International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 2 Issue 5, May - 2013

Performace Evaluation In Object Oriented Metrics

Mr. S. Pasupathy [1] and Dr. R. Bhavani [2]
[1] Associate Professor, Dept. of CSE, FEAT, Annamalai University, Tamil Nadu, India.
[2] Professor, Dept. of CSE, FEAT, Annamalai University, Tamil Nadu, India.

Abstract:

This paper presents the results derived from our survey on metrics used in object oriented environments. Our survey includes a small set of the most well known and commonly applied traditional software metrics which could be applied to object oriented programming and a set of object oriented metrics. In short, the metrics based assessment of a software system and measures taken to improve its design differ considerably from tool to tool. To support our case, we conducted an experiment with a number of commercial and free metrics tools. We calculated metrics values using the same set of standard metrics for three software systems of different sizes. These metrics were evaluated using object oriented metrics tools for the purpose of analyzing quality of the product, encapsulation, inheritance, message passing, polymorphism, reusability and complexity measurement. It defines a ranking of the classes that are most vital note down and maintainability. The results can be of great assistance to quality engineers in selecting the proper set metrics for their software projects and to calculate the metrics, which was developed using a chronological object oriented life cycle process.

Index: Software Engineering, Object Oriented Programming Concepts, Reusability, Performance Estimation.

1. INTRODUCTION

In software development, a metric (noun) is the measurement of a particular characteristic of a program's performance or efficiency. Similarly in network routing, a metric is a measure used in calculating the next host to route a packet to. A metric is sometimes used directly and sometimes as an element in an algorithm . In programming, a benchmark includes metrics. Metric (adjective) pertains to anything based on the meter as a unit of spatial measurement. One metric alone is not enough to determine any information about an application under development. Several metrics must be used in tandem to gain insight into improvements during a software process. There are several software packages that can be used to determine the metrics on a software applications.

The data harvesting mechanism gathers data daily from the production system, and loads it into a reporting warehouse. Reports are generated based on the warehouse data. The host administrator can schedule the time for the daily data harvesting operation. Changes in Project Tracker artifacts or Subversion activity are not immediately available for Metrics reports. They can be reported on only after a data harvesting operation has occurred. When a report is generated, the results page shows the date of the last data harvesting operation. Users with the "Project - Edit" permission can define reports and store them on the Project Metrics landing page [1,2]. Users with the "Project Content - Edit" permission can define reports and store them using the Metrics report component types. For a domain-level Subversion report, you can measure activity across the entire domain or in select projects in the domain. For Subversion reports, only projects in a domain that use Subversion for their version control system are listed as available for reporting.

The software metrics literature often describes complex models purporting to help predict various properties of software products and processes by measuring other properties. It

also contains lots of controversy about the value of the models and their predictions. But even if we remain theoretically skeptical of some of the models, we shouldn't throw away the corresponding measurements. The very process of collecting these measurements leads (as long as we confine ourselves to measurements that are meaningful, at least by some informal criteria) to a better organization of the software process and a better understanding of what we are doing. This idea explains the attraction and usefulness of process guidelines such as the Software Engineering Institute's Capability Maturity Model, which encourage organizations to monitor their processes and make them repeatable, in part through measurement [4].

2. METRICS MEASUREMENT

Metrics are units of measurement. The term "metrics" is also frequently used to mean a set of specific measurements taken on a particular item or process. Software engineering metrics are units of measurement that are used to characterize:

- software engineering products, e.g., designs, source code, and test cases,
- software engineering processes, e.g., the activities of analysis, designing, and coding, and
- software engineering people, e.g., the efficiency of an individual tester, or the productivity of an individual designer.

If used properly, software engineering metrics can allow us to:

- quantitatively define success and failure, and/or the degree of success or failure, for a product, a process, or a person,
- identify and quantify improvement, lack of improvement, or degradation in our products, processes, and people,
- make meaningful and useful managerial and technical decisions,
- identify trends, and
- make quantified and meaningful estimates.

Object-oriented software engineering metrics are units of measurement that are used to characterize:

- object-oriented software engineering products, e.g., designs, source code, and test cases,
- object-oriented software engineering processes, e.g., the activities of analysis, designing, and coding, and
- object-oriented software engineering people, e.g., the efficiency of an individual tester, or the productivity of an individual designer.

3. WHY ARE OBJECT-ORIENTED SOFTWARE ENGINEERING METRICS DIFFERENT?

OOSE metrics are different because of:

- localization,
- encapsulation,
- information hiding,
- inheritance, and
- object abstraction techniques.

3.1 Localization is the process of placing items in close physical proximity to each other:

- Functional decomposition processes localize information around functions.
- Data-driven approaches localize information around data.
- Object-oriented approaches localize information around objects.

In object-oriented software, however, localization is based on objects. This means:

- Although we may speak of the functionality provided by an object, at least some of our metrics identification and gathering effort (and possibly a great deal of the effort) must recognize the "object" as the basic unit of software.
- Within systems of objects, the localization between functionality and objects is not a one-to-one relationship.

For example, one function may involve several objects, and one object may provide many functions.

3.2 Encapsulation is the packaging (or binding together) of a collection of items:

- Low-level examples of encapsulation include records and arrays.
- Subprograms (e.g., procedures, functions, subroutines, and paragraphs) are mid-level mechanisms for encapsulation.
- In object-oriented (and object-based) programming languages, there are still larger encapsulating mechanisms, e.g., C++'s classes, Ada's packages, and Modula 3's modules.

3.2.1 Objects Encapsulate

- knowledge of state, whether statically maintained, calculated upon demand, or otherwise,
- advertised capabilities (sometimes called operations, method interfaces, method selectors, or method interfaces), and the corresponding algorithms used to accomplish these capabilities (often referred to simply as methods),
- [in the case of composite objects] other objects,
- [optionally] exceptions,
- [optionally] constants, and
- [Most importantly] concepts.

In many object-oriented programming languages, encapsulation of objects (e.g., classes and their instances) is syntactically and semantically supported by the language. In others, the concept of encapsulation is supported conceptually, but not physically [10,11].

Encapsulation has two major impacts on metrics:

- The basic unit will no longer be the subprogram, but rather the object, and
- We will have to modify our thinking on characterizing and estimating systems.

3.2.2 Information hiding is the suppression (or hiding) of details.

- The general idea is that we show only that information which is necessary to accomplish our immediate goals.
- There are degrees of information hiding, ranging from partially restricted visibility to total invisibility.
- Encapsulation and information hiding are not the same thing, e.g., an item can be encapsulated but may still be totally visible.

Information hiding plays a direct role in such metrics as object coupling and the degree of information hiding

3.3 Inheritance is a mechanism whereby one object acquires characteristics from one, or more, other objects.

- Some object oriented languages support only single inheritance, i.e., an object may acquire characteristics directly from only one other object.
- Some object-oriented languages support multiple inheritance, i.e. an object may acquire characteristics directly from two, or more, different objects.
- The types of characteristics which may be inherited, and the specific semantics of inheritance vary from language to language.

Many object-oriented software engineering metrics are based on inheritance, e.g.:

- number of children (number of immediate specializations),
- number of parents (number of immediate generalizations), and
- class hierarchy nesting level (depth of a class in an inheritance hierarchy).

3.4 Abstraction is a mechanism for focusing on the important (or essential) details of a concept or item, while ignoring the inessential details.

- Abstraction is a relative concept. As we move to higher levels of abstraction we ignore more and more details, i.e., we provide a more general view of a concept or item. As we move to lower levels of abstraction, we introduce more details, i.e., we provide a more specific view of a concept or item.
- There are different types of abstraction, e.g., functional, data, process, and object abstraction.
- In object abstraction, we treat objects as high-level entities (i.e., as black boxes).

There are three commonly used (and different) views on the definition for "class,":

- A class is a pattern, template, or a blueprint for a category of structurally identical items. The items created using the class are called instances. This is often referred to as the "class as a `cookie cutter'" view.
- A class is a thing that consists of both a pattern and a mechanism for creating items based on that pattern. This is the "class as an `instance factory'" view. Instances are the individual items that are "manufactured" (created) by using the class's creation mechanism.
- A class is the set of all items created using a specific pattern, i.e., the class is the set of all instances of that pattern.

A **metaclass** is a class whose instances are themselves classes. Some object-oriented programming languages directly support user-defined metaclasses. In effect, metaclasses may be viewed as classes for classes, i.e., to create an instance, we supply some specific parameters to the metaclass, and these are used to create a class. A metaclass is an abstraction of its instances.

A **parameterized class** is a class some or all of whose elements may be parameterized. New (directly usable) classes may be generated by instantiating a parameterized class with its required parameters. Templates in C++ and generic classes in Eiffel are examples of parameterized classes. Some people differentiate metaclasses and parameterized classes by noting that metaclasses (usually) have run-time behavior, whereas parameterized classes (usually) do not have run-time behavior.

Several object-oriented software engineering metrics are related to the class-instance relationship, e.g.:

- number of instances per class per application,
- number or parameterized classes per application, and
- ratio of parameterized classes to non-parameterized classes.

3.5 Coupling in software has been linked with maintainability and existing metrics are used as predictors of external software quality attributes such as fault-proneness, impact analysis, ripple effects of changes, changeability, etc. Many coupling measures for object-oriented (OO) software have been proposed, each of them capturing specific dimensions of coupling. This paper presents a new set of coupling measures for OO systems – named conceptual coupling, based on the semantic information obtained from the source code, encoded in identifiers and comments. A case study on open source software systems is performed to compare the new measures with existing structural coupling measures. The case study shows that the conceptual coupling captures new dimensions of coupling, which are not captured by existing coupling measures; hence it can be used to complement the existing metrics.

3.5.1 Types Of Coupling

Coupling can be "low" (also "loose" and "weak") or "high" (also "tight" and "strong"). Some types of coupling, in order of highest to lowest coupling, are as follows:

Content Coupling (High)

Content coupling is when one module modifies or relies on the internal workings of another module (e.g., accessing local data of another module). Therefore changing the

way the second module produces data (location, type, and timing) will lead to changing the dependent module.

Common Coupling

Common coupling is when two modules share the same global data (e.g., a global variable).Changing the shared resource implies changing all the modules using it.

External Coupling

External coupling occurs when two modules share an externally imposed data format, communication protocol, or device interface. This is basically related to the communication to external tools and devices.

Control Coupling

Control coupling is one module controlling the flow of another, by passing it information on what to do (e.g., passing a what-to-do flag).

Stamp Coupling (Data-Structured Coupling)

Stamp coupling is when modules share a composite data structure and use only a part of it, possibly a different part (e.g., passing a whole record to a function that only needs one field of it). This may lead to changing the way a module reads a record because a field that the module doesn't need has been modified.

Data Coupling

Data coupling is when modules share data through, for example, parameters. Each datum is an elementary piece, and these are the only data shared (e.g., passing an integer to a function that computes a square root).

Message Coupling (Low)

This is the loosest type of coupling. It can be achieved by state decentralization (as in objects) and component communication is done via parameters or message passing.

No Coupling

Modules do not communicate at all with one another.

3.5.2 Object-Oriented Programming

Subclass Coupling Describes the relationship between a child and its parent. The child is connected to its parent, but the parent isn't connected to the child.

Temporal Coupling When two actions are bundled together into one module just because they happen to occur at the same time.

3.5.3 Coupling Between Objects (Cbo)

1) coupling = class x is coupled to class y iff x uses y's methods or instance variables (includes inheritance related coupling)
2) CBO for a class is a count of the number of other classes to which it is coupled
3) High coupling between classes means modules depend on each other too much
4) Independent classes are easier to reuse and extend
5) High coupling decreases understandability and increases complexity
6) High coupling makes maintenance more difficult since changes in a class might propagate to other parts of software
7) Coupling should be kept low, but some coupling is necessary for a functional system

3.5.4 Coupling Versus Cohesion

Coupling and Cohesion are the two terms which very frequently occur together. Together they talk about the quality a module should have. Coupling talks about the inter dependencies between the various modules while cohesion describes how related functions within a module are. Low cohesion implies that module performs tasks which are not very related to each other and hence can create problems as the module becomes large.

Advantages

Whether loosely or tightly coupled, a system's performance is often reduced by message and parameter creation, transmission, translation and interpretation overhead. They will improve four type of performance.

3.6 Complexity Metrics

Complexity is everywhere in the software life cycle: requirements, analysis, design, and of course, implementation. is usually an undesired property of software because complexity makes software harder to read and understand, and therefore harder to change; also, it is believed to be one cause of the presence of defects. In a use net debate surrounding Intelligent Design, the issue of measuring complexity kept coming up. Are there any good objective metrics for "complexity"? "Number of parts" is limited because it could be a result of repetition, chaos, or waste. "Number of different parts" can be fudged and requires meaningful difference metrics.

All the artifacts produced in a software project, source code is the easiest option to measure complexity. However, several decades of software research have failed to produce a consensus about what metrics best reflect the complexity of a given piece of code. It's hard even to compare two pieces of code written in different programming languages and say which code is more complex. Because of this lack of resolution, a myriad of possible metrics are currently offered to measure the complexity of a program. What does the research say are the best metrics for each particular case? Are all these metrics any better than very simple source code metrics, such as lines of codes? We take advantage of the huge amount of open source software available to study the relationships between different size and complexity metrics. To avoid suffocating in the myriads of attributes and metrics, we focus only on one programming language: C, a "classic" in software development that remains one of the most popular programming languages.

3.6.1 Types Of Complexity Metrics

- Cyclomatic complexity (or conditional complexity)
- hierarchical complexity.
- Computational complexity.
- Kolmogorov complexity.
- Non-hierarchical object complexity.
- Non-hierarchical process complexity.
- hierarchical object complexity.

Cyclomatic Complexity	Risk Complexity
1-10	a simple program, without much risk
11-20	more complex, moderate risk
21-50	complex, high risk
51+	untestable, very high risk

Table 1. Standard Values of Cyclomatic Complexity

5. CONCLUSION AND FUTURE SCOPE

The above results can be used in order to determine when and how each of the above metrics can be used according to quality characteristics a practitioner wants to emphasize. Make sure the software quality metrics and indicators they employ include a clear definition of component parts are accurate and readily collectible, and span the development spectrum and functional activities. Survey data indicates that most organizations are on the right track to making use of metrics in software projects. For organizations which do not reflect "best practices", and would like to enhance their metrics capabilities, the following recommendations are suggested to Measure the "best practices" list of metrics more consistently across all projects. Focus on "easy to implement" metrics that are understood by both management and software developers, and provide demonstrated insight into software project activities. We have described in detail six metrics, chosen among the ones most widely known and used. They are relative to different phases of software development. In the requirements phase, we can use Function Points to measure the functionality starting from the user requirements. In the high-level design phase, the suite of metrics can be used: we have concept of measures for cohesion and coupling, which are important attributes of design.

A number of object oriented metrics have been proposed in the literature for measuring the design attributes such as

inheritance, polymorphism, message passing, complexity, Hiding Factor, coupling, cohesion, reusability etc,. In this paper, A metrics program that is based on the goals of an organization will help communicate, measure progress towards, and eventually attain those goals. People will work to accomplish what they believe to be important. Well-designed metrics with documented objectives can help an organization obtain the information it needs to continue to improve its software products, processes, and services while maintaining a focus on what is important. A practical, systematic, start-to-finish method of selecting, designing, and implementing software metrics is a valuable aid. The number of methods and the complexity of methods involved is a predictor of how much time and effort is required to develop and maintain the class.. While in the past the focus in research was on inventing new metrics, now the focus is more on measurement theory, in particular on the definition of new validation frameworks or of new set of axioms.

6. REFERENCES

[1] Kaur Amandeep, Singh Satwinder, K. Kahl. "Evaluation and Metrication of Object Oriented System", International Multi Conference of Engineers and Cmputer Scientists, 2009 vol. 1.

[2] J. Alghamdi, R. Rufai, and S. Khan. Oometer: A software quality assurance tool. Software Maintenance and Reengineering, 2009. CSMR 2009. 9th European Conference on, pages 190{191, 21-23}, March 2010.

[3] S. Conte, H. Dunsmore, V. Shen, Software Engineering Metrics and Models, Benjamin/Cummings, Menlo Park, CA.

[4] S. Chidamber, C. Kemerer, A Metrics Suite for Object Oriented Design, IEEE Trans. Software Eng., 20/6), 2000, pp. 263-265.

[5] A. Albrecht and J. Gaffney: Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation; in IEEE Trans. Software Eng., 9(6), 2008, pp. 639-648.

[6] B. Bohem, Software Engineering Economics, Prentice Hall, Englewood Cliffs, 1981 [Briand et al 94] L. Briand, S. Morasca, V. Basili, Defining and Validating High- Level Design Metrics, Tech. Rep. CS TR-3301, University of Maryland, 2009.

[7] S. Morasca, Software Measurement: State of the Art and Related Issues, slides from the School of the Italian Group of Informatics Engineering, Rovereto, Italy, September 2008.

[8] H. Bsar, M. Bauer, O. Ciupke, S. Demeyer, S. Ducasse, M. Lanza, R. Marinescu, R. Nebbe, O. Nierstrasz, M. Przybilski, T. Richner, M. Rieger, C. Riva, A. Sassen, B. Schulz, P. Steyaert, S. Tichelaar, and J. Weisbrod. The FAMOOS Object-Oriented Reengineering Handbook, Oct. 2006.