Measuring Code Complexity in Projects Designed with Aspect/J ä

Jana Dospisil
Monash University, Australia

jana.Dospisil@infotech.monash.edu.au

Abstract

The modularized code encapsulating object interactions is characterized by class hierarchies. In the implementation of mobile agents, we have observed that the changes in agent interaction protocols lead to uncontrolled subclassing and consequently to disorder. This phenomenon is known as entropy. The additional subclassing, modification to protocols, restructuring of the class hierarchies, changes to visibility of attributes, and method overloading result in increased complexity of the code. This problem in agent design has been tackled by Kendall (Kendall, 1999) who proposed development using Aspect/J and separation of concerns. Since there has been no proof of reduced complexity, we have proposed metrics for software complexity estimation, and ranking of compositional elements developed with Aspect/J. The metrics have been tested on Java code for mobile agents.

Keywords*:* complexity metrics, information theory, object-oriented programming, separation of concerns.

Introduction

The source of the problem in development of interaction protocols is that some kinds of behaviour or functionality are *orthogonal to* or *cross cut* classes in many object-oriented components, and they are not easily modularized to a separate class. Examples of such behaviour include the following:

- synchronization and concurrency
- performance optimization
- exception handling and event monitoring
- coordination and interaction protocols
- object views.

In Tarr, Ossher, Harrison, and Sutton (1999), Tarr states that *"Done well, separation of concerns can provide many software engineering benefits, including reduced complexity…"*. To measure the quality of separation either in *N-dimensional* space or even the orthogonal separation only as seen in *Aspect/J*, the new set of complexity metrics is required.

The paper is organized as follows:

- Section 2 provides an overview of established metrics and introduces the notion of complexity and entropy based metrics for complexity.

Paper Accepted as a Short Paper

- Section 3 deals with the concept of separation of concerns and main concepts in Aspect/J.
- Section 4 provides theoretical underpinning of proposed metrics suite.

Brief Survey of Object Oriented Metrics

Since 1995, the trend towards incorporating measurement theory into all software metrics has led to identification of scales for measures, thus providing some perspective on dimensions. The most common scale types based on measurement theory are: *Ordinal, Interval, Ratio,* and *Nominal.* Fetchke and Zuse (Fetchke, 1995; Zuse, 1994) analyzed the properties of object oriented software metrics on the basis of measurement theory. The underlying notion of measurement theory is based on intuitive or empirical existence of relationships among objects within our Universe of Discourse. These relationships are formally described by a mathematically derived formal relational system. Zuse also investigated how and under what conditions the software measures may be viewed as ordinal, ratio, nominal, and interval. He admits that these scale types present very little meaning with regard to quality measures such as maintainability and error-proneness of the application.

The contribution of Zuse and Fetchke's work is in the introduction of a specific perspective of measures. They emphasize preciseness of definition of scales as well as definition of an attribute that is measured.

The **Axiomatic** approach, proposed by Weyuker (Weyuker, 1988), provides a framework based on a set of nine axioms as listed in Table 1.

Axiom	Name	Description
1	Noncoarseness	$(\exists P)(\exists Q)(m(P) \neq m(QQ))$
2	Granularity	Let c be non-negative number. Then there is a finite number of class with the complexity = c
3	Nonuniqueness	There is distinct number of classes P and Q such that m(P) = m(Q)
4	Design detail matter	$(\exists P)(\exists Q)(P \equiv Q and m(P) \neq m(Q))$
5	Monotonicity	$(\forall P)(\forall Q)(m(P) \leq m(P+Q) and m(Q) \leq m(P+Q))$
6	Non-equivalence of interaction	$a)(\exists P)(\exists Q)(\exists R) m(P) = m(Q) and m(P+R) \neq m(Q+R)$ $b)(\exists P)(\exists Q)(\exists R) m(P) = m(Q) and m(R+P) \neq m(R+Q)$
7	Interaction among statements	Not considered among objects
8	No change on renaming	If P is renaming of Q then $m(P) = m(Q)$
9	Interaction CAN increase complexity	$(\exists P)(\exists Q)(m(P) + m(Q) < m(P+Q))$

Table 1: Weyuker's axioms

In Weyuker's metric proposal we observe the formalization of structural inheritance complexity metrics. The ninth axiom means that splitting one class into two classes can reduce the complexity. The experience supports the argument by Chidamber and Kemerer (1994) that the complexity of interaction may even increase when classes are divided.

Fenton (Fenton & Pfleger, 1997) uses the term *software metrics* to describe the following artifacts:

- A number that is derived, usually empirically, from a process or code (for example, Lines of Code (**LOC**) or number of function points).
- A scale of measurement (an example used in Fenton's book is nominal scale or classification).
- An identifiable attribute that is used to provide specific functionality. (An example is "portability" or class coupling metric.)
- Theoretical or data driven model describing a dependent variable as a function of independent variables. (An example can be the functional relationship between maintenance effort and program size.)

These descriptions typically lead to a widespread confusion between models and their ability to predict desired software characteristics, and thus their suitability to be used for estimation purposes.

Metric	Description
Weighted Methods per Class (WMC)	$WMC = \sum_{i=1}^{n} c_i$ where c_i is the static complexity of each of the n methods.
Depth of Inheritance Tree (DIT)	With multiple inheritance the max DIT is the length from the node to the root.
Number of Children (NOC)	Number of immediate subclasses
Coupling Between Object Classes (CBO)	Number of other classes to which a particular class is coupled. CBO maps the concept of coupling for a class into a measure.
The Response for a Class (RFC)	The size of response set for a particular class.
The Lack of Cohesion metric (LCOM).	$LCOM = \lvert P\rvert - \lvert Q\rvert if \lvert P\rvert > \lvert Q\rvert = 0 otherwise$

Table 2: Chidamber and Kemerer Metrics (Chidamber, 1994)

The metrics of Chidamber, summarized in Table 2, also have foundation in measurement theory. The authors do not base their investigation on the extensive structure. The criticism by Churcher and Sheppard (1994) is pointing to the ambiguity of some metrics, particularly **WMC**. Hitz and Montazeri (1996) and Fetchke (1995) showed that **CBO** does not use a sound empirical relation system, particularly, that it is not based on the extensive structures. Furthermore, **LCOM** metric allows representation of equivalent cases differently, thus introducing an additional error.

Coupling measures form the important group of measures in the assessment of dynamic aspects of design quality. Coupling among objects is loosely defined as the measure of the strength of the connection from one object to another. The approaches of different authors mostly differ in definition of the measured attribute – coupling among classes. Table 3 provides the summary of differences in definitions. Some of the attributes may be known very late in development.

Two aspects affect coupling between classes: the frequency of messaging between classes (cardinality and multiplicity of objects derived from these classes), and the type of coupling. The discussion in Eder and Kappel (1994) distinguishes among three types: 1) interaction coupling, 2) component coupling and 3) inheritance coupling.

The degree of coupling is based on defining a partial order on the set of coupling types. The low end is described by small and explicit inter-relationship and high end of the scale is assigned to large, complex and implicit inter-relationship. The definition is subjective and requires empirical assignment of values in order to be used as software quality indicator.

Attribute definition	Eder et al (Eder.1994)	Hitz & Montazeri (Hitz, 1995)	Briand *et al.* (Briand, 1999)
Public attribute visibility	X		
Method references attribute		X	
Method invokes method	X	X	X
Aggregation	X		X
Class type as a parameter in method signature or return type	X		X
Method's local variable is a class type	X		
A method invoked from within another method passes class type as a parameter	X		
Inheritance	X		
Method receives pointer to method			X

Table 3: Comparison of attribute definition for coupling (Briand, Daly, & Wurst, 1999)

Cohesion is defined as a degree to which elements in a class belong together. The desirable property of a good design is to produce highly cohesive classes. Comparison of different frameworks and discussion can be found in Briand's work (Briand, Daly, & Wurst, 1997). Eder (Eder. & Kappel, 1994) provides a comprehensive framework that requires semantic analysis of classes and methods. The metrics of Chidamber define *LCOM* as the number of disjoint sets created by intersection of the *n* sets. The definition in Equation 1 does not state how inheritance of methods and attributes is treated with regard to method override or overload and the depth of inheritance tree.

$$LCOM = \begin{cases} |P| - |Q|, if\ |P| > |Q| \\ 0, otherwise \end{cases}$$

Equation 1 LCOM definition

In Henderson-Sellers (1996) the cohesion measure is based on the number of attributes referenced by a method.

$$LCOM = \frac{\frac{1}{a}\sum_{j=1}^{a} m(A_j) - m}{1 - m}$$

Equation 2 LCOM by Henderson-Sellers

where a is number of attributes, and $m(A_j)$ is the measure which yields 0 if each method in the class references all attributes, and 1 if each method in a class references only single attribute.

Complexity Measures

Entropy-based complexity measures are based on the theory of information. This is the approach taken by Davis and LeBlanc (Davis & LeBlanc, 1988) who quantify the differences between *anded* and *neted* structures to provide an unbiased estimate of the probability of occurrence of event *m*. This measurement is based on chunks of FORTRAN and COBOL code (represented by nodes in the DAG (Directed Acyclic Graph)) with the same in-degree and the same out-degree to assess syntactic complexity.

Belady and Lehman (Belady & Lehman 1976) elaborated on the law of increasing entropy: the entropy of a system (the level of its unstructuredness) increases with time, unless specific work is executed to maintain or reduce it. Entropy can result in severe complications when a project is modified, and it is generally an obstacle to maintenance.

The use of entropy as a measure of information content, introduced by Harrison, has been around since 1992 (Harrison, 1992). Harrison's software complexity metric is based on empirical program entropy. A special symbol, reserved word, or a function call is considered as operator. It is assumed that they have certain natural probability distribution (Zweben & Hasltead, 1979). The probability p_i of i^{th} most frequently occurring operator is defined as

$$p_i = f_i / N_i$$

Equation 3 Probability of occurrence of i-operator

where f_i is the number of occurrences of the i^{th} operator, and N_i is the total number of nonunique operators in the program.

The complexity is defined as entropy

$$H = -\sum_{i=1}^{Ni} p_i \log_2 p_i$$

Equation 4 Entropy-based complexity measure

The *Average Information Content Classification* (***AICC***) measure is defined as:

$$AICC = -\sum_{i=1}^{Ni} \frac{f_i}{N_i} \log_2 \frac{f_i}{N_i}$$

Equation 5 Average Information Content Classification (*AICC*)

Harrison assessed the performance of these entropic metrics in two commercial applications written in C language with the total number of lines of code being over 130,000.

The work of Bansiya and Davis (Bansiya, Davis, & Etzkorn, 1999) introduces a similar complexity measure – *Class Definition Entropy* (*CDE*) – replacing the operators of Harrison with name strings used

in a class. The assumption that all name strings represent approximately equal information is related to the possible error insertion by misusing the string. The metric has been validated on four large projects in C++ and results have been used to estimate a *Class Implementation Time Complexity* measure.

Critical View of OO Metrics

Many metrics deal predominantly with static characteristics of code. Hitz (Hitz & Montazeri, 1995) clearly distinguishes the difference between *static* and *dynamic class method invocation*: *number of methods invoked by a class compared to frequency of method invocation*.

A metrics suite capable of capturing dynamic behaviour of objects with regard to coupling and complexity has been presented by Yacoub (Yacoub, Ammar, Hany and Robinson, 1999) where the dynamic behavior of an implementation is described by a set of scenarios. The *Export* and *Import Object Coupling* metrics are based on percentage of message exchange between class instances (objects) to the total number of messages. The *Scenario Profiles* introduce the estimated probability of the scenario execution. The complexity metrics are aimed predominantly at the assessment of stability of active objects as frequent sources of errors.

Obvious criticism of Henderson-Sellers metrics include the typical interaction among objects: 1) How should we treat inheritance, for example, access to superclass attributes? 2) How we treat method-to-method calls. Many metrics show dimensional inconsistencies, or their results are derived from correlative or regression analysis. As reported in Gursaran and Gurdev (2001), experienced object-oriented designers found memory management and run-time errors are more problematic and difficult to deal with.

Class size problems represent a confusing effect with regard to validity of object-oriented metrics. The confounding effect of class size has been reported by Khaled, Benlarbi, Nishith, & Shesh (2001), provoking some doubts about the validity of software metrics currently being used as early quality indicators. The relationship of high coupling factor to fault proneness seems to support the hypothesis that large class size may have a negative impact on the quality and occurrence of faults.

Harrison's entropic metrics are intended to order programs according to their complexity. Since entropy provides only the ordinal position thus restricting the way of usage, we cannot measure the *distance* between two programs.

Aspect/J Concepts

Separation of concerns is a methodology used to identify, encapsulate, and manipulate those software entities that are relevant to a particular concept or purpose.

Concern is a primary element of decomposition. At present, two products support this kind of modularization: *Hyper/J* from IBM (Tarr, 1999) and *Aspect/J* (Kiczales, Lamping, Mendhekar, Maeda, Lopes, Loingtier, & Irwin, 1997) from Xerox Palo Alto.

In large projects, many classes encapsulate additional code (called crosscutting code) to handle extra processing aspects such as synchronization and concurrency. These aspects cause the class design principle of well-defined responsibilities to be violated. *Dynamic Joint Point Model* (Aspect/J, 2002) provides a framework for many kinds of *Joint Points* in the execution of program. For example, *Method Joint Point* describes the actions of an object when it receives a method call. Point designators (*pointcuts*) identify the *joint points* in the program flow.

Definition of additional code at joint point is designated within *advice* (*before* and *after advice*). The modifications of classes and their hierarchy (structural modifications) are determined by *introduction*. The *introduction* also maintains the visibility of new elements with regard to the aspect addressed. *Reflection* is a special reference variable that contains reflexive information about the current joint point.

The newest feature of *Aspect/J* is the construct called *aspect*. *Aspects* are defined as units of modularity for crosscutting concerns.

Entropy Based Metrics Framework for Aspect/J

We are interested in certain measurable attributes that we want to control and predict, for example, the occurrence of faults or code maintainability. Software applications form a dynamic system composed of many elementary entities, each one behaving according to its own parameters. Such a system could be described by a set of differential equations with 10^{20} degrees of freedom or more. We are interested in two related phenomena:

- **Observables** arising from their average behavior, such as correct interactions, error handling aspects, and unexpected behavior.
- Impact of **evolutionary changes** caused by introducing additional methods, attributes, subclasses, etc. These changes result in increased disorders and complexity.

Complexity is a subjective property and it can be defined as the relationship between the observer and the application. If the observer is satisfied with a simple model and its accuracy then the application is not complex and well-defined test suites uncover possible unreliability. On the contrary, if the observer requires high degree of reliability, the application then represents a complex system with many degrees of freedom.

Two distinct classes of application complexity can be defined:

- ***Complexity of a solution (SC)*** is the amount of resources needed to produce a solution for a given problem.
- ***Algorithmic complexity (AC)*** is defined as the complexity of the structure that implements the solution. Aspect programming can improve modularization and encapsulation of commonalities. By extracting aspects common to multiple units we aim to reduce algorithmic complexity.

Control flow in a program is the order in which contained units of code are executed. Control flow is then a measure of object interaction.

The primary lexical abstraction in Java and Aspect/J is a symbol (called either *name* or *identifier*). It is represented by a character string with the following properties: *scope* (private, protected, public, and package), and *type and storage* class (class variable or method, import).

The set of symbols includes:

- Class names
- Public and static variables and methods
- *Aspect keywords* and *pointcut* names

Strictly local symbols are excluded from the model. Local variables within methods, and parameter names in the method signature do not alter object control flow. Furthermore, we define class as being the main dimension stream of modularity (*class stream*) and aspects as being secondary crosscutting streams (*aspect stream*). Each identified *joint point* in the *aspect stream* describes a relationship to global symbols in the *class stream*.

To extract symbols and obtain the graphical representation of control flow, we have used concern graphs and the FEAT tool (Murphy, Lai, Walker, & Robillard, 2001), which displays the concerns as a collection of class trees. The root of each tree is a class (see Figure 1), which implements some behaviour and

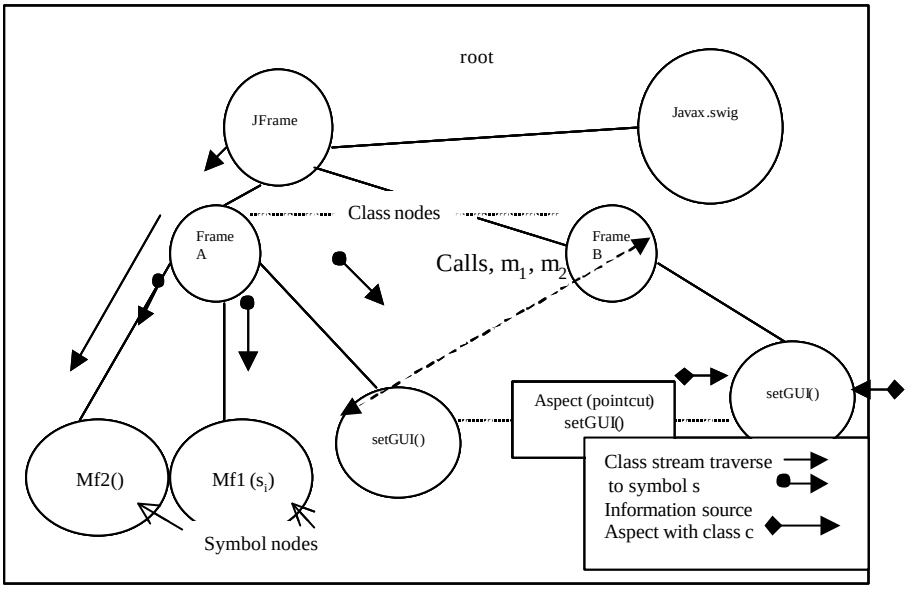

Figure 1 Control flow graph

data. Based on this analysis, we formed the structural system model as graph *P* composed of different types of nodes and edges (Figure 1):

- *Symbol nodes* s_i are end nodes that correspond to global symbols s in the class stream (for example, methods such as setGUI(), Mf1()),
- *Class nodes* c_j are represented by the top structural units from which all derived nodes are sourced (*JFrame* sources FrameA and *FrameB* as well as *setGUI()* method).

Edges represent dependencies between the following elements:

- *class nodes* and *aspects* that describe the dependency between aspects and classes using the particular aspects. *Aspect dependency* refers to treatment of crosscuts.
- *symbols and nodes* which directly provide the source for symbol. *Symbol dependency* refers to dependencies relevant to processing logic.

Definitions and abbreviations used in further equations are summarized in Table 4. Each symbol invocation requires V_s messages, one message for each symbol node. Since the structure is already known, the messages will traverse nodes in order of dependency edges. Source and destination symbol nodes denote each edge. We can describe the path of symbol s_i as follows:

$$message(path\_to(s_i)) = \{V_e(s_i), node_1, node_2, \ldots node_n\}$$

The entropy of the message describing a symbol node s_i is the sum of entropies (entropy of the total number edges which serve as information source to this symbol node and entropy of finding the edge destination if we know the symbol source). This is the *Path Entropy*:

$$H(P) = H(S) + \frac{V_e}{V_s + V_c} H(E)$$

Total number of edges in graph is denoted by the sum of edges:

$V_{e\_all} = \sum V_e$	
$V_s = \sum_{i=0}^{I} s_i$	Total number of all **symbol nodes** in the graph. We assume that the number of symbol nodes is within the range 0 to I-1.
$V_c = \sum_{j=0}^{C} c_j$	Total number of class nodes range from 0 to C-1
$V_e = V_d + V_a$	Number of all **edges** in the graph (aspect edges - and symbol edges) relevant to the particular symbol and aspect.
$V_d = \sum_{i=0}^{I} e_i^s$	Total number of dependency edges in the class stream traversed to a symbol *s (for0<s<=S).* *A*ssumption: the symbol s may occur in multiple nodes.
$V_a = \sum_{a=0}^{A} e_a^c$	Total number of edges with aspects that have an entry in class *c* (e_a^c → a[th] aspect edge associate with class c); the aspect is used by multiple classes)
$p(e_i) = \frac{V_d}{V_s}$	Probability that *i*[th] symbol node is sourced by *e* edges
$p(e_a) = \frac{V_a}{V_d}$	Probability that a random symbol node *i* has an aspect associated with it
$p_e(d)$	Probability that the dependency edge has length d (the number of nodes needed to be traversed to reach the edge *e* is *d*)
$p(d)$	Probability that two random symbol nodes *i* will have *d* distance between them.
$H(S) = -\sum_{i=1}^{M} p(e_i)\log_2 p(e_i))$	Entropy of the total number edges which serve as information source to a symbol node
$H(E) = \sum p(d)\log_2(V_s p(d))$	Entropy of finding the edge destination if we know the starting point (symbol source)
$H(A) = -\sum_{i=1}^{M} p(e_a)\log_2 p(e_a)$	Entropy of aspects

Table 4 Definitions

We divided our metrics proposal into two parts:

1. Entropy based ordering of symbols within modules.
2. Weighted entropy measures.

Symbol ordering provides perspective on the complexity of each module and the usage of symbols. Weighted entropy measures provide a subjective view based on the associated failure risk and maintainability of each module.

Entropy based ordering of symbols: We consider the entropy for the aspect stream as the reduction in uncertainty for those edges that have some information "outsourced" to aspects. Aspects connect the classes (via symbols) that would not be connected otherwise.

The reduction in total entropy is then calculated as

$$I = H(E) - H(A)$$

Equation 6 Reduction in entropy with aspects

The probabilities have been computed as shown in Table 4. They can be also derived empirically from execution scenarios.

Weighted entropy: A criterion for a qualitative differentiation of the units of a given code segment represented by the relevance, the significance, or the utility of the information they carry with respect to an outcome, and to a qualitative characteristic.

The occurrence of a symbol removes a double uncertainty: the quantitative one, related to the probability with which it occurs (it is found in the code), and the qualitative one, related to a given qualitative characteristic (anticipated failure risk factor). For instance, a symbol of a small probability (such as an object with small cardinality number) can have a great utility with respect to possible incorrect coding. Likewise, a symbol of high probability can have small impact on maintainability (we shall relate this observation to utility). The weight of one symbol may express some qualitative objective characteristics, but also it may express the subjective utility of the respective symbol with respect to the software complexity.

In order to distinguish the dependency edges according to their importance with respect to a given qualitative characteristics, we assign to each edge type a non-negative weight proportional to its importance and significance. Weights are treated as the ratio of the objective probability that the edge path is the source of information for symbol s_i.

$$H(e) = -\sum_{i=1}^{I} w(e_i)\, p(e_i) \log_e p(e_i)$$

$$w(e_i) = -\frac{p(e_i)}{\log_e p(e_i)}$$

In this case we obtain the following expression for weighted entropy:

$$H(e) = \sum_{i=1}^{I} p(e_i)^2$$

Equation 7 Weighted entropy

In order to acquire more objective weights for different types of dependency edges, we have introduced finer granularity for edges, based on the information provided by the FEAT tool and execution scenarios. Each edge type can assume one of the subtypes in Table 5.

Edge correspondence in FEAT	Description	Coupling measure as weight in % *MOB-Trader*
Calls, m_1, m_2	Method m_1 calls m_2	80
Reads m, f	Method m reads value of f	0
Writes m,f	Method m writes in f	50
Checks m, c	Method m checks class c	0
Creates m, c	Method m creates object of class c	30
Declares c, {m}	Class c declares method m	0
Superclass c_1, c_2	Class c_2 is the superclass to c_1	No – included in class nodes dependency

Table 5 Symbol node classification and weight assignment

By introducing weighted entropy, we can distinguish between the quantitative uncertainty, related to the probability with which a symbol occurs, and the qualitative one, related to a given qualitative characteristic (anticipated failure risk factor or property of software we want to measure). Furthermore, by balancing weights for observables and evolutionary changes (using empirical values) we can obtain sufficient foundation for prediction models.

Some Results

We have used entropy metrics for ordering of symbols to estimate the complexity of the mobile agent application *MOB-Trader*. The application implements complex trading scenarios of multiple vendors (*Moderator* module) and mobile buyer agents (*Buyer* modules). In the *Buyer* modules, Aspect/J enhances the modularization by placing exceptions and handling of preconditions and post conditions in aspects instead of special classes and subclasses.

Some submodules required threaded architecture to implement dynamic behaviour. With threads, the breach of concurrency rules often results in random faults difficult to debug. Placing an aspect in the *Runnable* interface may propagate such faults across entire system.

Some results of the experiments are in Figure 2, which shows the comparison of entropy values for two modules: Module *Moderator* and Module *Buyer*. The *Moderator* had 53 symbol nodes with 20 class nodes, and only a few aspects. The *Buyer* had 10 symbol nodes and deep class hierarchies (35 classes and subclasses). Many classes had attached aspects implementing the negotiation strategy or error handling submodules.

The *Moderator* module shows higher entropy for finding edge destination, path entropy (total entropy) and weighted entropy due to its multithreaded implementation. Entropy H(A-1) was calculated for both modules on implementation without aspects.

Usability Remarks on Proposed Metrics

Object-oriented code is characterized by class hierarchies that are shared structures. Very often some additional subclassing, modification to existing classes, the restructuring of the hierarchy itself, and changes in the visibility of attributes and sometimes even methods are made. Given these changes (with

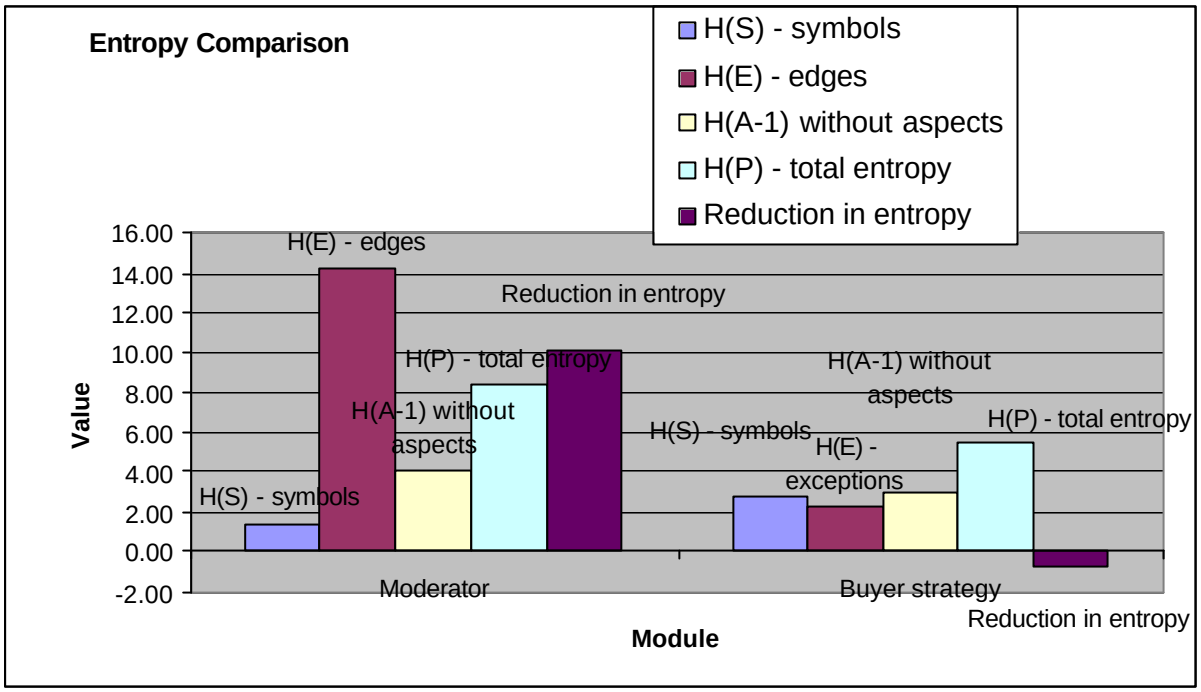

Figure 2: Entropy comparison for Moderator and Buyer modules

and possible lack of comprehensive documentation) and time constraints, we assume that class hierarchies will become the subject of increased entropic tendencies in the implementation. In our mobile application, we have observed that the probability that a subclass will not consistently extend the content of its superclass is increasing with the depth of hierarchy. The tools like *Hyper/J* and *Aspect/J* support the separation of concerns, thus allowing a different approach to evolving the content rather than extending the class hierarchies.

This paper presents proposed entropy based metrics for object-oriented development with *Aspect/J*. The entropy metrics are useful in ranking different modules and symbols with regard to their complexity. The single-valued measure of complexity is appealing as a quality indicator. However, as also discussed in Fenton's book (Fenton, 1997), the results may not be suitable for use in prediction models or as guidance for improving the quality of the product. In order to tackle these shortcomings, we have introduced weighted entropic values to accommodate the subjective perspective of an observer. This approach provides multi-valued space more suitable for prediction models.

We recognize that larger and more diverse samples must be collected to acquire more realistic weighting. Due to the limited space, we have not included more results or concise methodology for collecting data and the metric results explanations.

References

Aspect/J (2002). The AspectJ™ Programming Guide. Xerox Corporation, http://AspectJ/doc/progguide/printable.html.

Bansiya, J., Davis, C., & Etzkorn, L. (1999). An Entropy-Based Complexity Measure for Object-Oriented Designs, *Theory and Practice of Object Systems*, Vol. 5(2), pp.11-118.

Belady, L.A. & Lehman, M.M. (1976). A Model of a large program development. *IBM Systems Journal*, Vol 15(3), pp.225-252.

Briand, L. Daly, J., & Wurst. (1999). A Unified Framework for Coupling Measurement in Object Oriented Systems. *IEEE Transactions on Software Engineering*, Vol. 25, No. 1, pp 99-121.

Briand, L. Daly, J., & Wurst. (1997). A Unified Framework for Cohesion Measurement in Object Oriented Systems. *Technical Report*, ISERN-97-05.

Chidamber, S. & Kemerer, C. (1994). A Metric Suite for Object Oriented Design. *IEEE Transactionas on Software Engineering*, Vol.20,No.6, June, pp. 476-49.

Churcher, N. & Shepperd, M. (1994). A Metric Suite for Object Oriented Design. *IEEE Transactionas on Software Engineering*, Vol.20,No.6, June, pp. 476-49.

Davis, J.S., & LeBlanc, R.J. (1988). A Study of the Applicability of Complexity Measures. *IEEE Transactions on Software Engineering*, Vol 14(9), pp.1366-1371.

Eder, J. & Kappel, G. (1994). Coupling and Cohesion in Object Oriented Systems. *Technical Report*, University of Klagenfurt.

Khaled E., Saida Benlarbi, Nishith Goel, & Shesh N. Rai. (2001). The Confounding Effect of Class Size on Validity of Object-Oriented Metrics. *IEEE Transactions on Software Engineering*, Vol. 27. No.7.

Fetchke, T. (1995). Software Metriken bei der Objectorientierten Programmierung. *Diploma Thesis*, GMD, St. Augustin.

Fenton, N. & Pfleger, S. L. (1997). *Software Metrics: A Rigorous & Practical Approach*. International Thomson Computer Press.

Gursaran & Gurdev Roy. (2001). On the Applicability of Weyuker Property 9 to Object Oriented Structural Inheritance Complexity metrics. *IEEE Transactions on Software Engineering*, Vol. 27. No 4. April.

Harrison, W. (1992). An Entropy-Based Measure of Software Complexity. *IEEE Transactions of Software Engineering*, Vol. 18, No. 11, November, pp. 1025-1029.

Henderson-Sellers, B. (1996). *Object-Oriented Metrics measures of Complexity*. Prentice Hall PTR.

Hitz, M., & Montazeri, B. (1995). Measuring Product Attributes of Object-Oriented Systems. In *Proc. 5th European Software Engineering Conference (ESEC'95*), Barcelona, Spain, pp. 124-136.

Hitz, M., and Montazeri. (1996). B. Chidamber & Kemerer's metric Suite: A Measurement Theory Perspective. *IEEE Transactions on Software Engineering*, Vol. 22. No. 4, pp270-276.

Kendall, E. (1999). Role Model Designs and Implementations with Aspect Oriented Programming, Proceedings of the 1999 Conference on Object- Oriented Programming Systems, Languages, and Applications (OOPSLA'99), ACM Press, November.

Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. - M. Loingtier, & J. Irwin. (1997). Aspect Oriented Programming. Xerox Corporation. http://www.parc.xerox.com/spl/projects/aop/

Kitchenham, B. (1996). *Software metrics*. Blackwell.

Murphy, G., C. Albert Lai, Robert J. Walker, & Martin P. Robillard. (2001). Separating Features in Source Code: Exploratory Study. *Proc. Of the 23th International Conference on Software Engineering*, Toronto, pp. 275-284.

Tarr, P. Ossher, H. Harrison, W, & Sutton, Jr. (1999). N degrees of Separation: Multi-Dimensional Separation of Concerns. *Proc. Of the 21st International Conference on Software Engineering*.

Weyuker, E. J. (1988). Evaluating Software Complexity Measures. *IEEE Transactions on Software Engineering*, Volume: 14, No. 9, pp. 1357 – 1365.

Zweben, S. & Hasltead, M. (1979). The Frequency Distribution of Operators in PL/I Programs. *IEEE Transactions of Software Engineering*, Vol SE-5. pp.91-95, March.

Zuse, H. (1994). *Software Complexity Metrics/Analysis*. Marciniak, J. (Editor-in Chief): Encyclopedia of Software Engineering, Vol. I, John Wiley& Sons, Inc., pp. 131-166.

Yacoub S. M., Ammar, Hany,H. & TomRobinson. (1999). Dynamic metrics for Object Oriented Designs. *Proc. Of 6th International Symposium on Software Metrics (METRICS'99)*, Boca Raton, Nov. 4-6, pp. 50-61.