JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2006**

Vol. 6, No. 1, January-February 2006

Software Design Metrics for Object-Oriented Software

K.K.Aggarwal, Yogesh Singh, Arvinder Kaur and Ruchika Malhotra
University School of Information Technology, Guru Gobind Singh Indraprastha University, Kashmere Gate, Delhi 110006, India

Abstract

The importance of software measurement is increasing leading to development of new measurement techniques. As the development of object-oriented software is rising, more and more metrics are being defined for object-oriented languages. Many metrics have been proposed related to various object-oriented constructs like class, coupling, cohesion, inheritance, information hiding and polymorphism. The applicability of metrics developed by previous researchers is mostly limited to requirement, design and implementation phase. Exception handling is a desirable feature of software that leads to robust design and must be measured. This research addresses this need and introduces a new set of design metrics for object-oriented code. Two metrics are developed that measure the amount of robustness included in the code. The metrics are analytically evaluated against Weyuker's proposed set of nine axioms. These set of metrics are calculated and analyzed for standard projects and accordingly ways in which project managers can utilize these metrics are suggested.

1 INTRODUCTION

Programs fail mainly for two reasons: logic errors in the code and exception failures. Exception failures occur when a program is prevented by unexpected circumstances from providing its specified service. Exception failures can account for up to two-thirds of system crashes [Cristian95]; hence, are worthy of serious attention. On 4th June 1996, maiden flight 501 of the European Space Agency's new Ariane 5 heavy-lift rocket, developed at a cost of $7000 M over a 10 year period, ended in failure, after 39 seconds of its launch [Aggarwal01, Maxion98]. The problem was identified as a software exception caused during execution of a data conversion from 64-bit floating point to 16-bit format; the number was too big, so that an overflow error resulted after 36.7 seconds. This resulted in an operand error. The data conversion instructions were not protected from causing an operand error. No justification was found for not making the software robust.

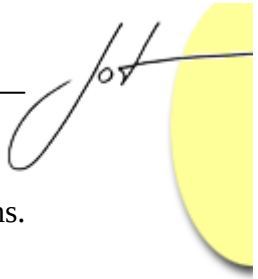
Cite this article as follows: K.K.Aggarwal, Yogesh Singh, Arvinder Kaur, Ruchika Malhotra,: "Software Design Metrics for Object-Oriented Software", in *Journal of Object Technology*, vol. 6. no. 1, January-February, pp. 121-138 http://www.jot.fm/issues/issue_2007_01/article4

Software is robust if it behaves “reasonably”, even in circumstances that were not anticipated in the requirement specification-for example, when it encounters incorrect input data or some hardware malfunction (say a disk crash). A program that assumes perfect input and generates an unrecoverable run-time error as soon as the user inadvertently types an incorrect command would not be robust. It might be correct, though, if the requirement specification does not state what the action should be upon entry of an incorrect command. It implies that software should be capable of diagnosing certain classes of errors. Robust design dictates error conditions to be anticipated and error-handling paths to be set up to re-route or cleanly terminate processing, when an error does occur. This is achieved after using exception-handling mechanism. A number of metrics have been proposed by researchers [Braind98, Briand99, Lorenz94, Harrison98, Chidamber94, Chhabra04], which measure the desirable characteristics of software. Exception handling is also a desirable software feature that leads to robust design and must be measured. A set of metrics has been proposed in this paper, to measure the robustness of design. The metrics are proposed, analyzed and evaluated on sample data set. The effect of using error handling functions in place of exception handling mechanism is also studied.

The paper is organized into following sections: Section 2 gives introduction to exception handling. Section 3 gives overview of proposed software metrics and formally defines them. Section 4 introduces various techniques available to find number of possible exceptions. Section 5 states Weyuker’s properties and evaluates the metrics on these properties. A brief description of the sources from which the empirical data is collected and then analysis of metrics based on this data has been done in section 6. The discussion of work carried is presented in section 7.

2 EXCEPTION HANDLING FUNDAMENTALS

Exceptional conditions are any unexpected occurrences that are not accounted for in a system's normal operation. Many different types of conditions can cause exceptions, depending on the specific program under consideration. Examples of such conditions include trying to divide by zero, stack overflow/underflow, array index out of bound, illegal use of null pointer reference, type mismatch, wrong command-line argument, security violation, and invalid data returned from another program. Failure due to exceptional conditions is a serious problem, not only in mission-critical applications, but also in commercial software systems and custom, home-developed code where quick, accurate results are essential. Programs are often logically correct but, nevertheless, fail due to mishandled exceptions. Some sort of error handling mechanism can be used to deal with such exceptional conditions. Error handling mechanism helps in constructing a robust system and reduces the cost to failure. In OO languages such as C++ and Java, this error handling mechanism is called exception handling. In computer languages that do not support exception handling, errors must be checked and handled manually through error codes, which is quite troublesome and cumbersome process. Exception handling is



the method of building a system to detect and recover from exceptional conditions. Reducing the occurrence of exception failure is beneficial for several reasons:

1. Software would be more robust, resulting in lower operating cost and higher availability.
2. A substantial percentage of security vulnerabilities would be removed. The computer security community has observed common mechanisms (e.g., buffer overflow) that cause exceptions and security vulnerabilities simultaneously and claim that eliminating exception failures would remove about 50 percent of security vulnerabilities [Maxion00].

Exception Handling Model in Java and C++ consists of three constructs [Venugopal97]:

Try: The block of code in which the exception can occur is specified in try block. There can be nested try blocks.

Catch: In catch block, the action to be taken if any exception occurs is specified. In some cases, more than one exception can be raised by a single piece of code. In such cases more than one catch block can be specified. Each catch clause catches a different type of exception.

Throw: It is used to manually raise an exception. The innermost try block in which the exception is raised is used to select the catch block that specify action to be taken when the exception occurs.

Java adds two more constructs finally and throws:

Throws: An exception thrown out of a method is specified by throws.

Finally: It contains code that must be executed before a method return.

Syntax of Try and Catch block is as follows:

```
try {
  . . .
} catch ( . . . ) {
  . . .
} catch ( . . . ) {
  . . .
} . . .
```

Java defines several exception classes and also allows the programmer to create new exception classes to handle situations specific to particular application. Exceptions can be generated by runtime system or they can be manually generated by the code [Schildt01].

Some popular runtime exceptions are: arithmetic error e.g., divides by zero, array index out of bound, assignment of an array element of incompatible type, creation of an array with negative index, invalid use of null reference etc. Exceptions generated manually can be invalid range of an input variable, incorrect date format and so on. These exceptions can be specific to an application.

3 DESIGN MEASURES

Two metrics are defined here that measures the robustness of the software by measuring the amount of exceptions considered while designing the software. If the developer includes exception handling, few methods in each class consist of try blocks to monitor code as well as catch blocks, which specify action to be taken if an exception occurs. The metrics are defined as follows:

Metric 1: Number of Catch Blocks per Class (NCBC)

Definition: Consider a class K_1, with methods $M_1,\ldots\ldots..M_n$. Let each method have $C_1,\ldots\ldots..C_m$ catch blocks. Then it is defined as the ratio of catch block in a class to the total number of possible catch blocks in a class:

$$\text{NCBC} = \frac{\sum_{i=1}^{n}\sum_{j=1}^{m} C_{ij}}{\sum_{i=1}^{n}\sum_{k=1}^{l} C_{ik}} \times 100 \qquad (1)$$

where n = Number of Methods in a class
m = Number of Catch Blocks in a Method
C_{ij} is j^{th} Catch Block in i^{th} Method
C_{ik} is k^{th} Catch Block in i^{th} Method
l = Maximum Number of possible Catch Blocks in a Method

The metric counts the percentage of the catch blocks in each method of the class. The NCBC denominator represents the maximum number of possible catch blocks for class C_{ik}. This would be the case where all possible exceptions in C_{ik} have a corresponding catch block to handle these exceptions. Thus the value of denominator will be equal to maximum possible exceptions in a class. The metric is applied on the example code shown in Figure 1.

Let number of possible catch blocks in A and B is 3 and 6 respectively.
The values of proposed metric NCBC for classes A and B are:
NCBC (A) = 2/3
NCBC (B) = 4/6

```
class A{
        public void M1(){
                try{//monitor block of code
                        try{//nested try block
                                …}
                        catch(…)
                        {       //action to be taken if exception occurs
```

```
                    …}
                catch(…)
                {       //action to be taken if exception occurs
                …}
                }//end of upper try block
        }//end of function M1()
}
class B {
        public void M2(){
                try{ //monitor block of code  ….}
                catch(…){
              //action to be taken if exception occurs…}
            try{ //monitor block of code
                    ….}
                catch(…){
              //action to be taken if exception occurs
                …}
        }//end of function M2()
        public void M3(){
                try{ //monitor block of code
                ….}
                //each catch block catches a different type of exception
            catch(…){
              //action to be taken if exception occurs
                …}
                catch(…){
              //action to be taken if exception occurs
                …}
        }//end of function M3()
}
```

Figure 1: Source code for calculating metric NCBC

Metric 2: Exception Handling Factor (EHF)

Definition: It is formally defined as the ratio of number of exception classes to the total number of possible exception classes in software:

$$EHF = \frac{\text{Number of Exception Classes}}{\text{Total Number of Possible Exception Classes}} \times 100 \qquad (2)$$

Number of exception classes is the count of exceptions covered in a system. The exception class is passed as an argument to the catch construct as *type of argument arg*. This type of argument specifies types of exception classes. The count of exception classes is the total number of exception classes that occur irrespective of the number of times the same exception class occurs. For example, ArithmeticException class in Java can be used as argument in multiple catch blocks but to calculate EHF it will be counted once only.

The metric is applied on the example code shown in Figure 2.

In the example code, Java Build in exception used in classes A and B are :

1. ArithmeticException
2. ArrayIndexOutOfBoundsException
3. IllegalAccessException

Subclass of Exception class created in class B is:

4. MyException

Thus, Number of Exception classes = 4 and if we assume Number of possible exception classes=6 (class A and class B).

Exception Handling Factor (EHF) = 4/6.

```
class A{
        public void A1(){
                try{ //monitor block of code
                …}
                catch(ArithmeticException e){          //catch if exception occurs
                        System.out.println("Divide by 0:"+e);
                }
            try{ //monitor block of code
                …}
                catch(ArrayIndexOutOfBoundsException e){
            //catch if exception occurs
                        System.out.println("Array out of bound exception:"+e);
                }
        }//end of method A1()
        public void A2(){
                try{ //monitor block of code
                …}
                catch(IllegalAccessException e){       //catch if exception occurs
                        System.out.println("Access to the class id denied:"+e);
                }
            try{ //monitor block of code
                …}
                catch(ArrayIndexOutOfBoundsException e){
            //catch if exception occur
                        System.out.println("Array out of bound exception:"+e);
```

```
                }
        }//end of method A2()
}
class B{
        public void B1(){
                try{ //monitor block of code
                …}
                catch(ArithmeticException e){          //catch if exception occurs
                        System.out.println("Divide by 0:"+e);
                }
            try{ //monitor block of code
                …}
                catch(MyException e){          //catch if exception occurs
                        //Manually created exception class i.e. custom exception type
                        System.out.println("Array out of bound exception:"+e);      }
        }//end of method B1()
}
```

Figure 2: Source code for calculating metric EHF

4 TECHNIQUES TO FIND POSSIBLE EXCEPTIONS

To apply above metrics, there is need to calculate the number of possible exceptions. The number of possible exceptions can be counted by using techniques such as Collaboration, N-Version diversity and Dependability cases [Maxion00]. Collaboration technique states that few people work together to find the possible exception cases. N-version diversity technique states that more than one person independently finds all possible exceptions and then finally these exceptions are merged. Another technique called Dependability cases comprises of an organizing framework for thinking about exceptions and the conditions under which they occur. For this purpose, hazard analysis, fault trees or fishbone diagrams can be used. Figure 3 shows an example roughly in the shape of a fish, which shows exceptions that could be encountered in a software system. At the "head" of the fishbone are the exception failures, which should be avoided. The ribs are labeled with categories of events that cause exception failures; the events within each rib are examples of specific causes. For example, the rib labeled "computational exception" lists uninitialized variable as an exemplar. The fishbone in Figure 3 lays out a set of exception causes, which commonly occur in a program. A complete fishbone for a particular program would require an in-depth analysis of the faults to which it is susceptible and a corresponding modification of the fishbone. These techniques should be used by both designers to cover maximum exceptions during design as well as by testing team to find all possible exceptions.

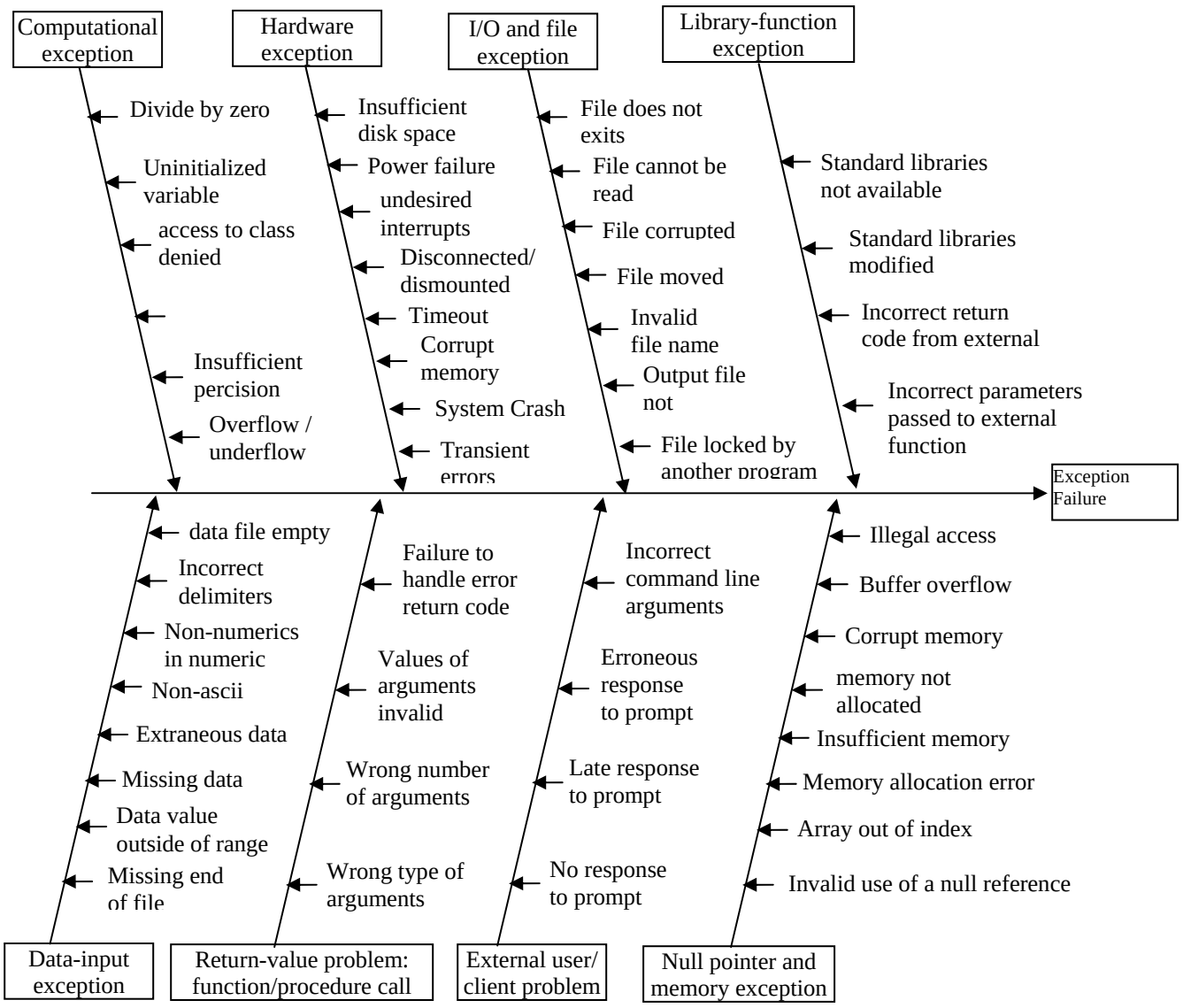
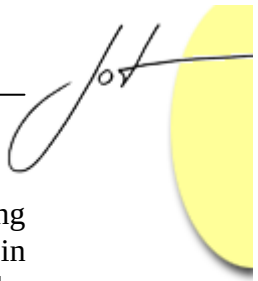


Figure 3: Fishbone diagram showing exception categories and exemplars [reprinted from IEEE Trans. on Software Engineering, vol 26, No. 9,September 2000]

5 AXIOMATIC EVALUATION OF METRICS ON WEYUKER'S PROPERTIES

Several researchers have recommended properties that software metrics should possess to increase their usefulness. For instance, Basili and Reiter suggest that metrics should be sensitive to externally observable differences in the development environment, and must



correspond to notions about the differences between the software artifacts being measured [Basili79]. However, most recommended properties tend to be informal in evaluation of metrics. It is always desirable to have a formal set of criteria with which the proposed metrics can be evaluated. Weyuker has developed a formal list of properties for software metrics and has evaluated number of existing software metrics against these properties [Weyuker98]. Although many authors have criticized [Zuse90, Fenton91] this approach but still it is a widely known formal analytical approach and have been referred by many notable authors for evaluating their measures [Henderson96, Chidamber94]. The similar approach has been followed in this paper for analysis of proposed metrics.

Weyuker's (1988) first four properties address how sensitive and discriminative the metric is. The fifth property requires that if two classes are combined their metric value should be greater than metric value of each individual class. The sixth property addresses the interaction between two programs/classes. It implies that interaction between program/class A and program/class B is different than interaction between program/class C and program/class B given that interaction between program/class A and program/class C is same. The seventh property requires that a measure be sensitive to statement order within a program/class. The eighth property requires that renaming of variables does not affect the value of a measure. Last property states that the sum of the metric values of a program/class could be less than the metric value of the program/class when considered as a whole [Henderson96].

Let u be metric of program/class P and Q

Property 1: This property states that

$$(\exists P),(\exists Q)(u(P) \neq u(Q))$$

It ensures that no measure rates all program/class to be of same metric value.

Property 2: Let c be a nonnegative number. Then there are finite numbers of program/class with metric c. This property ensures that there is sufficient resolution in the measurement scale to be useful.

Property 3: There are distinct program/class P and Q such that $u(P) = u(Q)$.

Property 4: For object-oriented system, two program/class having the same functionality could have different values.

$$(\exists P)(\exists Q)(P \equiv Q \text{ and } u(P) \neq u(Q))$$

Property 5: When two program/class are concatenated, their metric should be greater than the metrics of each of the parts.

$$(\forall P)(\forall Q)\ (u(P) \leq u(P+Q) \text{ and } u(Q) \leq u(P+Q))$$

Property 6: This property suggests non-equivalence of interaction. If there are two program/class bodies of equal metric value which, when separately concatenated to a same third program/class, yield program/class of different metric value.

For program/class P, Q, R

$$(\exists P)(\exists Q)(\exists R)\ (u(P) = u(Q) \text{ and } u(P+R) \neq u(Q+R))$$

Property 7: This property is not applicable for object-oriented metrics [Chidamber94].

Property 8: It specifies that "if P is a renaming of Q; then $u(P) = u(Q)$"

Property 9: This property is not applicable for object-oriented metrics [Chidamber94].

Axiomatic Evaluation of NCBC on Weyuker's Properties

Property 1 is satisfied by the proposed metric as two object-oriented classes P and Q can always differ in catch block count. Let $\mu(P) = n$ then there are finite numbers of class with metric value n ensuring sufficient resolution. Thus Property 2 is also satisfied. There is probability (non zero) that $u(P) = u(Q)$. Therefore Property 3 is satisfied. The decision of implementing catch blocks in methods per class is totally independent of functionality i.e. logic used in the class. This satisfies Property 4.

Let $u(P) = p$ and $u(Q) = q$.

Then $u(P + Q) = p + q - m$,

where m is the catch blocks common to a class. In some cases

$$u(P) \geq u(P + Q)$$
$$u(Q) \geq u(P + Q)$$

This does not satisfies Property 5. Now, Let $u(P) = p$ and $u(Q) = p$, and a class R has number of catch blocks r such that it has m catch blocks common with class P and n common with Q, where $m \neq n$.

$$u(P + R) = p + r - m$$
$$u(Q + R) = p + r - n$$

This shows that $u(P + R) \neq u(Q + R)$ and Property 6 is satisfied. The renaming of class or method does not affect the value of metric proposed hence satisfying Property 8.

Axiomatic Evaluation of EHF on Weyuker's Properties

Let $u(P)$ and $u(Q)$ be metric of Program P and Q.

Property 1 is satisfied by the proposed metric, as two object-oriented program can always differ in exception value. There can be many different values of the proposed metric. This satisfies Property 2. The proposed metric also satisfies Property 3. It is assumed that classes are not dependent on each other in system P, then the metric of P will be nothing but the summation of exception classes, which can be same for another different unrelated system Q. The choice of exception classes is design decisions, which does not depend upon the functionality of software, therefore Property 4 is satisfied.

Let $u(P) = p$ and $u(Q) = q$.

Then $u(P + Q) = p + q - m$.

Let y and b be total number of possible exception classes in program P and program Q. Let x and a be number of exception classes in program P and program Q then,

$$u(P) = x / y \text{ and } u(Q) = a / b$$
$$u(P + Q) = (x + a) / (y + b)$$

where $x + a$ is the number of exception classes combined by concatenating program P and program Q.

$y + b$ is the number of possible exception classes combined by concatenating program P and program Q.

Thus, it is possible that, $u(P) \geq u(P + Q)$ or $u(Q) \geq (P + Q)$

Therefore, Property 5 is not satisfied. Now, let P and Q be two programs such that $u(P) = u(Q) = p$ and let R be another program with EHF value $u(R) = r$. Then, $u(P + Q) = p + r - m$, similarly $u(Q + R) = p + r - n$. This shows that $u(P + R) \neq u(Q + R)$. Therefore, Property 6 is satisfied. The renaming of systems P and Q does not affect the metric values. Hence, Property 8 is satisfied.

Thus all properties except 5 are satisfied by both metrics. The property 5 is not satisfied by many earlier metrics like DIT of Chidamber and Kemerer [Chidamber94].

6 ANALYSIS OF METRICS ON EMPIRICAL DATA

To analyze proposed metrics their values are computed for five different projects out of which case studies and design of three projects are presented in book "Introduction to Object-Oriented Analysis and Design" and "Object-Oriented and Classical Software Engineering", authored by S.R. Schach. Their respective codes are available on Internet at [Schach02, Schach04]. The projects are developed in Java language and will be referred herein after as Project 1, Project 2, and Project 3. The proposed metrics is also applied on another project referred here as Project 4. Project 4 enables intranet users to book holidays and uses Java programming for developing the system. Project 5 is same version of Project 1 but uses validation functions instead of exception handling in some cases whereas in other cases probable faults are not covered by the code.

Analysis of Metric NCBC on Collected Data

The summary statistics and histogram for Project 1 to Project 4 are shown below. To indicate the metric value the histograms of the NCBC value are given. The percentages of classes are shown on Y-axis. In the first column of Figure 4(a) the classes for which the NCBC value was 0 percent is shown. The other columns indicate the classes for which metric NCBC equal to 37, 57, 71, 89 and 100 percent respectively.

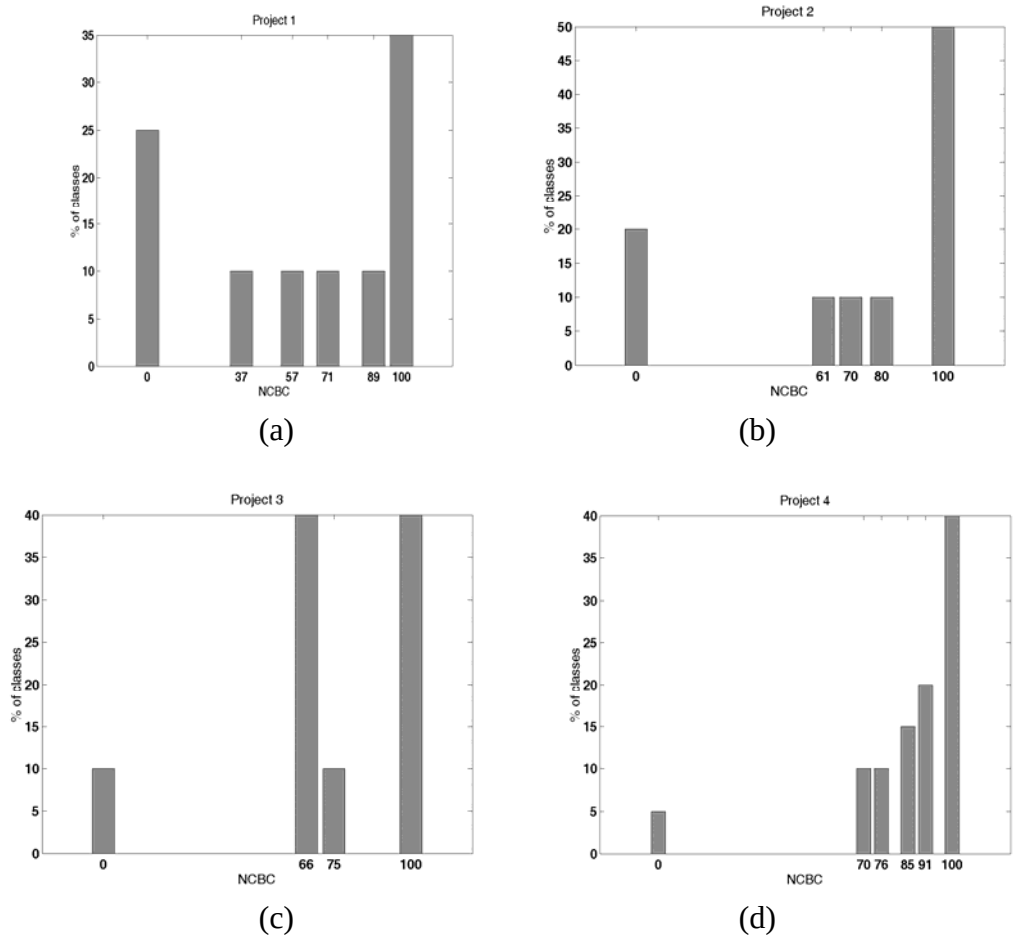


Figure 4: Number of Catch Blocks per Class for (a) Project 1 (b) Project 2 (c) Project 3 (d) Project 4

Similar histograms are drawn for projects 2 to 4. The histogram for Project 5 is not shown, as the metric values for all the classes in this project are zero. In Table 1 the min., max. and mean values for NCBC metric applied on Project 1 to Project 4 are shown.

	Metric	**Project 1**	**Project 2**	**Project 3**	**Project 4**
Min	NCBC	0	0	0	0
Max	NCBC	100	100	100	100
Mean	NCBC	53.33	62.2	70.33	80.5

Table 1: Data Statistics for NCBC metric

There is similarity in the distribution of the metric value in all four projects; most of the classes have NCBC values 0 or 100. This shows that one of the advantages of using information from metrics is that a clear picture of system emerges with greater insight into class implementation. The reason for 0 metric value could be that developers do not use exception-handling mechanism in practice and/or exceptions is not taken into

consideration. The data collected from projects gives the project team information about number of expected faults covered in each class.

Project 5 is the same version of Project 1 and does not use exception handling at all. We have considered Project 5 and Project 1 for gaining further insight, to find the effect of using validation functions instead of exception handling on lines of code (LOC). The following observations are made when NCBC metric applied on Project 1 and Project 5. In Project 5 the value of NCBC is zero. One of the exceptions handled by the classes in these projects was to check the format of date at various points. Project 1 used exception handler indicated by catch keyword to take action if the format of date entered was incorrect whereas in Project 5 function called valid_date() was used for validating the date format. By adding valid_date() function, LOC for this class increased to almost three times. Increase in the size of code increases unnecessary effort and complexity. Figure 5 compares absolute LOC values of classes in Project 1 and Project 5. The percentage of LOC increased in classes, which include validation functions instead of using exception handling, is shown in Figure 6. The increase in LOC varies from 7- 41%. Some exceptions like "File cannot be read" were not caught in classes at Project 5. At class testing time it was observed that test cases in Project 5 had increased as compared to Project 1. Thus classes with less NCBC values will have more LOC count in including error handling mechanism in classes.

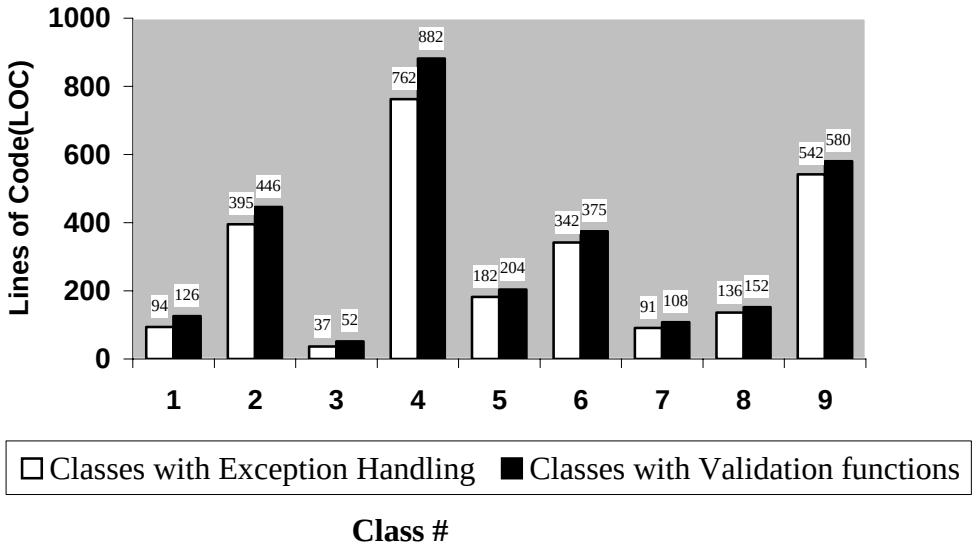


Figure 5: Comparison of LOC in classes of Project 1 and Project 5

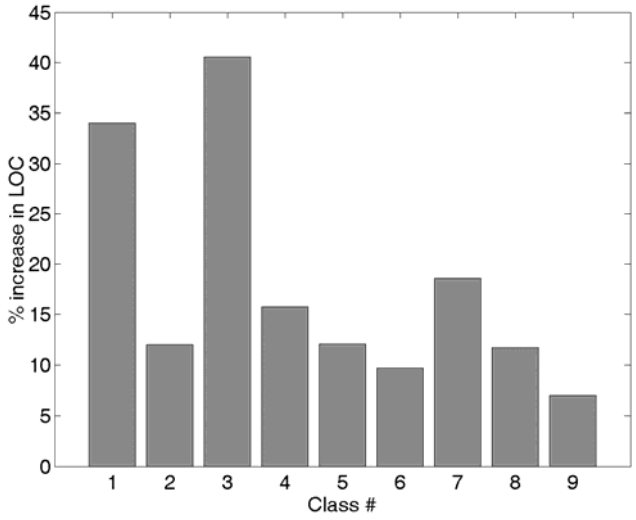


Figure 6: Percentage increases in LOC in Project 5

Analysis of Metric EHF on Collected Data

The summary statistics and histogram for Project 1 to Project 5 are shown in Figure 7. The bar graph as in Figure 7 shows the percentage of EHF value calculated for all projects taken for analysis. In Table 2 the min., max. and mean values for EHF metric applied on Project 1 to Project 5 are shown.

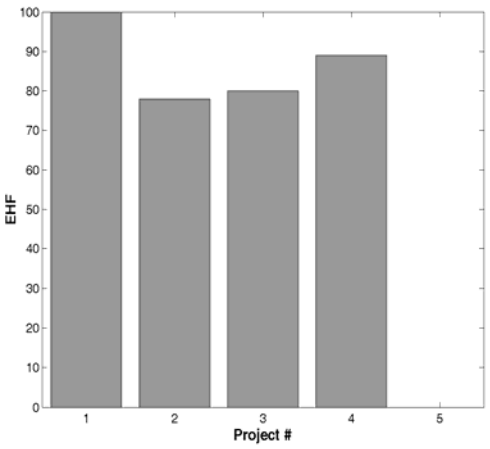


Figure 7: Bar Chart of Exception Handling Factor

Metric	Min	Max	Mean
EHF	0	100	72.6

Table 2: Data Statistics for EHF metric

Exceptions in Project 1	Validation functions in Project 5
EOFException	CheckEOF
Exception	FileNotfound
ParseException	CheckDate

Table 3: Exception and Validation functions used in Projects

The EHF value in Project 1 is 100 % whereas in Project 5 it is 0 % (Figure 7). The reason for abnormal termination and invalid output would be least revealed in Project 5 whereas Project 1 is more helpful in finding faults. Project 1 allows project team to gather information about cause of exception during validation testing. Also the metric gives project team information about the amount of error-handling effort already included in the code at the system level. Table 3 shows some classes included in Project 1 and corresponding validation functions in Project 5. In Project 5 EHF value indicates that more development effort is needed in adding functionality to incorporate some validations so that exceptions (runtime or manual) do not cause invalid termination or/and output. Some validations have not been added in Project 5 so the system fails at these points during testing. These failures were revealed in later stages of software development, which will increase the total cost. If these exception/faults had been included during development time the cost would have been much less. In Project 5 the more effort would be required in tracing the causes of abnormal termination. The metric value in Project 1 gives the project team information about causes of occurrences of failure by displaying diagnostic messages. In Project 2 and Project 3 more exception classes were covered as compared to Project 5 but less than Project 1 and Project 4. Therefore, the project team in Project 1 and Project 4 would require less development effort as compared to Project 2 and Project 3.

7 DISCUSSIONS

Two new measures have been proposed and axiomatically evaluated against Weyuker's properties. It has been observed that higher the value of proposed metric, more robust will be the class/system. The small values of these metrics indicate that either developer have not made sufficient effort in including code for possible exceptions or they are not experienced to look for possible exceptions. These metrics should be applied and evaluated by review teams. The effect of using error handling functions in place of exception handling mechanism is also studied. The result shows that the size and hence complexity of the software increases with error handling functions. Also more development effort is required in adding error functions in place of exception mechanism.

REFERENCES

[Aggarwal01] K.K Aggarwal, Yogesh Singh, *Software Engineering*. New Age International, Publishers, 2001.

[Basili79] V.Basili, R.Reiter. Evaluating Automable Measures of Software Models. IEEE, Workshop on Quantitative Software Models, 107-116, 1979.

[Briand98] L.Briand, W.Daly and J. Wust, Unified Framework for Cohesion Measurement in Object-Oriented Systems. *Empirical Software Engineering*, 3, pp. 65-117, 1998.

[Briand99] L.Briand , W.Daly and J. Wust, A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on software Engineering*, 25, pp.91-121, 1999.

[Chidamber94] S.R.Chidamber and C.F.Kamerer, A metrics Suite for Object-Oriented Design. *IEEE Trans. Software Engineering*, vol. SE-20, no.6, 476-493, 1994.

[Cristian95] F. Cristian, Exception Handling and Tolerance of Software Faults, Software Fault Tolerance, M. R. Lyu, ed., Wiley, Chichester, pp.81-107, 1995.

[Fenton96] N.Fenton et al, *Software Metrics: A Rigorous and practical approach*. International Thomson Computer Press, 1996.

[Harrison98] R.Harrison, S.J.Counsell, and R.V.Nithi, An Evaluation of MOOD set of Object Oriented Software Metrics. *IEEE Trans. Software Engineering*, vol. SE-24, no.6, pp. 491-496, June 1998.

[Henderson96] B.Henderson-sellers, *Object-Oriented Metrics, Measures of Complexity*. Prentice Hall, 1996.

[Jitender04] Jitender Kumar Chhabra, K. K. Aggarwal and Yogesh Singh, Measurement of, object-oriented software spatial complexity. Information and Software Technology, Volume 46, Issue 10(2004) 689-699

[Lorenz94] M.Lorenz, and J.Kidd, *Object-Oriented Software Metrics*. Prentice-Hall, 1994.

[Maxion00] R. A. Maxion and R. T. Olszewski, Eliminating Exception Handling Errors with Dependability Cases: A Comparative, Empirical Study. *IEEE Transactions on Software Engineering*, vol. 26, no. 9, pp.888-906, 2000.

[Maxion98] R. A. Maxion and R. T. Olszewski, “Improving Software Robustness With dependability cases”, Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing, pp.346-355, June 1998.

[Pressman01] R.Pressman, *Software Engineering*. McGraw-Hill, Boston, 2001.

[Schach04] S.R.Schach, *Introduction to Object-Oriented Analysis and Design*. Tata, McGraw-Hill, 2004, Site: http://highered.mcgraw-hill.com/sites/0072826460/student_view0/case_studies.html.

[Schach02] S.R Schach, *Object-Oriented and Classical Software Engineering*. Tata, McGraw-Hill, 2002, Site: http://www.mhhe.com/engcs/compsci/schach5/student/airgourmet.java.java.

[Schildt01] H.Schildt, *Java 2: The Complete Reference* Tata McGraw-Hill, 2001.

[Venugopal97] K.R. Venugopal, Rajkumar, T.Ravishankar, *Mastering C++*. Tata McGraw Hill,1997.

[Weyunker98] E.Weyuker, Evaluating Software Complexity Measures. *IEEE Transactions on Software Engineering*, vol 14, pp.1357-1365, 1998.

[Zuse90] H.Zuse, *Software Complexity: Measures and Methods*. Walter de Gruyter, Berlin, 1990.

About the authors

K. K. Aggarwal is vice chancellor at the Guru Gobind Singh Indraprastha University, India. He received his doctorate from Kurushetra University. He was president of the Institution of Electronics and Telecommunication Engineers (IETE) from 2002 through 2004. Recently he was awarded "Delhi Ratan Bhuddhijeevi Samman" by the All India Conference of Intellectuals (AICI). Prof. Aggarwal has written few books and many of his articles have appeared in several books published by IEEE of USA. He is coauthor of a book on software engineering and has published more than 300 publications in national and international journals and conferences. He can be reached by e-mail at kka@ipu.edu.

Yogesh Singh is a professor with the University School of Information Technology and the School of Engineering and Technology, Guru Gobind Singh Indraprastha University, Kashmere Gate, India. He received his master's degree and doctorate from the National Institute of Technology, Kurukshetra. His area of research is Software Engineering focusing on Planning, Testing, Metrics and Neural Networks. He is coauthor of a book on software engineering, and is a Fellow of IETE and member of IEEE. He has more than 150 publications in international and national journals and conferences. Singh can be contacted by e-mail at ys66@rediffmail.com.

Arvinder Kaur is a Reader with the University School of Information Technology. She obtained her doctorate from Guru Gobind Singh Indraprastha University and her master's degree in computer science from Thapar Institute of Engg. and Tech. Her research interests include software engineering, object-oriented software engineering, software metrics, microprocessors, and operating systems. She is also a lifetime member of ISTE and CSI. Kaur has published more than 30 research papers in national and international journals and conferences. Her paper titled "Analysis of object oriented Metrics" was published as a chapter in the book *Innovations in Software Measurement* (Shaker -Verlag, Aachen 2005). She can be reached by e-mail at arvinderkaurtakkar@yahoo.com.

Ruchika Malhotra is a research scholar with the University School of Information Technology, Guru Gobind Singh Indraprastha University, India. She is working as a visiting faculty with Amity Institute of Information Technology, India. She received her master's degree in software engineering from the University School of Information Technology, Guru Gobind Singh Indraprastha University, India. Her research interests are in improving software quality, statistical and adaptive prediction models for software metrics, neural nets modeling, and the definition and validation of software metrics. She has more than 10 publications in international journals and conferences. Her paper titled "Analysis of object oriented Metrics" was published as a chapter in the book *Innovations in Software Measurement* (Shaker -Verlag, Aachen 2005). She can be contacted by e-mail at ruchikamalhotra2004@yahoo.com.