Metrics and Heuristics in Software Engineering

Rakesh Kumar[1], Deepali Gupta[2]

GJCST Classification
I.2.8, D.2.8, D.4.8

Abstract- **Heuristics plays an important role in software development and are widely used to provide a link between design principles and software measurement. They offer insightful information based upon experience that is known to work in practice. Heuristics are not meant to be exact; in fact, they derive their benefits from this imprecision by providing an informal guide to good and bad practices. They provide a means by which knowledge and experience can be delivered from the expert to the novice. The paper is set out to bring techniques for building maintainable object oriented software closer to the developer in the form of design heuristics. Heuristics document common design problems that developers encounter during software development. Some heuristics in software engineering can be expressed in high-level abstract terms while others are more specific. The heuristic catalogue provides a comprehensive reference point for both novice and expert developers to apply well-documented techniques for building maintainable software.**

Keywords- Heuristics, OO Design, Metrics, Software Engineering and software metrics.

I. Introduction

Software engineering is the systematic collection of decades of programming experience together with the innovations made by researchers towards developing high quality software in a cost effective manner. In other words, software engineering is a systematic & cost effective approach to develop software. The basic objective of software engineering is to develop methods for developing software that can scale up & can be used to consistently develop high quality software at low cost. The improvements to software engineering over the last four decades have indeed been remarkable. Notable changes have occurred in software from error correction to error prevention. Now there are several development activities apart from coding like design, testing & maintenance. A lot of effort is now paid to requirements specification. Periodic reviews, testing, documentation, software project management are carried out during all stages of software development process. Software engineering has traditionally been an expensive and time-intensive process. Object-oriented analysis and design is the principal industry-proven methodology that answers the call for a more cost-effective, faster way to develop software and systems [6]. The basic design behind OOD is fundamentally different from the paradigm of function oriented design. In Object Oriented Design, data and operations are considered together, where as in case of function oriented approach the two are kept separate.Hence, OO design is one where the final system is represented in terms of object classes, relationships between objects and relationships between classes. An object is an instance of a class & has state, behaviour and identity. Key features of OOD are inheritance, information hiding, encapsulation, polymorphism, low coupling, high cohesion and modularity.A key element in software process is measurement. Software engineering, by is nature, is a quantitative discipline. Within the software engineering context a measure provides a quantitative indication of the extent, amount, dimension, capacity or size of some attribute of a product or process. Measurement is the act of determining a measure. The IEEE standard glossary [2] defines metric as "a quantitative measure of the degree to which a system, component, or process processes a given attribute". Software metrics provide a quantitative way to assess the quality of internal product attributes, there by enabling a software engineer to assess quality before the product is built. Metrics provide the insight necessary to create effective analysis and design models, solid code, and thorough tests.

There are set of attributes [1] defined that should be encompassed by effective software metrics. These are: -

1. **Simple and computable: -** It should be relatively easy to learn how to derive the metric and its computation should not demand inordinate effort and time.
2. **Empirically and intuitively persuasive: -** The metric should satisfy engineer's intuitive notions about the product attribute under consideration.
3. **Consistent and objective: -** The metric should always yield results that are unambiguous.
4. **Consistent in the use of units and dimensions: -** The mathematical computation used should be consistent in the use of units and dimensions.
5. **Programming language independent: -** Metric should be based on analysis model, design model, or the structure of the program itself.
6. **An effective mechanism for high quality feedback: -** Metric should lead to higher quality end product.

II. Objectives

The aim of Object Oriented (OO) Metrics is to predict the quality of the object oriented software products. Various attributes, which determine the quality of the software, include maintainability, defect density, fault proneness, normalized rework, understandability, reusability etc.Object Oriented (OO) Metrics are required because in OO code, complexity lies in interaction between objects, a large

About[1] - Reader, Department of Computer Science and Applications, Kurukshetra University, Kurukshetra, India. (Telephone: 09896336145 email: rsagwal@rediffmail.com)
About[2]- Dean Academics, Geeta Institute of Management and Technology, Kanipla, Kurukshetra, India. (Telephone: 09215710291 email: deepali_gupta2000@yahoo.com)

portion of code is declarative, OO models real life objects: classes, objects, inheritance, encapsulation, message passing. Software metrics helps to improve software process and its product. The use of object oriented software development techniques introduces new elements to software complexity both in software development process and in the final product [8].Our research investigates ways to help designers with the task of understanding, evaluating and improving their software products. While we view the art of design and the judgments of how to apply heuristics as beyond the reach of current technology, we argue that tools can provide valuable information to assist the designer with these judgments. OOD heuristics encapsulate software problems and their solutions in supporting an informal approach to design evaluation. Software design and development involves a range of practices with varying levels of formality: examples include formal methods, coding styles, design patterns and test-driven development. The common goal is the production of high quality software.However, quality is a concept that can not be measured directly. In order to measure and understand quality, it is necessary to relate it to measurable quantities. The field of software metrics deals with the identification of meaningful quantitative measures of specific properties of software.Heuristics enable a softer model to be constructed in order to obtain a more holistic and subjective, view of quality. This potentially places a greater burden on the developers who must interpret this view since it consists of potentially conflicting indicators with varying degrees of precision and relevance. Heuristics may occur as individual pieces of developers' or may be presented as a suite covering multiple aspects of software development.

III. HEURISTICS IN SOFTWARE ENGINEERING

Everyday in our life, we do make the use of heuristic to solve the problem and software engineering is not an exception. In the past also we are seeing the use of some metric based heuristics in design and development of the software. For example, if the number of parameters in a function is more than five gives impression that module may not be having function cohesion. The heuristics are not written as hard and fast rules; they are meant to serve as warning mechanisms which allow the flexibility of ignoring the heuristic as necessary. Use of heuristics in modern OO software engineering has also been observed. Design is a difficult task because it involves finding compromises between conflicting pressures, cost and reliability. Designers must find ways to provide specific capabilities required by stakeholders, while attaining sufficient quality in emergent properties such as usability, efficiency, and flexibility. Software designers aim to satisfy the expectations of stakeholders by meeting functional and non-functional requirements.But in order to make this possible, they must first address the needs of the software developers themselves. Keeping the complexity of the design in check is foremost among these. Object-orientation (OO) allows software to be structured in a way that helps to manage complexity and change. However, as software reuse practitioners have discovered, realizing the benefits of OO is not straightforward. Competence with the mechanisms of classes and objects, attributes and methods, inheritance and polymorphism is far from sufficient to ensure successful designs.Metric-based heuristic framework is used to detect and locate object-oriented design flaws from the source code [7]. It is accomplished by evaluating design quality of an object-oriented system through quantifying deviations from good design heuristics and principles.Classes and Objects are the Building Blocks of the Object-Oriented Paradigm. Some of the heuristics proposed by Riel [3] are listed in Table 1 as follows:

Sno.	Heuristics
1	All data should be hidden within its class.
2	Users of a class must be dependent on its public interface, but a class should not be dependent on its users.
3	Minimize the number of messages in the protocol of a class.
4	Implement a minimal public interface that all classes understand.
5	Do not put implementation details such as common-code private functions into the public interface of a class.
6	Do not clutter the public interface of a class with things that users of that class are not able to use or are not interested in using.
7	A class should only use operations in the public interface of another class or has nothing to do with that class.
8	A class should capture one and only one key abstraction.
9	Keep related data and behavior in one place.
10	Spin off non related information into another class (i.e., non communicating behavior).
11	Be sure the abstractions that you model are classes and not simply the roles objects play.

Table 1. List of heuristics proposed by Riel.

Riel [3] documents 61 "golden rules" for OO design, while Fowler and Beck describe 22 code smells [4]. Smells evokes a subjective, subtle process of perceiving something about a design. Beck and Fowler noted that code smells do not lend themselves to automatic quantification [4]. The designer must form an impression of the net product of many factors at work in the design. This requires judgment and insight beyond the capabilities of simple automata. A notable characteristic of design patterns is that they often break rules. For example, the Composite pattern advocates the use of methods that are overridden to do nothing, contrary to a common maxim, expressed by Riel's heuristic as "It should be illegal for a derived class to override a base class method with a NOP method, i.e. a method which does nothing." However, the Gang of Four chose to break this rule deliberately, in their words preferring transparency over safety. Many similar examples of conflicting forces can be found. Some conflicts are so pervasive that they apply to nearly all design situations. Separation of concerns, for example, encourages decoupling portions of a design, while another heuristics, “Keep related data and behavior in one place” often suggests the opposite. Even within an organized set of heuristics, conflicts occur. One heuristic says “Theoretically, inheritance hierarchies should be deep, i.e. the deeper the better”, while another adds the qualification that “In practice, inheritance hierarchies should be no deeper than an average person can keep in his or her short-term memory. A popular value for this depth is six”. Heuristics are a valuable tool for identifying design forces (whether conflicting or not) and evaluating design quality, but their application is not straightforward for many reasons [5], such as:

Lack of consensus on which heuristics should be adopted: Some conflicting heuristics usefully illuminate matters of concern to the designer. Other conflicts, however, reflect differing design philosophies, and a particular designer is likely to be interested only in one side of the debate. Many of the tenets arising from software reuse culture, for example, are in opposition to more recent refactoring and agile methods approaches. The open/closed principle, for example, encourages anticipation of future needs by making the design open for extension (reusable), but without requiring modification of existing code; refactoring culture discourages anticipation of future needs and prefers modifying existing code when necessary. This cultural difference might show up in unexpected ways, such as a stronger preference for small methods in the reuse culture, so that methods constitute small overridable units.

Nebulous definitions: One heuristic, for example, says “A class should capture one and only one key abstraction”, but rigorously specifying the meaning of “key abstraction” is problematic. Similarly, heuristic “Model the real world whenever possible”, is only as firm as our grip on reality.

Subjectivity and calibration: Code smells require the designer to judge when some intangible threshold has been crossed. The “large class smell”, “lazy class smell” and “long method smell” are obvious examples where different standards might apply. The relative importance of conflicting heuristics is also dependent on the value system of the designer. If breaking up a large class produces a lazy class, is the result better?

Interpretation in different contexts: Many heuristics are expressed abstractly, in order to apply to any OO design. It may be necessary, however, to adapt a heuristic to local conditions. For example, when deciding if an inheritance hierarchy is too deep, should the root class be counted in programming languages that enforces a single root? Or, in an organisation that has adopted a refactoring approach to software development, how much emphasis should be placed on a heuristic motivated by software reuse, such as heuristic “All base classes should be abstract classes”?

Diverse levels of abstraction. Some heuristics can be interpreted at different levels. For example, “All data should be hidden within its class”, might be viewed as a syntactic restriction make attributes private or as a semantic one, which might also discourage the use of getters. A “long method smell” could be detected at a lexical level by counting lines of code, at a syntactic level by counting statements and expressions, at a language semantic level by counting method invocations, collaborators, etc, or at a problem-domain semantic level by gauging the conceptual size of the method.

Information overload: Heuristics are intended to help software engineers manage the complexity of software, but injudicious application of heuristics could compound the problem.

Acquiring relevant data and relating it to heuristics: Many heuristics require substantial data gathering. Heuristic “Minimize fan-out in a class” and another “Most of the methods defined on a class should be using most of the data members most of the time” are examples. Additionally, the correspondence between available information and heuristics is not always clear.These issues, and the inherent fuzziness of heuristics, make automated support of heuristics difficult. In consequence, designers usually must gauge the quality of their products without assistance from tools. The designer builds a mental model of the software, and evaluates, according to a subjective, and perhaps even subconscious, process that is likely to be informed by heuristics, but may explicitly apply few.

IV. Conclusion

One of the mature engineering disciplines is the ability of its practitioners to quantify the quality of a product that is the ability to establish metrics. The use of metrics can be a valuable aid in understanding the effect of actions that are implemented for improving the software development process. The metrics provide visibility and control for the complex software development process, and therefore they are valuable for providing guidance on improving the software development process, and for meeting organizational goals to improve software quality and productivity. This causes new requirements for software metrics. While some of the traditional metrics can be used, new metrics must be introduced. The introduction of object-oriented (OO) methods to software development has changed the process of building and managing software in a profound way.Quality of software is increasingly important and testing related issues are becoming crucial for software. Although there is diversity in the definition of software quality, it is widely accepted that a project with many defects lacks quality. Methodologies and techniques for predicting the testing effort, monitoring process costs, and measuring results can help in increasing efficiency of software testing. Prediction of fault-prone modules supports software quality engineering through improved scheduling and project control. It is a key step towards steering the software testing and improving the effectiveness of the whole process In order to measure and understand quality, it is necessary to relate it to measurable quantities. Heuristics provide a link between sets of abstract design principles and quantitative software metrics. They are an important part of software design and are becoming more widely used. Effective visualization of heuristics includes quantitative, qualitative and ambient aspects. Visualisation of heuristics provides many challenges. Heuristics are likely to be studied both individually and in comparison with others.The researchers are not primarily concerned with the relevance or validity of individual heuristics: the main focus is on their evaluation and interpretation. Our work is intended to provide the basis for an exploratory framework in which heuristics may be postulated, explored and managed.

V. References

1) Lem O. *Ejiogu*, *Software engineering with formal metrics*, QED Information Sciences, Inc., Wellesley, MA, 1991.
2) *IEEE Standards Collection: Software Engineering,* IEEE Standard 610.12-1990, IEEE, 1993.
3) Riel A., *Object-Oriented Design Heuristics,* Addison-Wesley, 1996.
4) Fowler, M., *Refactoring: Improving the Design of Existing Code*, Addison - Wesley, 1999.
5) Churcher, N., et al., "Supporting OO Design Heuristics", *Proceedings of the 2007 Australian Software Engineering Conference,* IEEE Computer Society, 2007, pp. 101-110.
6) Berard, E.V, "Essays on Object-Oriented Software Engineering", vol. 1, Addison Wesley, 1993.
7) Salehie, M., Li, S., Tahvildari, L., "A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws",*Proceedings of 14th IEEE International Conference on Program Comprehension,* IEEE Computer Society, 2006, pp.159-168.
8) Brooks, I., "Object-Oriented Metrics Collection and Evaluation with a Software Process," Proc. OOPSLA '93 Workshop Processes and Metrics for Object-Oriented Software Development, Washington, D.C., 1993.