Object Oriented Metrics for Prototype based Languages

Syed Ahsan, Faisal Hayat, Muhammad Afzal, Tauqir Ahmad, Khadim H. Asif, H.M. Shahzad Asif, Yasir Saleem

Department of Computer Science & Engineering
University of Engineering and Technology, Lahore (Pakistan)
Ahsancs.gmail.com

Abstract: Prototype (classless) based object oriented programming approach has several advantages for representing default knowledge and dynamically modifying concepts over traditional class based languages. Many modern languages like C#, JavaScript and others are in part or completely utilizing astounding features of prototypes. With this growing interest in adoption of prototypes a sheer need is emerging to redesign software metrics for prototype based languages. These paper highlights issues for prototype based software metrics for object oriented programming.

[Syed Ahsan, Faisal Hayat, Muhammad Afzal, Tauqir Ahmad, Khadim H. Asif, H.M. Shahzad Asif, Yasir Saleem. **Object Oriented Metrics for Prototype based Languages**. *Life Sci J* 2012;9(4):63-66] (ISSN:1097-8135). http://www.lifesciencesite.com. 10

Keywords: Software life cycle, software complexity, design metrics, prototype object modeling

1. Introduction

Espousing of prototype based features such as delegates in part or completely in modern languages like C#, JavaScript led towards consideration for prototype based approach for object oriented programming. When compared to class-based languages, prototype-based languages are conceptually simpler, and have many other characteristics that make them suitable especially to the development of evolving, exploratory and distributed software systems. Significant work has previously been done and many languages have already been designed implementing prototypes: SELF [1-4], KEVO [5-6], AGORA [7], GARNET [8-9], MOOSTRAP [10-11], CECIL [12], OMEGA [13] and NEWTON-SCRIPT [11] are such languages which represent another view of the object oriented programming. This approach is advantageous in sense that one does not need to rely so much on advance categorization and classification, instead the focus should be to make the concepts in the problem domain as tangible and intuitive as possible. In turn prototypes may give rise to a broad spectrum of interesting technical, conceptual and philosophical issues. Although interest in espousing prototypical features is increasing, yet no serious attempt has been made towards designing software metrics for prototype based languages till now. For class based approaches various software metrics related to quality assurance have been proposed in the past and are still being proposed but none of them explicitly discuss prototype based language.

Many metrics have been proposed explicitly in context of class based object-oriented programming such as class, coupling, cohesion, inheritance, information hiding and polymorphism. As the development of object-oriented software is rising, more and more metrics are required for object-oriented languages but the applicability of metrics developed previously are mostly limited to requirement, design and implementation phase instead of data representation and nature of problems. According to Moreau [10-11] traditional metrics are inappropriate for OO systems for several reasons. First, the assumptions relating program size and programmer productivity in structured systems do not apply directly to OO systems. Second, the traditional metrics do not address the structural aspects of OO systems. Third, the computation of the system's complexity as the sum of the complexity of the components is not appropriate for OO systems. The Significance of software measurement along-with the increasing interest in prototypical features leads towards reconsideration of software metrics and to answers the questions associated with the effectiveness of applying traditional software metrics to OO systems.

Functionally object oriented metrics can be divided into three categories from prototype perspective:

1. Some metrics needs to be redesigned for prototype based languages.
2. Some metrics needs no change as some features of class based and prototypes based languages are similar.
3. For some metrics old day's structural languages metrics can be used.

4. Some metrics are no longer needed for prototype based languages.
5. Some new metrics needs to be presented for novel features of prototypes.

Table 1: Comparison of Object Oriented and Prototype languages [13]

No.	Features	Cass-Based Techniques	Prototype based techniques
1	Basis	Mathematical concept – set of knowledge representation	Knowledge representation through object observation
2	Object Modeling Parameters	(i) defined an object by distinct parameters structure and state (ii) Object is not defined in an incremental fashion	(i)defines an object by a single parameter prototype and doesn't differentiate between structure (or meta data) and state (or data) of the object.
3	Organization of objects of a system	Objects are organized into a hierarchical structure, called a class lattice	Objects are not organized in any hierarchical structure – no class-latice
4	Tracing of changes to a specific object	Not possible	Possible since each change to an object is stored in a separate prototype
5	Knowledge sharing mechanism	Inheritance mechanism, and it is static mechanism	Delegation mechanism and it is dynamic mechanism
6	Fixation of Message Passing Pattern	Message passing pattern is fixed at compile time	Message passing pattern is fixed at run time
7	Retention control while message passing	Control remains with the self class while the search goes to the next super class	Control is passed to the next prototype with the search delegation
8	Flexibility and Efficiency	(i) In case of simple inheritance and single parent delegation, both mechanism are equally powerful (ii) Otherwise it is less flexible and less powerful than the delegation mechanism (iii) Efficiency is predictable	(i)In case of simple inheritance and single parent delegation, both mechanism are equally powerful (ii)More flexible and powerful than the inheritance mechanism (iii)Efficiency is not predictable.

Type 1. Metrics to be redesigned: In the traditional object oriented systems knowledge sharing between object and classes is typically done by a mechanism called inheritance, initially used by the language Simula and later adopted by most of the modern object oriented languages. Every class contains a common behavior for a set of objects along-with the description of what characteristics are allowed to vary among objects. It is important to note that all instances of a class share the same behavior, but can maintain unique values for a set of state variables pre-declared by the class. There are a number of metrics available for object oriented systems to deal with inheritance. Some of such metrics are Attribute Inheritance Factor (AIF) [11], FAN-IN [9], Method Inheritance Factor (MIF) [2] and Number of Methods Inherited (NMI) [12]. Here AIF counts the ratio of the sum of inherited attributes in all classes of the system under consideration to the total number of available attributes for all classes. FIN is the number of classes from which a class is derived and high values indicate excessive use of multiple inheritances. MIF is the ratio of the sum of the inherited methods in all classes to the total number of available methods for all classes and NMI measures the number of methods a class inherits.

Type 2: While on the other hand prototypical approach for sharing knowledge in object oriented systems is based on an alternative mechanism called delegation, appearing in several languages [7], [5],and [8]. In this approach the distinction between classes and instances is removed in sense that any object can serve as a prototype. To create an object that shares knowledge with a prototype, an extension object is constructed, which contains a list of its prototypes which may be shared with other objects. When an extension object receives a message, it first attempts to respond to the message using the behavior

stored in its personal part. If the object's personal characteristics are not relevant for answering the message, the object forwards the message on to the prototypes to see if one can respond to the message. This process of forwarding is called delegating the message. Keeping in mind this difference of sharing knowledge among objects and classes the traditional metrics being used for inheritance needs to be changed with more sophisticated metrics for delegation. For instance the prototypical version for above mentioned metrics can be designed such as in AIF instead of counting the ratio of inherited attributes/objects, number of delegated objects can be calculated. FIN is the number of classes from which a class is derived while in class-less languages prototype is used instead of classes so FIN can be redesigned for number of prototypes from which other prototypes are delegated. Similarly a prototypical version of MIF and NMI can also be designed.

Type3: Source Lines of Code (SLOC or LOC) is one of the most widely used sizing metrics in industry and literature. Size is one of the most important attributes of a software product. It is not only the key indicator of software cost and time but also a base unit to derive other metrics for project status and software quality measurement. According to [7] survey on cost estimation approaches, size metric is used as an essential input for most of cost estimation models. SLOC is the traditional and the most popular sizing metric. Its long-standing tradition is due to the fact that SLOC is the direct result of programming work. In the early age of software development, most of software cost was spent on programming, and SLOC emerged as the most perceivable indicator of software cost. Unfortunately, SLOC has a number of shortcomings [3]. One significant deficiency is the lack of precise and methodical guideline for determining what SLOC means. Another feature lacking in SLOC counts, reducing its usefulness as an effective size measure for understanding a piece of code is that it doesn't account for complexity of a line of code.

These ambiguities of complexity between difference lines of code is addressed in the new metric of S/C (size/Complexity) described by Pant [3]. The S/C measure is based on the notion that, in a high-order programming language, decision making and iterative statements are normally more complex than assignment statements. This metric count's one for simple statements, one for each binder and one for each simple predicate. It also takes into account the number of mental paths within the control flow structure and allows for nested structures. Contrary to this notion prototype languages donot take advantage of such complex or nested structure instead all statements in the languages are used just like simple message passing statement and functionally same behavior is followed by most statements in prototype languages. This analysis implies that in prototype languages there is no need to use the enhanced version of SLOC and the older version of this metric can easily be deployed to measure size of the program as each statement uses equal level of complexity.

Type 4 Metrics: There are many metrics in used to measure cohesion and complexity of classes such as attribute hiding factor[4], class cohesion[5] and class entropy complexity[6] etc. Hereby attribute hiding factor measures the ratio of the sum of inherited attributes in all system classes under consideration to the total number of available classes attributes. While class cohesion measures relations between classes and class entropy complexity measures the complexity of classes based on their information content. Dony, Malenfant & Cointe [2] outlined two primary arguments in favor of prototype-based object oriented programming. First, it is easier to figure out concrete examples before generalizing concepts into abstract definitions. Second, classes add unnecessary constraints by preventing the customization of individual object instances as well as the inheritance of member data values. Although the first issue tends to be more of a philosophical or process-oriented distinction, the second issue regarding class constraints deals with a potential limitation in the available programming constructs that support the abstraction and encapsulation of concerns in software. In order to remove this limitation there is a sheer need to use classless approach which entails to omit the use of metrics used for measuring classes.

Type 5 Metrics: One of the notable feature of prototype based languages is that the object system are dynamic in that the objects can be created, updated and destroyed during program execution and even the type of values can be changed [8,9]. Every object is self-describing and can be changed independently and each change to an object is stored in a separate prototype. In class based systems any object cannot be created or associated with class at run time. Although this might seem handy, it also means that you have to be very careful when you update a prototype object. You have to know the object dependency graph of the objects derived from the prototype object, in order to do a safe update. The object systems also support a part-owner hierarchy, by which objects can be grouped together [9]. For instance, the graphics in a

window are added as *parts* of the window.

References

1. O. Agesen, L. Bak, C. Chambers, B.W. Chang, U. Holzle, J. Maloney, R.B. Smith, D. Ungar and M. Wolczko. The Self 3.0 Programmer's Reference Manual. Sun Microsystems Inc and Stanford University, 1993.
2. O. Agesen, L. Bak, C. Chambers, B.W. Chang, U. Holzle, J. Maloney, R.B. Smith, D. Ungar and M. Wolczko. The Self 4.0 Programmer's Reference Manual. Sun Microsystems Inc and Stanford University, 1995.
3. Moreau, *A Programming Environment Evaluation Methodology for Object-Oriented Systems*, Ph.D. Dissertation, University of Southwestern Louisiana, Sep. 1987 .
4. D. R. Moreau and W. D. Dominick, "Object-Oriented Graphical Information Systems: Research Plan and Evaluation Metrics," *Journal of Systems and Software*, vol. 10, pp. 23-28, 1989.
5. D. R. Moreau and W. D. Dominick, "A programming environment evaluation methodology for object oriented systems: part I - the methodology," *Journal of Object-Oriented Programming*, vol. 3, pp. 38-52, May/Jun. 1990.
6. B. Boehm, C. Abts, S. Chulani, "Software development cost estimation approaches: A survey", *Annals of Software Engineering, 2000*.
7. Pant, H. Sellers and Verner, 1995, S/C: a software size /complexity measure, chapter 50 in software quality and productivity , theory, practice, education and training
8. K.L. Morris, "Metrics for Object-Oriented Software Development Environments", Master thesis, M.I.T., 1988.
9. R.S. Chidamber & C.F. Kemerer, "Towards a metrics suite for object–oriented design", Proceedings of the OOPSLA '91 Conference, 1991, pp 197-211.
10. J. Bansiya & C. Davis, "Using QMOOD++ for object-oriented metrics", Dr. Dobb's Journal, December 1997.
11. The Prototype-Instance Object Systems in Amulet and Garnet, Brad A. Myers, Rich McDaniel, Rob Miller, Brad Vander Zanden, Dario Giuse, David Kosbie and Andrew Mickish, *To appear in:*James Noble, Antero Taivalsaari and Ivan Moore, eds., *Prototype Based Programming*, Springer-Verlag, 1998.
12. Inheritance vs. delegation: Is one better than the other? Peter Bosch April 6, 1993
13. Syed Ahsan " Development of and Extensible DBMS for Biological Data". PhD Dissertation 2009.

8/8/2012