ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 1, Issue 7, September 2014

Automated Defect Recognition Methodology by Mistreatment Digital Image Process

Kannan Subramanian

Dept. of MCA, Bharath University, Chennai-600073, India

ABSTRACT: Quality control is an important issue in the ceramic tile industry. On the other hand maintaining the rate of production with respect to time is also a major issue in ceramic tile manufacturing. Again, price of ceramic tiles also depends on purity of texture, accuracy of color, shape etc. Considering this criteria, an automated defect detection and classification technique has been proposed in this report that can have ensured the better quality of tiles in manufacturing process as well as production rate. Our proposed method plays an important role in ceramic tiles industries to detect the defects and to control the quality of ceramic tiles. This automated classification method helps us to acquire knowledge about the pattern of defect within a very short period of time and also to decide about the recovery process so that the defected tiles may not be mixed with the fresh tiles.

KEYWORDS*-:* Quality Control; Pattern of Defect; Defected Ceramic Tiles; Fresh Tiles

I. Selection of Programming Language

Selection of a particular language should be made based on overall goals of the project. Before starting an important project, it would be important to create several independent testing code programs in various languages. This will describe which one will be best for your task and that will go to save the time over selecting a particular language without unnecessary headache. Another approach is to divide project into separate modules or small projects with different languages for each module and selecting the one which may be best suited for the project. Some programming languages need high end configurations whereas other needs common configurations, it is very important to consider the user's machine configuration while selecting language. For selection of particular language, the developer needs to consider the platform dependency, employees who know about the language, popularity of the language, and unanimity of the project team about language. The selected language must be easily understandable by user as well as the developer. It is expected to select the language which can contribute to save time, money, efforts and suitability in project.

II. What is Complexity?

The first problem encountered when attempting to understand program complexity is to define what it means for a program to be complex. Basili defines complexity as a measure of the resources expended by a system while interacting with a piece of software to perform a given task [1]. If the interacting system is a computer, then complexity is defined by the execution time and storage required to perform the computation. If the interacting system is a programmer, then complexity is defined by the difficulty of performing tasks such as coding, debugging, testing, or modifying the software. The term software complexity is often applied to the interaction between a program and a programmer working on some programming task.

Usually these measures are based on program code disregarding comments and stylistic attributes such as indentation and naming conventions. Measures typically depend on program size, control structure, or the nature of module interfaces. The most widely known measures are those devised by Halstead and his colleagues that are collectively known as software science [2].

The Halstead measures are functions of the number of operators and operands in the program. The major components of software science are
n_1, the number of unique operators,
n_2, the number of unique operands,
N_1, the total number of operators,
N_2, the total number of operands.

Halstead defined the volume, V, of a program to be
$V = (N_1 + N_2) \log2 (n_1 + n_2)$

and program difficulty, D, to be
$D = (n_1 X N_2) / 2n_2$

Halstead derived a number of other measures. The most extensively studied of these is an estimate of the effort, E, required implementing a program:
$E = D X V$

III. The Properties of Software Complexity Measures

Several properties of measures determine the way in which the measure can be used.

A. strength

If software complexity measures are to be used to evaluate programs, then it is important to consider the measure's responsiveness to program modifications. Not only should the measure be shown to reliably predict the complexity of the software, but programming techniques that minimize the measure should be examined to assure that reductions in the measure consistently produce improvements in the program. In particular, it should not be possible to reduce the measure through incidental modifications of the program. Also, programming techniques that modify the program in a desirable way with respect to one property must not produce an undesirable change in another property as a side effect. A robust measure of software complexity is sensitive to the underlying complexity of the program and cannot be misdirected by incidental modifications to the program.

Several authors have examined the relationship between complexity measures and commonly accepted axioms for good programming [3, 4, 5, 6]. Their strategy has been to study how complexity measures are affected by following maxims of good programming style. Halstead's E, the cyclomatic number, and the number of lines have been examined for their responsiveness to modularization, the use of temporary variables, initialization procedures, and such. The results of these analyses do not provide strong support for these measures. For some classes of programs, some measures are reduced by some good programming practices.

B. Normativeness

The interpretation of complexity measurements is facilitated if the metric provides a norm against which measurements can be compared. Without such a standard, it is meaningless to apply the metric to programs in isolation. To judge whether or not a program is overly complex, a norm that identifies some acceptable level of complexity must be specified.

C. Specificity

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 2, February 2014

Software complexity analysis may provide an review tool that can be used during program development and testing. Designers and programmers could use the measure to find insufficiency in program construction. A complexity measure might also be used as a guide for testing and maintenance efforts. The degree to which a measure is able to perform these functions will depend on how well it specifies what its contribution to the complexity of a program.

D. Prescriptiveness

If software complexity measures are to prove useful in the restraint of program complexity, then they must not only index the level of a program's complexity, but also should suggest methods to reduce the amount of complexity. A measure could prescribe techniques to avoid excess complexity as well as direct modification of overly complex programs already written.

IV. TASK COMPLEXITY

A. Task Size

We measured task size as the number of thousands of lines of software instructions. As the software product was written in one programming language, lines of code provide a reasonable measure of software size. Naturally, we expect that more software instructions will take longer to develop.

B. Structural Complexity

Structural complexity can be evaluated by measuring characteristics that make programming language difficult to understand and change [7]. As the number of modules affected increases, it becomes more difficult to understand how the parts being modified will affect other parts of the system which increases the amount of information that developers need to process to implement

V. MAJOR CAUSES OF PROJECT FAILURE

Project development in the IT service industry is the crucial factor because the whole IT industry profit depends upon the project's success. Figure 1 shows major causes of IT project failure, in which 29% of IT project fail due the inadequate coordination of resources in the project where programming language and database management system is one the major resources in the project. While discussing the project development some issues need to be sorted out in the early stages because IT project's time duration is very long and hence if we do not settle the issues early in the development stage then it become unavoidable.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 2, February 2014

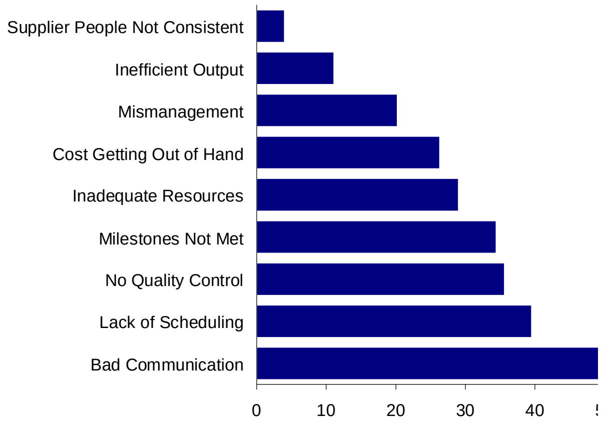

Figure 1 Major Causes of Project Failure

V. Testing

Once a measure has been developed, it must be tested to be sure it actually measures what it purports to measure.

A. Experimental Design

Researchers attempting to validate measures of software complexity face a methodological morass. An enormous number of parameters may influence the outcome of an experiment. Subject selection, programming language, programming task, and the algorithms implemented can all profoundly affect both the nature of the results and the extent to which experimental observations will generalize to a larger set of programming environments. The problem is compounded by the uncertainty of how these parameters interact to determine programmer behavior. Worse yet, there are not even good methods to quantify parameters such as programmer ability and problem difficulty.

Once the programming environment is specified, the experimenter must devise a method to manipulate the independent variable-typically some program property. If research is conducted as a natural experiment (observing actual programs produced in a real work setting), then the problem is to find programs that differ only in the variables of interest. The difficulty of obtaining uncontrived programs that vary only in one or two dimensions should not be underestimated.

VI. Conclusion

The primary goal of the software developer is to develop a project which satisfies user’s need as well as the project will complete within time deadline. If the selection of programming language fail at initial stage in the project, then developer team need to go to basics and start the work again. It is wastage of resources for the organization. This paper provides guidelines for choosing a programming language for project development. If the developer working with unsuitable programming language then project will definitely not function properly and due to which the user is not satisfied with project. Complexity measures currently available provide only a crude index of software complexity. Advancements are likely to come slowly as programming behavior becomes better understood. Users of complexity measures must be aware of the limitations of these measures and approach their applications cautiously. Before a measure is incorporated into a programming environment, the user should be sure that the measure is appropriate for the task at hand. The measure must possess the properties demanded by the use. Finally, users should always view complexity measurements with a critical eye.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization)

Vol. 2, Issue 2, February 2014

REFERENCES

1 Basili, V.R. Qualitative software complexity models: A summary. In Tutorial on Models and Methods for Software Management and Engineering. IEEE Computer Society Press, Los Alamitos, Calif., 1980.
2 Halstead, M.H. Elements of Software Science. Elsevier North-Holland, New York, 1977.
3 Baker, A.L. The use of software science in evaluating modularity concepts. IEEE Trans. Softw. Eng. SE-S, 2 (Mar. 1979), 110-120.
4 Baker, A.L., and Zweben, S.H. A comparison of measures of control flow complexity. IEEE Trans. Softw. Eng. SE-6,6 (Nov. 1980). 506-512.
5 Evangelist, W.M. Software complexity metric sensitivity to program structuring rules. I. Syst. Softw. 3, 3 (Sept. 1983), 231-243.
6 Gordon, R.D. Measuring improvements in program clarity. IEEE Trans. Softw. Eng. SE-5, 2 (Mar. 1979), 79-90.
7 Curtis, B., S. B. Sheppard, P. Milliman, M. A. Borst, T. Love. 1979. Measuring the psychological complexity of software maintenance tasks with the Halstead and Mccabe metrics. IEEE Trans. Software Engrg. 5(2) 96–104.