ISSN:2229-6093

An Heuristic Approach To Object Oriented Paradigm

Deepali Gupta [*] and Rakesh Kumar [**]

[*] Dean Academics, Geeta Institute of Management and Technology, Kanipla, Kurukshetra, India

[**] Reader, Kurukshetra University, Kurukshetra, India

deepali_gupta2000@yahoo.com

***Abstract*— Quality of software is increasingly important and testing related issues are becoming crucial for software. 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. The aim of object oriented software metrics is to predict quality and improve productivity of the software products. Object-orientation (OO) allows software to be structured in a way that helps to manage complexity and change. This paper shows role of heuristics in object oriented software engineering and how object oriented paradigm differentiate from conventional function oriented paradigm.**

Keywords: Software metrics, Function-oriented paradigm, Object-oriented paradigm, Software quality and Heuristics.

I. Introduction

Engineering is the analysis, design, construction, verification, and management of technical (or social) entities. Like all other types of engineering, software engineering is not just producing products but means producing products in most efficient and cost effective way. Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.

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 [2]. With the increasing use of object-oriented methods in new software development, there is a growing need to both document and improve current practices in object-oriented design and development. Object-oriented paradigm exhibits different characteristics from the procedural paradigm. So, different software metrics have to be used.

Three important concepts differentiate the OO approach from conventional software engineering.

- Encapsulation packages data & the operations that manipulate the data into a single named object.
- Inheritance enables the attributes and operations of a class to be inherited by all subclasses and the objects that are instantiated from them.
- Polymorphism enables a number of different operations to have the same name, reducing the number of lines of code required to implement a system and facilitating changes whenever made.

In object-oriented viewpoint, the problem domain is characterized as a set of objects that have specific attributes and behaviors. The objects are manipulated with a collection of functions called methods, operations, or services and communicate with one another through a messaging protocol.

II. Use of Quantitative approaches

Measurement can be used throughout a software project to assist in estimation, quality control, productivity assessment, project control and to assist in tactical decision making as a project proceeds. There are four reasons for measuring software processes, products, and resources: to characterize, to evaluate, to predict, or to improve [3]. Metrics is defined as "The continuous application of measurement based techniques to the software development process and its products to supply meaningful and timely management information, together with the use of those techniques to improve that process and its products". The IEEE standard glossary [1] defines metric as "a quantitative measure of the degree to which a system, component, or process processes a given attribute". Software metrics is all about measurement and these are applicable to all the phases of software development life cycle from initiation to maintenance.

Metrics are grouped into three main categories:

- Product metrics measure the software product properties, such as its documentation, design and performance, regardless of its development stage. Product metrics are indicator of external software attributes such as cyclomatic complexity for testability and coupling factor for maintainability.
- Process metrics emphasize on the software development process, such as development time, methodology used and quality assurance techniques.
- Resource metrics emphasize on the human, hardware and software resources such as developer skill level, hardware reliability, software component quality.

To achieve both the quality and productivity objectives, it is always recommended to go for the software reuse that not only saves the time taken to develop the product from scratch but also delivers the almost error free code. Prediction models based on software metrics, can estimate number of faults in software modules [4].

III. Objectives

Science begins with quantification; you cannot do physics without a notion of length and time; you cannot do thermodynamics until you measure temperature. All engineering disciplines have metrics (such as metrics for weight, density, wavelength, pressure and temperature) to quantify various characteristics of their products. The most important question you can ask is "how big is the program?" Without defining what big it means, it is obvious that makes no sense to say, "This program will need more testing than that program" unless we know how big they are relative to one another. 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. We need them because in OO code, complexity lies in interaction between objects, a large portion of code is declarative, OO models real life objects: classes, objects, inheritance, encapsulation, message passing. Heuristic also provide valuable information to assist the designer to design and develop better software and focuses on improving quality of software products.

IV. Heuristics in Software Engineering

Heuristics are used everyday in our daily life to solve problem and software engineering is not an exception. Some metric based heuristics are used in design and development of the software in the past also. For example, if the number of parameters in a function is more than five gives impression that module may not be having function cohesion.

Use of heuristics in modern OO software engineering has also been observed [5]. Heuristics plays an important role in software development and is 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.

Design is a difficult task because it involves finding compromises between conflicting pressure, cost and reliability and many of these pressures ultimately arise from human concerns, with all that implies in complexity, diversity and changeability. 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 designer's aim is 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 the foremost among these. Object-orientation (OO) allows software to be structured in a better way and 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 object oriented classes and objects, attributes and methods, inheritance and polymorphism is far from sufficient to ensure successful designs. Over some decades, software engineers have developed a number of approaches and principles to elevate design considerations above programming language mechanisms. Concepts such as abstraction, separation of concerns, information hiding, cohesion and coupling provide guidance to designers. On top of these general principles, the OO design community has developed a rich doctrine of principles and practices to inform designers. This is supplemented by design patterns and idioms, which provide prototypical solutions to common problems. Some authors have collated parts of this complex web of concepts into sets of heuristics. Johnson and Foote [6] provide an early example, which describes design maxims intended to promote reuse. Riel [7] documents 61 'golden rules' for OO design, while Fowler and Beck describe 22 code smells [8]. Smells evokes a subjective, subtle process of perceiving something about a design. Beck and Fowler note that code smells do not lend themselves to automatic quantification [8]. 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.

V. object-oriented paradigm

The object-oriented paradigm uses the concepts of class and object as basic building blocks in the formation of a consistent model for the analysis, design, and implementation of applications. There are many things in the real world that we are capable of using without knowing anything about their implementation: refrigerators, cars, photocopy machines, and computers, just to name a few. The reason they are easy to use without knowledge of their implementation is that they are designed to be used via a well-defined public interface. This interface is heavily dependent on, but hides from its users, the implementation of the device.

All implementation constructs in your system should be hidden from their users behind a well-defined, consistent public interface. Users of the construct need to know about the public interface but are never allowed to see its implementation. This allows the implementer to change the implementation whenever he or she desires, so long as the public interface remains the same. Class is a concept that captures the notion of data and behavior in one package. The relationship between the notion of class and object is instantiation relationship. In addition to fixed data and behavioral descriptions, objects have local state (i.e., a snapshot) at runtime of the dynamic values of an object's data descriptions. The collection of all possible states of a class's objects, along with the legal transitions from one state to another, is called the dynamic semantics of the class. Dynamic semantics allow an object to respond differently to the same message sent at two different times in the life of the object. Classes with interesting dynamic semantics include those classes having a finite number of states, with well-defined transitions from one state to another. Abstract classes are the classes that do not know how to instantiate objects where as classes that do know how to instantiate objects are called concrete classes.

VI. Function-oriented paradigm vs. object-oriented paradigm

In the function-oriented world, it is easy to find data dependencies simply by examining the implementation of functions and there does not an explicit relationship between data and functionality.

While function-oriented software development is involved with functional decomposition through a very centralized control mechanism, the object-oriented paradigm focuses more on the decomposition of data with its corresponding functionality in a decentralized setting.

It is this decentralization of software that gives the object-oriented paradigm its ability to control essential complexity.

Many developers correctly claim that the function-oriented paradigm focuses only on the functionality of a system and typically ignores the data until it is required. They then claim that the object-oriented paradigm focuses exclusively on the data, ignoring the functionality of the system until it is required. This is not possible, because the behavior of a system often drives the decomposition of data. It is preferable to think of the object-oriented paradigm as keeping data in the front of a developer's mind while keeping functionality in the back of his or her mind. The result of this process is the decomposition of our system into a collection of decentralized clumps of data with well-defined public interfaces. The only dependencies of functionality on data are that the operations of the well-defined public interfaces are dependent on their associated data (i.e., implementation).

There are two very distinct areas where the object-oriented paradigm can drive design in a dangerous direction. The first is a problem of poorly distributed system intelligence, while the second is the creation of too many classes for the size of the design problem. We refer to these pitfalls as the god class problem and the proliferation of class problem. The god class problem manifests itself in two forms, the behavioral form and the data form. The proliferation of classes' problem is produced by a number of factors.

The behavioral form of the god class problem is caused by a common error among action-oriented developers in the process of moving to the object-oriented paradigm. These developers attempt to capture the central control mechanism so prevalent in the function-oriented paradigm within their object-oriented design. The result is the creation of a god object that performs most of the work, leaving minor details to a collection of trivial classes. There are a number of heuristics that work together cooperatively toward the avoidance of these classes.

Violations of these heuristics imply the creation of a behavioral god object. Another symptom of this problem is the creation of many get and set functions in the public interfaces of your applications' classes. These get and set functions are called accessor methods. Such functions simply that some larger class is getting a great deal of data from other classes, performing computations on this data, and then setting the states of many objects to reflect the updated information. These classes are examples of behavioral god classes. Their design also violates the heuristic stating that related data and behavior should be kept in one place.

Some of the heuristics proposed by Riel [7] are listed in Table I as follows:

TABLE I.
SUMMARY OF RIEL'S HEURISTICS

S.No.	Function-Oriented Vs. Object-Oriented Paradigm
1	Distribute system intelligence horizontally as uniformly as possible, i.e. the top level classes in a design should share the work uniformly.
2	Do not create god classes/objects in your system. Be very suspicious of an abstraction whose name contains Driver, Manager, System, or Subsystem.
3	Beware of classes that have many accessor methods defined in their public interface; many of them imply that related data and behavior are not being kept in one place.
4	Beware of classes which have too much non-communicating behavior.
5	The model should never be dependent on the interface. The interface should be dependent on the model.
6	Model the real world whenever possible
7	Eliminate irrelevant classes from your design.
8	Eliminate classes that are outside the system.
9	Do not turn an operation into a class.
10	Agent classes are often placed in the analysis model of an application.

The following are some of the heuristics proposed by Riel [7] related to topologies of function-oriented vs. object-oriented applications:

Heuristic 1: Distribute system intelligence horizontally as uniformly as possible, i.e. the top level classes in a design should share the work uniformly.

Heuristic 2: Do not create god classes/objects in your system. Be very suspicious of an abstraction whose name contains Driver, Manager, System, or Subsystem.

Heuristic 3: Beware of classes that have many accessor methods defined in their public interface. Having many of them imply that related data and behavior are not being kept in one place.

Heuristic 4: Beware of classes which have too much non-communicating behavior, i.e. methods which operate on a proper subset of the data members of a class. God classes often exhibit lots of non-communicating behavior.

Accessor methods give away implementation details. Such methods are dangerous because they indicate poor encapsulation of related data and behavior. There are two reasonable explanations for the need for accessor methods. Either the class performing the gets and sets is implementing a policy between two or more classes, or it is in the interface portion of a system consisting of an object-oriented model and a user interface.

The second rationale for using accessor methods revolves around domains whose architecture involves an object-oriented model interacting with a user interface. By definition, user interfaces display the internals of a model, allow a user to update those internals, and put the internals back into the model. The heuristic here is that a model should be independent of its user interface. In order to accomplish this goal, the interface must be allowed to extract and replace details from the model via accessor methods. The use of the accessor methods should be restricted to classes within the interface portion of the code. It is important to note that the model classes of these types of systems rarely have any interesting behavior.

Heuristic 5: In applications which consist of an object-oriented model interacting with a user interface, the model should never be dependent on the interface. The interface should be dependent on the model.

Heuristic 6: Model the real world whenever possible.

This heuristic is often violated for reasons of system intelligence distribution, avoidance of god classes, and the keeping of related data and behavior in one place

Heuristic 7: Eliminate irrelevant classes from your design.

It is always better to eliminate those classes who have no meaningful behavior in the domain of your system. A class that has no meaningful behavior in the domain of a system is an irrelevant class. These classes have no operations besides set, get, and print type functions. The reason sets, gets, and prints are not counted as meaningful behavior is that all too often they operate solely on the descriptive attributes of a system.

Heuristic 8: Eliminate classes that are outside the system.

If a class is outside the system, it is irrelevant with respect to the given domain. It is difficult to detect classes that are outside the system. During successive iterations of design, it eventually becomes clear that some classes do not require any methods to be written for them. These are classes that are outside of the system. The hallmark of such classes is an abstraction that sends messages into the system domain but does not receive message sends from other classes in the domain. This heuristic is really a special case of the previous heuristic.

Heuristic 9: Do not turn an operation into a class.

Be suspicious of any class whose name is a verb or is derived from a verb, especially those that have only one piece of meaningful behavior (i.e., do not count sets, gets, and prints). Ask if that piece of meaningful behavior needs to be migrated to some existing or undiscovered class. Violations of this heuristic are a leading cause of proliferation of classes. Classes whose names are verbs, or are derived from verbs, are especially suspected. Newcomers to the object-oriented paradigm are especially prone to violations of this heuristic. It is important to note that not all classes whose names are verbs need to be eliminated.

Heuristic 10: Agent classes are often placed in the analysis model of an application.

Agent classes are irrelevant classes. They simply accept messages from other classes and resend them to the desired target. Agent classes are the classes whose sole purpose is to decouple two or more additional classes. These are characterized by delegating its methods to messages on other classes. During design time, many agents are found to be irrelevant and should be removed.

There are inter-dependencies that exist between heuristics and the complex nature of large object-oriented software systems. Heuristics document common design problems that developers encounter during software development. The heuristic catalogue provides a comprehensive reference point for both novice and expert developers to apply well-documented techniques for building maintainable software.

VII. Conclusion

Software quality is an important aspect in software development. 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 thereby managing project planning.

Object oriented heuristics encapsulate software problems and their solutions and provide a link between sets of software development principles and quantitative software metrics to produce high quality software.

References

[1] *IEEE Standards Collection: Software Engineering,* IEEE Standard 610.12-1990, IEEE, 1993.

[2] E.V Berard, "Essays on Object-Oriented Software Engineering", vol. 1, Addison Wesley, 1993.

[3] R.E. Park, W.B Goethert and V. Florac, "Goal Driven Software Measurement—A Guidebook", CMU/SEI-96-BH-002, Software Engineering Institute, Carnegie Mellon University, 1996.

[4] D. Giovanni, "Estimating Software Fault-Proneness for Tuning Testing Activities", *Proceedings of the 22nd International Conference on Software Engineering (ICSE2000*), Limerick, Ireland, 2000.

[5] N. Churcher, "Supporting OO Design Heuristics", *Proceedings of the 2007 Australian Software Engineering Conference,* IEEE Computer Society, pp. 101—110, 2007.

[6] R. Johnson and B. Foote, "Designing reusable classes", Journal of Object-Oriented Programming, vol 1, no.2 , pp. 22--35 , 1988.

[7] A. Riel, Object-Oriented Design Heuristics, Addison-Wesley, 1996.

[8] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison - Wesley, 1999.

[9] M. Salehie, S. Li, L. Tahvildari, "A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws", *Proceedings of 14th IEEE International Conference on Program Comprehension,* IEEE Computer Society, pp. 159—168, 2006.
.