evolution process. For this purpose, we shall present a case study showing the evolution of the C & K and reuse parameters during the lifetime of a software project, in order to characterize patterns of evolutions both in the internal software organization and in the integration of external frameworks. The above metrics taken at three distinct phases of the lifetime of a Java-based application for data visualization in a nuclear fusion experiment will be compared in order to highlight which aspects of the software organization evolved, and which did not appear to change during a product's lifetime.

Exploiting common evolution patterns in object organization may provide useful information for the initial definition of software architectures. For example, the evidence of little inheritance restructuring during a product lifetime, along with an increasing system complexity, would indicate that new components tend to be added using object

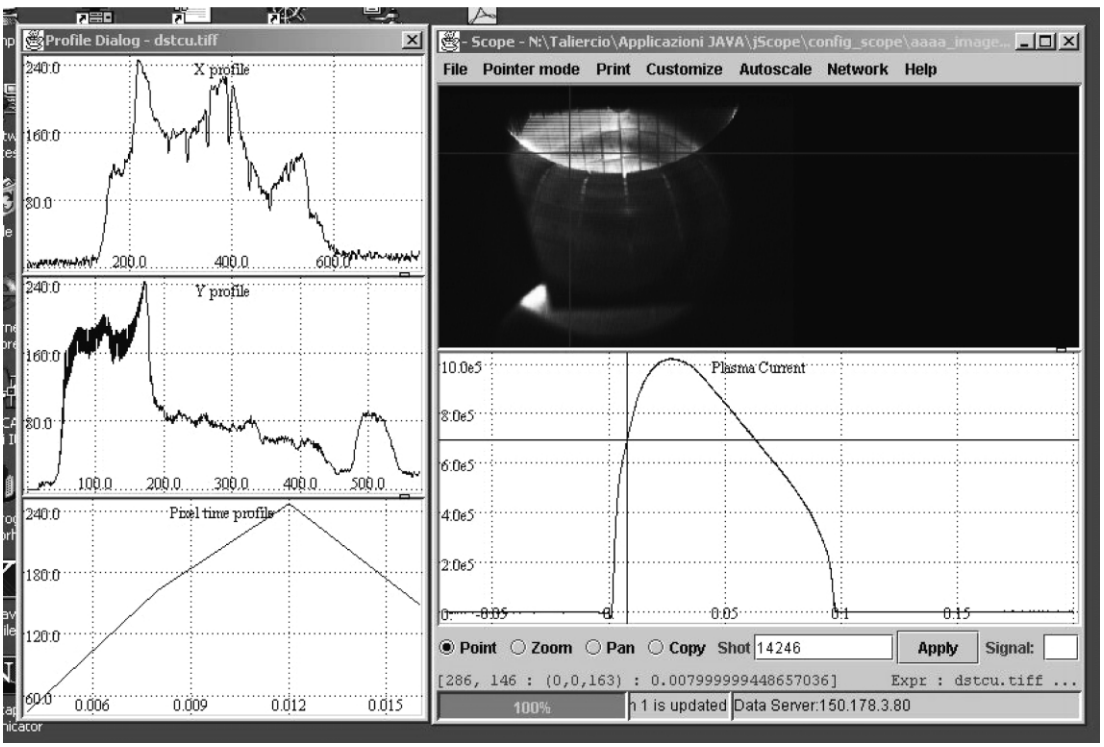

Fig. 1. A sample data display of the data visualization tool showing both acquired waveforms and image sequences.

composition, rather than by specializing generic components. This would suggest an architectural organization in which object composition is preferred over inheritance, paying more attention to the definition of interaction patterns.

2. System description

The system analyzed in this case study belongs to a suite of tools for data visualization used in several nuclear fusion experiments [15]. It is written in Java and provides tools for interactive waveform display and visualization of frame sequences (Fig. 1). Developed originally for a specific data platform, the tools have been then made generic so that they can be employed in different experiments using proprietary data acquisition systems. A set of Java interfaces define the data access layer, which has to be implemented to adapt the data visualization tools to a new experiment. Support for remote data access is also provided in order to allow visualization of data acquired in other experiments. This feature is important as fusion research is today moving towards the collaborative use of fewer and larger machines, and laboratories therefore often participate in the conduction of experiments located elsewhere.

The tools are entirely written in Java and make extensive use of the Java framework for graphics, data manipulation and network communication. The interaction patterns used in the Java framework have been borrowed in the organization of this application. For example, the Observer pattern adopted in the organization of events in graphical interfaces suggested a more coherent interaction among communication classes than that used in the first releases of the tools. Other design patterns have been imported by the Java framework, improving the organization of application classes. This process however required an iterative approach, as patterns are often the outcome of the experience gained in the development of previous product versions.

In the following we shall concentrate on one particular tool of the suite. The reason is that, since we are interested in characterizing how software organization evolves during a product's lifetime, the selected tool represents the component of the suite most used in laboratories and its evolution is based on feedback collected from its user base. Consequently, the evolution of such component is more similar to that of other commercial packages than other in vitro developments presented as case study in literature.

The development of the presented application started in 1996, and since then several versions of the tools have been developed. The application is still evolving as the consequence of a growing user base introducing new requirements.

Two major code rearrangements have been carried out during the product lifetime. The first change is due to the adoption of the Swing graphical interface which replaced the AWT-based one. In a second major rearrangement, a set of design patterns has been introduced in order to make the tools easily expandable. Minor changes are frequently

carried out to integrate new graphical features or to support new protocols for remote data access.

The two major rearrangements represent two typical modifications which are likely to occur during the lifetime of a software product. The first one represents, in fact, an adaptation of the system to the evolving technology, while the second one is a consequence of the need for maintaining continuous ability in providing new tasks or performing old ones under new conditions. This is usually accomplished by the insertion of new components, but it may happen that rearrangements in the underlying architecture are required in some stages in front of new unforeseen requirements, or when the gained experience indicates a better underlying organization.

3. Metrics overview

In the following we shall give a brief description of the metrics used to quantify reuse and class organization.

3.1. Metrics definition

Considering reuse, several criteria have been introduced in the literature [8,16]. The extent of reuse defined as the *source* criterion, can be public or private. Public reuse means that the reused component is external to the application, while in private reuse components are internal to the application. The *extension* criterion defines whether reuse occurs directly or indirectly through some intermediate component. The *shape* criterion defines whether components are used with or without modification. The first case corresponds to subclassing and is called *leveraged* reuse. The second case corresponds to object instantiation and is called *verbatim* reuse. Finally, the *perspective* criterion defines the client or the server perspective. The client perspective provides an indication on how much used are the framework classes. The server perspective gives an indication on how general are the framework components, i.e. how many clients use a given framework component.

In the analysis we shall restrict our attention to the client perspective, considering both leveraged and verbatim reuse. We consider here only the client perspective since we are interested in the organization of the application classes. The measurement of the public reuse provides an indication on the extent of integration of the Java framework classes in the application, while private reuse measurement gives an indication on the degree of inter-correlation among application classes.

The C & K metric suite used for the collection of OO metrics is composed of the following:

- Weighted Method per Class (WMC). Used to measure the complexity of class implementation, and defined as the sum of the complexity of each method defined in the class. WMC is usually simplified by considering the number of the methods for the given class.
- Depth of Inheritance Tree (DIT). Defined as the maximum depth of the inheritance tree for each class. Since Java does not allow multiple inheritances, DIT simply gives the number of superclasses for the given class. Interfaces are not considered in the counting as they do not provide method implementation.
- Number Of Children (NOC). The number of direct descendants for each class.
- Response For a Class (RFC). The number of methods that can be executed in response to a message received by the class, restricted to the first level of nesting of method calls.
- Coupling Between Objects (CBO). The number of coupled classes for each class. Class A is coupled with class B if some methods or fields of A are used by the methods of B.
- Lack of Cohesion On Methods (LCOM). The number of pairs of methods which do not share instance variables minus the number of method pairs which share instance variables. If the result is negative, the metric is set to 0.

As the software is composed of both application and framework classes, we shall consider two versions of the DIT metric: *internal* and *external* DIT. Internal DIT is computed considering only application classes. In this case we shall ignore the fact that an application class derives from a framework class. In external DIT, the measurement considers also inheritance from Java framework classes. A similar approach has been taken in Ref. [11] to quantify reuse sensitiveness of complexity metrics.

3.2. Metrics collection

In order to collect the necessary information for the evaluation of the reuse and C & K metrics, a syntactical analysis of the Java source code of all the application and framework classes is required. For this purpose we developed a parser for the Java language which collects structural information such as classes, fields, methods and inheritance structure. Based on the collected information, it is then easy to develop analysis tools for the collection of the desired metric measurement.

The use of a syntactical analyzer is necessary to properly handle OO language issues such as method overloading, which requires not only method names, but also the actual types of method parameters. The parser has therefore to track down the actual type of the arguments for every method call in order to uniquely identify the called methods.

Two Java specific language constructs, namely interfaces and inner classes, deserve a further consideration, as they are not considered in the C & K metrics. Interfaces in Java define a collection of methods without implementation. A class may declare to implement a given interface, provided it gives an implementation of the methods defined in the

Table 1
Reuse metrics in the considered stages of the sample application

	First release	Intermediate release	Last release
Average number of used Java classes per application class	6.5	7.4	7.9
Average number of used application classes per application class	3.15	3.25	2.90

interface. Unlike class hierarchy, which defines only single inheritance, a class may implement more than one interface. Interfaces are typically used when a given functionality is required, but there is no need to know how such functionality is implemented. In this case the exact nature of the class providing the required methods needs not to be known and is determined run-time. A correct interpretation of interfaces in reuse metrics depends on the perspective into which the measurement is performed. If reuse metrics are used to estimate the amount of framework code effectively reused in the application, it makes sense to consider all the classes implementing a given interface. In other words, under this interpretation, if class A uses the methods declared in an interface, then every class B implementing such interface is assumed to be used by class A. If, on the other side, the goal is to estimate the complexity of the framework usage from the client perspective, a better interpretation is to consider interfaces alone. From the client's point of view, in fact, an interface hides completely the actual implementation and it is used not differently from other classes. The client is aware of the actual implementing class only if it is required to directly instantiate the class implementing the interface. This is typically not the case in the Java framework which uses extensively creational patterns (such as Abstract Factory) which delegates to a separate (Factory) component the actual instantiation of interface implementation.

Inner classes, introduced in JDKI 1.1, are typically used when a class instance is required to freely access the fields of a container class instance. Using inner classes represents an effective way to avoid the proliferation of object references as class fields, thus contributing in keeping the code organization clearer, but conceptually they are not much different from another class definition. For this reason in analysis inner classes have been considered as they were declared outside the container class. In this case the access to a field of the container object is considered as a conventional field access. Anonymous classes are instead treated differently and have not been considered independent classes, rather as segments of code declared inside the container class. The reason for this choice is that anonymous classes are normally used as adapter classes and normally have a very short body which simply calls some method of the container class.

Table 2
C & K metrics in the considered stages of the sample application

	First release	Intermediate release	Last release
WMC	11.13	11.89	13.6
Internal DIT	1.31	1.33	1.35
External DIT	2.56	2.71	3.16
NOC	0.31	0.33	0.26
CBO	9.75	11	11.51
RFC	33.6	37.8	41.3
LCOM	77.4	110.6	189.96

4. Results

The values for the considered metrics are listed in the following three tables which report the measurements taken at the three considered stages of the tool evolution.

The first row of Table 1 refers to external reuse and reports the average number of different Java framework classes used per application class, considering both verbatim and leveraged reuse. The reason for considering both verbatim and leveraged reuse derives from the fact that often either inheritance and object composition can be used to achieve the same goal. For example, to create a new form using Swing components, it is possible to derive a subclass of *Javax.swing.JFrame* whose constructor creates all the required graphical components to be inserted in the frame. Alternatively, an object of the same class can be instantiated within another class' method which then adds the components to the created instance.

From Table 1 it can be seen that the average number of used classes per application class keeps increasing during the system lifetime, thus indicating an increasing integration of the Java framework into the application.

The second row of Table 1 illustrates the internal reuse, that is the average number of used application classes for each class. It can be seen that internal reuse metric does not exhibit a significant trend during the application lifetime, giving an indication that the complexity of the internal class organization tends to remain unchanged after the first product release.

This fact appears to be confirmed by several results reported in Table 2, showing the C & K metrics for the three versions of the considered tool. In particular, looking at the DIT metric when considering only the application classes hierarchy (internal DIT), we can see that the metric is essentially the same in all the three versions of the tool. This suggests that little restructuring of inheritance hierarchy took place during the evolutionary process. The same consideration applies to the NOC metric for the application classes.

Table 3
Other metrics in the considered stages of the sample application

	First release	Intermediate release	Last release
Application classes	65	71	103
NLOC	12,300	17,000	24,300
Average NLOC per method	14.96	16.96	13.5
Number of used Java classes	90	108	144

If we consider instead the DIT metric considering also the Java framework classes in the hierarchy (external DIT), the metric exhibits an increased average depth, suggesting therefore an increased integration of the Java framework during the product lifetime. This fact is also confirmed by the increasing value of the CBO metric, indicating an increasing number of involved framework classes per application class. As the average number of reused application classes does not significantly change during the product's lifetime, the increase in the CBO metric can be explained by an increased reuse of framework classes.

Other two metrics exhibit a significant trend across the three versions of the considered application. The RFC metric, which indicates the number of different methods which can be activated per method call, can still be interpreted as an indicator of an increased usage of framework classes, still based on the fact that the internal class organization does not significantly change.

The increase in the LCOM metric can be explained by the increased usage of accessor methods in the application classes. As stated before, the patterns of interaction introduced by the Java framework guide the developer towards a more abstract view of the used classes, using interfaces when possible. As a consequence, accessor methods have been increasingly used instead of direct field access. Each accessor method normally represents the public interface of a given field, and therefore does not share instance variable with other accessor methods.

Table 3 reports other metrics for the considered application. The first row displays the number of classes developed in the application. The number of Non-Commented Lines of Code (NLOC) is shown in the second row, while the third row shows the average NLOC per method. It is worth noting that while the number of NLOC per method does not exhibit a significant change, the average number of methods per class (WMC metric, reported in the first row of Table 2) slightly increases, probably as a consequence of the increased number of accessor methods. In any case the complexity of single class' implementations appear not to increase despite the functionality added during the system lifetime, reflected instead in the increased number of application classes.

5. Discussion

The above DIT, RFC and CBO measurements indicate an increase in the overall system complexity. This represents a common pattern in software evolution, as summarized in Lehman's Continuing Growth law stating, "The functional content of a system must be continually increased to maintain user satisfaction over their lifetime" [17]. Considering the presented case study, during its lifetime the system has been integrated with new graphical features in data visualization, the management of new protocols for data communication, improved compression techniques and web support. Most times the required new tasks have been implemented by integrating new components in the existing architecture, but in several occasions a change in the component interaction patterns has been required, usually to make component interaction more generic in order to accomplish unforeseen requirements.

Despite the increasing system complexity, the parameters related to the internal class organization (internal DIT and number of used application classes per class) give an indication towards few changes in the complexity of the internal organization of single classes. This would therefore indicate that the added functionality has been mostly achieved by integrating new components. This result is not surprising as it represents an empirical evidence of a widely accepted statement. The OO development process concentrates in fact on the architecture, i.e. on how classes are organized and their interaction, as an important factor for the success of a software product is the ability of supporting the integration of new features without the need of a deep modification in the code organization. A careful definition of the architecture in the early stages of the project represents the key concept of modern development methods, such as the Unified Software development process [18]. A good architecture represents in this case the 'skeleton' of the application which remains basically unaltered during the development (which adds the 'muscles') and during the lifetime of the products during which new features are likely to be integrated.

Another observation derives from the DIT and NOC measurements showing that the use of class inheritance is not so high as it would be expected from the usage of the Java language. Considering also the time evolution of the

DIT and NOC parameters,[1] it can be seen that a limited use of inheritance has been done during all the stages of the product lifetime. A more accurate inspection in the class structure of the software under study showed that the increment in mean external DIT was mainly due to the increased number of framework classes, having an average inheritance depth greater than the application classes. These classes were however mainly used in composition, rather than inherited. It is worth noting that there is in practice almost no need for the inheritance mechanism when using the components of the JRE framework. This does not mean that the functionality of the Java framework cannot be integrated using inheritance, but that inheritance can most times be substituted by object composition. For example, graphical interfaces can be built by subclassing *javax.swing.JDialog* or *javax.swing.JFrame*, and overriding their constructor in order to add the required components. The same result can however be obtained by using the above classes in an independent class, whose constructor instantiates the container and the contained elements. Similarly, multithreading can be achieved in Java either by inheriting from the *java.lang.Thread* class or developing independent classes implementing the *Runnable* interface, and passing their instances to the constructor of Thread objects.

The reason for this fact derives from the basic philosophy of the Java framework organization in which framework classes implement generic algorithms handling generic components. The generic functionality is then specialized by providing application-specific components adhering to a well-defined set of interfaces, and *not* by redefining the generic methods in derived subclasses. This last approach would necessarily involve the inheritance mechanism, while the first one can be achieved either by inheritance or composition. The consequence of this organization is that it is possible to hide completely the internals of the framework classes whose functionality is always accessible through a set of public methods without exposing protected class members. This choice is dictated by experience: every experienced programmer knows that the use of a programming interface for accessing components' functionality is less error prone than exposing internal instance variables. This is also confirmed by several studies: in Ref. [19] it is shown for example that there is a significant difference in the defect density between those classes that participate in inheritance structure and those which did not. The tendency of software developers in avoiding an extensive use of inheritance is reported also in Ref. [20] and the fact that little class restructuring occurs during the lifetime of a software project is highlighted in Ref. [5], reporting C & K measurements not far from ours, despite the fact that the authors analyzed a software product written in C++.

It is worth noting that the above discussion does not mean that the inheritance mechanism has to be avoided 'tout court'. Inheritance has been successfully used inside the considered application to insulate generic algorithms, parts of which are implemented by methods which are then overridden by derived classes. This allowed moving a significant amount of code towards more generic superclasses, thus effectively increasing code reuse inside the application. In this case it makes sense to expose the superclass' structure, as subclasses are typically implemented by the same developers. In any case the reported measurements and experience give a clear indication that great care is required when the inheritance mechanism is used to import external components.

A final observation derives from the LCOM measurements. The LCOM parameter proved to be mainly dependent on accessor methods. Typically, such a method accesses only a single field of the class: being the use of accessor methods preferred over direct field access in order to improve decoupling, the LCOM parameter appears to be related to the number of accessor methods rather than other structural properties. This agrees with the results reported in other works, such as in Ref. [4] in which no correlation between LCOM and the probability of faults is found.

6. Conclusions

This work presented an investigation on the effectiveness of some OO metrics in understanding the software evolution process. For this purpose, a set of software metrics taken at three different stages of the lifetime of a Java application has been presented in order to provide empirical evidence on how code organization changes when a software product evolves. It is worth noting that results derived from software metrics collection do not provide by themselves a classification of the software organization. In other words, programs with a completely different organization could in principle be characterized by the same value for a given software metric. Software measurement results in any case provide an information which, when integrated with other metrics, may suggest evidence for several aspects of software organization.

It is also worth noting that this work represents a case study and therefore the presented results need to be confirmed by further measurements considering different software applications.

Both reuse and C & K metrics give evidence to the fact that little class restructuring took place during the product lifetime. In other words, the original class organization seems to have been maintained and the integration of new features has been accomplished by integrating new classes rather than changing the current organization. While this case study provided evidence of little class restructuring, the C & K and the reuse metrics showed an evolution towards a deeper integration of the Java framework classes into the application. It is worth considering that the project started in 1996, and the developers acquired further experience after

[1] Recall that in Java every class inherits from Java.lang.Object and therefore has a DIT of at least 1.

the first release of the tool. This fact is probably common to many Java applications developed in the same period, but may change for new software projects whose developers have already a mature experience of the Java framework.

In any case, the full comprehension of the many facets of the Java framework requires years of experience and we can therefore expect that even future software projects in Java will exhibit an increasing integration of the framework during their lifetime.

References

[1] T.J. McCabe, A complexity measure, IEEE Transactions on Software Engineering 2 (4) (1976) 308–320.
[2] E.J. Weyuker, Evaluating software complexity measures, IEEE Transactions on Software Engineering 14 (9) (1988) 1357–1365.
[3] S.R. Chidamber, C.F. Kemerer, A metric suite for object oriented design, IEEE Transactions on Software Engineering 20 (6) (1994) 476–493.
[4] V.R. Basili, L.C. Briand, W.L. Melo, A validation of object oriented design metrics as quality indicators, IEEE Transactions on Software Engineering 22 (10) (1996) 751–761.
[5] F.G. Wilkie, B. Hylands, Measuring complexity in C++ application software, Software—Practice and Experience 28 (5) (1998) 513–546.
[6] G. Manduchi, Developing Java applications for a nuclear fusion experiment: a test case for Java applicability in a demanding environment, Software—Practice and Experience 31 (2001) 1025–1042.
[7] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns, Addison-Wesley, Reading, MA, 1994.
[8] S.A. Mamrak, S. Sinha, A case study: productivity and quality gains using an object-oriented framework, Software—Practice and Experience 29 (6) (1999) 501–508.
[9] V.R. Basili, L.C. Briand, W.L. Melo, How reuse influences productivity in object-oriented systems, Communication of the ACM 39 (10) (1996) 104–119.
[10] T. Kamija, S. Kusumoto, K. Inoue, Y. Mohri, Empirical evaluation of reuse sensitiveness of complexity metrics, Information and Software Technology 41 (1999) 297–305.
[11] L.A. Belady, M.M. Lehman, A model of large program development, IBM Systems Journal 15 (1) (1976) 225–252.
[12] V.R. Basili, L.C. Briand, S. Condon, Y.M. Kim, W. Melo, J. Valett, Understanding and predicting the process of software maintenance releases, Proceedings of 18th International Conference on Software Engineering, Berlin (1996).
[13] C.F. Kemerer, S. Slaughter, An empirical approach to studying software evolution, IEEE Transactions on Software Engineering 25 (4) (1999) 493–509.
[14] T. Tamai, T. Nakatami, An empirical study of object evolution processes, Proceedings of International Workshop on Principles of Software Evolution, Kyoto (1998) 33–37.
[15] G. Manduchi, A. Luchetta, C. Taliercio, The Java interface of MDSplus: towards a unified approach for local and remote data access, Fusion Engineering and Design 48 (2000) 163–170.
[16] F. Brita, E. Abreu, R. Carapuca, Candidate metrics for object-oriented software within a taxonomy framework, Journal of System Software 26 (1994) 87–96.
[17] M.M. Lehman, J. Ramil, P.D. Wernick, D.E. Perry, W.M. Turski, Metrics and laws of software evolution—the nineties view, Proceedings of the Fourth International Symposium on Software Metrics (Metrics 97) 97 (1997) 20–32.
[18] I. Jacobson, G. Booch, J. Rumbaugh, The unified software development process, Addison-Wesley, Reading, MA, 1999.
[19] M. Cartwright, M. Shepperd, An empirical investigation of an object-oriented software system, IEEE Transactions on Software Engineering 26 (8) (2000) 786–796.
[20] S.R. Chidamber, D.P. Darcy, C.F. Kemerer, Managerial use of metrics for object-oriented software: an exploratory analysis, IEEE Transactions on Software Engineering 24 (8) (1998) 629–639.