03 Jan 1982

A Theory Of Small Program Complexity

Kenneth I. Magel
Missouri University of Science and Technology

Recommended Citation

K. I. Magel, "A Theory Of Small Program Complexity," *ACM SIGPLAN Notices*, vol. 17, no. 3, pp. 37 - 45, Association for Computing Machinery, Jan 1982.
The definitive version is available at https://doi.org/10.1145/947912.947913

A Theory of Small Program Complexity

Kenneth Magel

University of Missouri-Rolla

Abstract

Small programs are those which are written and understood by one person. Large software systems usually consist of many small programs. The complexity of a small program is a prediction of how difficult it would be for someone to understand the program. This complexity depends on three factors: (1) the size and interelationships of the program itself; (2) the size and interelationships of the internal model of the program's purpose held by the person trying to understand the program; and (3) the complexity of the mapping between the model and the program. A theory of small program complexity based on these three factors is presented. The theory leads to several testable predictions. Experiments are described which test these predictions and whose results could verify or destroy the theory.

Small programs are those which are written by one person in approximately two weeks or less. They generally have fewer than two hundred executable source statements, often considerably fewer. Small programs are distinguished from larger programs in a very significant qualitative way as well as by size: a small program can be understood in its entirety by one person.

Belady and Lehman have treated a large software system as a statistical object made up of many small programs [1,2,3]. Just as thermodynamics can manipulate the properties of gases without considering the properties of the individual atoms in each gas, Belady and Lehman have developed several statistical laws based on observations of very large software systems. Nevertheless, to really understand how large software systems evolve, we must look at the small programs which form the parts of any large system.

Computer scientists who work on the complexity of small programs always seem to know what the term program complexity means, but they rarely are explicit about revealing their definitions. They might propose program complexity metrics, but different metrics yield different relative results for the same set of programs [4,5,6,7,8,9]. A new metric is justified on the grounds that it gives results consistent with modern precepts of good programming style [10,11].

A useful program complexity metric is one which predicts how difficult some future task will be. Difficult can mean either how long the task will take, how many resources it will require, or how successfully it will be preformed. A minimally useful metric could be one which when applied to two programs predicted on which the task would take longer than on the other, but neither how long nor the relative proportions of time required. This would be a comparative metric. A more useful metric

would indicate the relative proportions of time required. This would be a relative metric. The most useful metric would indicate how much time would be required. This would be an absolute metric.

There are at least four tasks for which predictions of required time, required resources, or probable success are needed:

(1) understanding the program as evidenced by answering questions on the program;
(2) maintaining the program;
(3) modifying the program;
(4) testing the program.

Understanding the program is necessary for the other three tasks. We will assume that all reasonable tasks depend on how easily and how well the program is understood.

The remainder of this paper has three sections. The first describes the features which existing program complexity metrics consider to contribute significantly to complexity. The second section presents the theory and some published data which can be interpreted to support it. The final section begins with some predictions which arise from the theory. Experiments are proposed to test those predictions. The types of results which would support or conflict with the theory are discussed.

Factors in Small Program Complexity

Many computer scientists have proposed program complexity metrics of limited scope. Each metric identifies a different set of syntactic features as contributing to complexity. The oldest and probably the most widely applied complexity metric is program size: the larger a program, the more difficult it is to understand. Several authors have pointed out that program size can be measured in many ways, e.g. lines of code including or not including comments and declarations versus number of statements versus number of lexical tokens or even characters [12]. Nevertheless, the very limited studies of complexity metrics as predictors of program understandability or maintainability indicate that program size does about as well as any other metric [13,14]. None of the metrics seem to do very well. They work as comparative metrics most of the time, but there are unexplained and unpredicted anomalies. There is no published evidence to indicate that any existing metric can do more than order programs. Size clearly is an important contributor to complexity, but not the only important contributor.

Barry Boehm and several colleagues at TRW did an extensive study in the early 1970's of characteristics of software quality [15]. The characteristics which they identified as being most significant include those which existing software complexity metrics have tried to measure. The characteristics identified by the TRW study as most important are:

(1) self-descriptiveness;
(2) completeness;
(3) accessability;
(4) communicativeness;

(5) device independence;
(6) consistency;
(7) structuredness;
(8) accuracy [15 pages 4-6 through 4-11].

These are explained in the rest of this section.

Self descriptiveness measures the extent to which a program provides its own documentation. Included are such characteristics as the degree to which variable names are descriptive of their purposes in the program and the use of header blocks of commentary for major sections of the program.

Completeness measures the degree to which the program explicitly checks for unexpected data or data which might cause the program to perform incorrectly. For example, if the program contains the statement, C=A/B there should be a previous check that B is not zero. Another aspect of completeness is the clarity and usefulness of the error messages produced by the program.

Accessability measures the degree to which the program allows the user to select desirable features such as printouts of intermediate values.

Communicativeness measures how easily a user can provide needed inputs and interpret the outputs of the program in meaningful ways.

Device independence measures the extent to which a program uses machine independent features instead of machine or compiler dependent ones. Programs which are written entirely in ANS standard languages and which do not make use of the implementations of data structures (e.g. the fact that on many CDC machines, there are six characters in an integer variable) would have high measures of device independence.

Consistency encompasses two more primitive characteristics. Internal consistency is a measure of the degree to which the program always does the same operations in the same way. External consistency measures the degree to which the program's operations and structure match those called for in the specifications.

Structuredness measures the extent to which the program has a definite, easily understood organization. Structured programming produces programs with a very high degree of structuredness. Patching and the addition of new code without consideration of the overall program organization reduce the structuredness.

Accuracy measures the degree to which the program does calculations and produces output with sufficient precision to satisfy their intended uses. A financial program which loses a penny occasionally would have a low degree of accuracy.

Existing program metrics have concentrated on measuring structuredness, and self-descriptiveness. No existing metric considers more than one or two of the characteristics the TRW study found to be important. Quality, of course, is a more general aspect of programs than complexity. Complexity is an indication of how difficult it would be to do some task on the program. Quality includes complexity as well as how difficult it would be to do some task with the program (i.e., use the program). Never-

theless, the characteristics of quality should be important for complexity as well since the tasks we want to do on the program arise from the circumstances of using the program. The success of something we do on the program is measured solely by its effects on the use of the program.

The Theory

Program complexity depends on much more than the program alone. When someone tries to understand a program, he performs three major tasks:

(1) develops an hypothesis for what the program's function is,
(2) develops an hypothesis for how the program works,
(3) matches the hypothesis of program function with the hypothesis for program operation.

The first task usually employs documentation for the system of which the program is a part or comments at the beginning of the program itself. The second task depends primarily on the program itself, but can use the first hypothesis to guide the activity. The second and third tasks often are done concurrently. A portion of the program is examined and then a tentative hypothesis for how that portion of the program operates is matched with a portion of the program's purpose. The third task might fail, forcing a return to the first or second tasks. When the third task succeeds, the program is understood. Figure 1 illustrates the procedure.

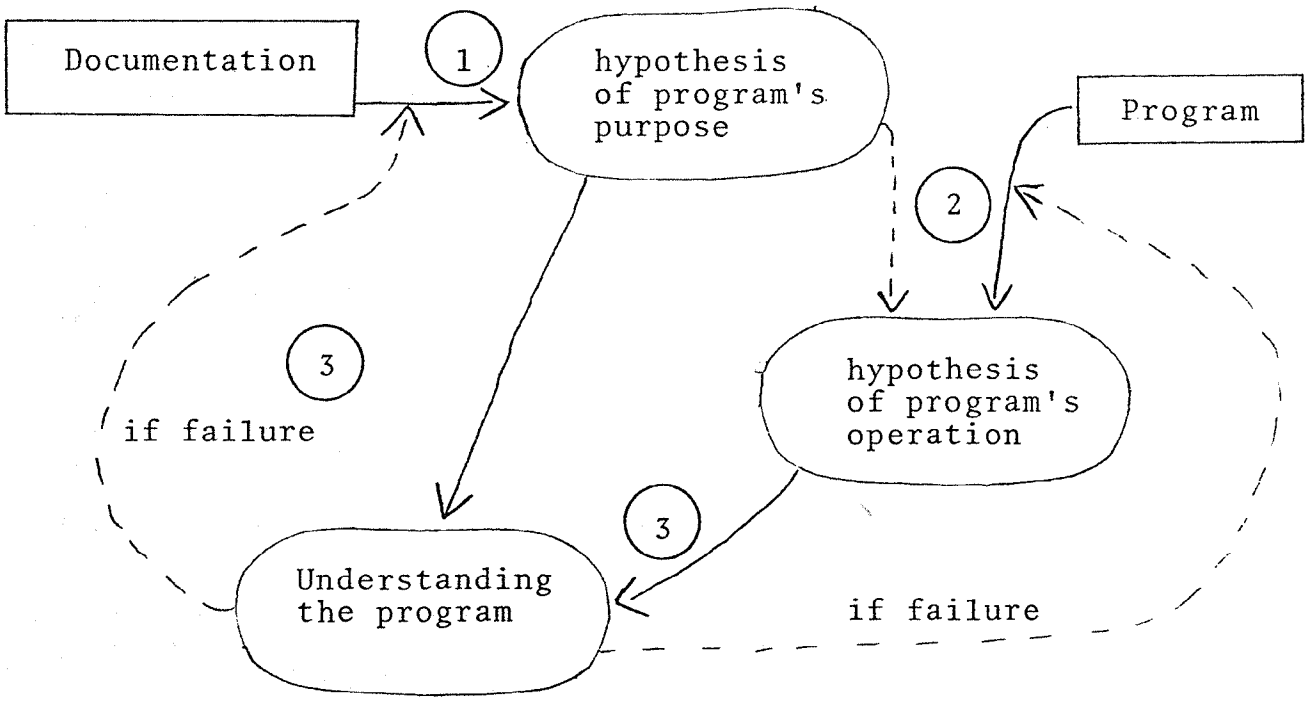

Figure 1: Understanding A Program

Tasks 1, 2, and 3 are all translations. The complexities of the three translations determines the complexity of the program. Preliminary work on the complexity of a translation is reported in [16]. A rough approximation to the complexity of a translation can be derived by dividing the source and result into logical units and drawing arrows from each source unit to the result units which implement it. The approximate complexity is the count of the total number of arrows. Second order contributers to the complexity of a translation are the number of source units with outdegree greater than four and the number of target units with indegree greater than four and the number of target units with outdegree greater than four and the number of target units with indegree greater than three. Experiments are continuing to determine the relative importance of these factors and their reliability in predicting the translation's complexity.

The theory presented here is qualitative. Substantial refinement is required before it can make quantitative predictions. Nevertheless, a great deal of evidence exists to support this theory. Some of that published evidence will be discussed in the remainder of this section. Further evidence is described in a technical report [17]. The explanations given here for the evidence, of course, are not the only possible explanations.

First, consider the eight characteristics of software quality identified by the TRW study discussed in the last section. The more self descriptive a program is, the less necessary external documentation is. This means the person trying to understand the program can use the program to develop both the function hypothesis and the operation hypothesis. The third task becomes relatively trival. The second task becomes easier as well since the function hypothesis can provide substantial direction for developing the operation hypothesis.

The more complete a program is, the more explicitly it reveals how different input are processed. The second task is easier as the program's degree of completeness increases.

Accessability helps the second task also. For a complex program someone trying to understand it might have to execute it on sample inputs in order to develop the operation hypothesis. Communicativeness also contributes to effective use of sample executions to develop the operation hypothesis. Device independence, consistency and expecially structuredness also make the development of the operation hypothesis easier and less error prone. Accuracy is theonly one of the eight characteristics which does not seem to reduce complexity according to the theory.

Moher and Schneider have recently reported a study of 100 students and sixty professional programmers [8]. They asked each programmer to perform three tasks: (1) comprehension of a fifty-one line program as measured by the time required to answer correctly questions about the program; (2) the same comprehension task for a 221 line program; (3) coding of a small program (length ranged from 38 to 94 statements). Moher and Schneider tried to correlate a large number of background characteristics of the programmers with their performances. Student programmer performance correlated well with the amount of programming experience and with student aptitude (as measured by student grade point average). Professional programmer performance correlated well with years of programming

experience, but not with any measure of aptitude. Moher and Schneider report that the inexperienced student programmers (one computer science course) did 23% worse on the short writing exercise than on the short comprehension exercise. As the students experience increased, performances on these two tasks converged. The professional programmers with zero to two years of experience had similar relative performances on the three tasks, but as experience increased to over six years, the performances on all three tasks converged.

The Moher and Schneider results are consistent with the theory of small program complexity. That theory implies that the major factors in individual differences in understanding a program are: (1) knowledge of the constructs used in the program and in its documentation; (2) ability to discern patterns of similarity between the function hypothesis and the operation hypothesis; (3) ability to abstract in meaningful ways from the program and from the documentation. The last two can be taught, but depend at least initially on innate skills and on previous experience and training in nonprogramming areas. Therefore, the theory predicts that individual differences in ability and previous nonprogramming experiences with abstraction forming and pattern recognition should determine performance by naive students. As the students gain more experience with programming, differences in ability should become less important. Performance should improve due to practice with each additional year of experience to overwhelm the aptitude factor.

The most supportive of the Moher and Schneider result for the theory is the differences in performance on comprehending and writing short programs and the observation that these differences disappear with increasing experience. Writing a program requires the author to develop the implementation hypothesis without the benefit of consulting the program. This should be much more difficult for inexperienced programmers. As the programmer becomes more experienced, he can call on previous experiences with similar programs to help develop the implementation hypothesis. Eventually, it might even be easier to use experience to develop an implementation hypothesis and write the program from it, then to develop an implementation hypothesis from an unfamiliar program. Some Moher and Schneider results for experienced professional programmers indicate that this reversal in the difficulties of comprehending and writing does occur.

Space permits mention of only one other set of published data. Sheppard, Kruesi and Curtis have reported the results of a study on the influence of specification format on comprehension of a program [19]. Seventy-two professional programmers were given nine specification-program pairs and asked questions to assess their comprehension of the programs. The specifications were presented in natural language, constrained language and flowchart ideograms. The specifications were presented in three different spatial arrangements: sequential list of statements, branching similar to a flowchart and hierarchical similar to a HIPO hierarchy chart. There were three different programs. Results showed significant performance differences due to experience with similar applications, and experience with a greater number of programming languages. These results are consistent with the theory's assumption that someone trying to understand a program uses experience to help develop the function and operation hypotheses.

Test The Theory

Of course none of the results mentioned in the previous section validates the theory. There are many other possible explanations for each of the results. Ruven Brooks has published a different theiry which is also compatible with most of those results [20]. The Brooks theory considers the complexity of a program to be based on how many different knowledge domains must contribute information to explain the operation of the program. Brooks believes that the programmer travels conceptually through a series of knowledge domains from the problem being solved to the program. The size and number of these domains determines the complexity of the program. The Brooks theory is described in extremely general terms. It is difficult to envision any evidence which would contradict it. The theory presented in this paper is less nebulaus. Specific predictions can be derived from it and tested.

For example the theory predicts that a program should be less complex when presented along with an accurate specification than when presented alone, if the person trying to understand the program knows how to use the specification. If the specification and the program do not address the same problem, the presense of the specification should make the program more complex. This prediction suggests an experiment. Professional programmers with similar experience should be divided into three groups. Each group should be given a series of program comprehension tasks where one group works with the programs alone, the second group works with the programs and correct specifications in a familar format and the third group works with the programs and incorrect specifications. The third group is not told that the specifications do not match the programs. Care must be exercised to ensure that the third group does not become suspicious of the specifications before the sequence of tasks is completed. A fourth group also could be used which was given specifications in an unfamilar format.

The theory predicts that training people in forming abstractions should improve their performances on program understanding tasks. Two groups of novice programmers could be used to test this prediction. The two groups would first be tested with a small series of programs and specifications in familar formats. Then one group could be given training and practice in constructing abstractions. This training and practice would not have to be in a programming setting. The two groups could then be tested with a different series of programs and their specifications. The trained group should improve more than the untrained group. The Moher and Schneider results indicate that the two groups should be selected to have similar aptitudes as measured by grade point averages.

The predictions mentioned above all come from the first and second tasks that the theory states are involved in understanding a program. The third task (matching the function and operation hypotheses) also can be tested. Two groups of programmers can be given specification program pairs and asked questions. The first group should be given pairs where the specification and the program have the same structure and vocabulary. The second group should be given pairs where the specification and program have very different structure and use different vocabulary when discussing

the same concept. The first group should perform better on understanding the programs than the second.

The theory assumes that someone trying to understand a program forms a function hypothesis and an operation hypothesis. This assumption could be tested by asking a group to answer questions about a program from the program and its specification. Then each member of the group could be asked to give the function and operation hypotheses without looking at the program or specification. Those who do well in answering the questions should do well in recalling function and operation hypotheses. Those who do poorly in answering the questions also should do poorly on the recall.

Many other experiments and variations on these experiments can be done. We have experiments of these types underway now and expect to report on the results in six months to a year. The importance of independent efforts to validate or reject this theory is very great.

References

(1) Belady L. A. and M. M. Lehman, "A Model of Large Program Development", IBM Systems Journal, Vol. 15, no. 3 (1976), pages 225-252.

(2) Lehman, M. M. and F. N. Parr "Program Evolution and its Impact on Software Engineering", Proceedings of the Second International Conference on Software Engineering, San Francisco, October, 1976, pages 350-357.

(3) Belady, L. A. and M. M. Lehman, "The Characteristics of Large Systems", in Research Directions in Software Technology, Peter Wegner, editor, The MIT Press, Cambridge, Mass., 1979, pages 106-138.

(4) McCabe, T. J., "A Complexity Measure", IEEE Transactions on Software Engineering, December, 1976, pages 308-320.

(5) Chen, E. T., "Program Complexity and Programmer Productivity", IEEE Transactions on Software Engineering, May, 1978, pages 187-194.

(6) Curtis, B., S. B. Sheppard, P. M. Milliman, M. A. Borst and T. Love, "Measuring the Psychological Complesity of Software Maintenance Tasks With The Halstead and McCabe Metrics", IEEE Transactions on Software Engineering, March, 1979, pages 95-104.

(7) Woodward, M. R., M. A. Hennell, and D. Hedley, "A Measure of Control-flow Complexity in Program Text", IEEE Transactions on Software Engineering, January, 1979, pages 45-50.

(8) Zolnowski, J. and D. Simmons, "Measuring Program Complexity in a COBOL Environment", Proceedings of the AFIPS National Computer Conference, Vol. 49 (1980), pages 757-766.

(9) Harrison, W. and K. Magel, "A Complexity Measure Based on Nesting Level", ACM SIGPLAN Notices, March, 1981, pages 63-74.

10) Kerninghan, B. W. and P. J. Plauger, The Elements of Programming Style, Second edition, McGraw-Hill Book Company, New York, 1978.

(11) Ledgard, H. L. Programming Proverbs, Hayden Book Company, New York, 1975.

(12) Walston, C. E. and C. P. Fleix, "A Method of Programming Measurement and Estimation", IBM Systems Journal, Vol. 16, no. 1 (1977), pages 54-73.

(13) Curtis, B., S. B. Sheppard and P. Milliman, "Third Time Charm: Stronger Prediction of Programmer Performance By Software Complexity Metrics", Proceedings of the Fourth International Conference on Software Engineering, Munich, Germany, September, 1979, pages 356-360.

(14) DeYoung, G. G. and G. R. Kampen, "Program Factors as Predictors of Program Readability", Proceedings of IEEE COMPSAC '79, Chicago, November, 1979, pages 668-673.

(15) Boehm, B. W., J. R. Brown, H. Kaspar, M. Kipow, G. J. MacLead and M. J. Merrit, Characteristics of Software Quality, North-Holland Publishing Company, New York, 1978.

(16) Magel, K. "Dependence Diagrams in Software Development", Proceedings of ACM North Central Regional Conference, September, 1981.

(17) Magel, K. "Support for A Theory of Software Complexity", Computer Science Department University of Missouri-Rolla, April 1981.

(18) Moher, T. and G. M. Schneider, "Methods for Improving Controlled Experimentation in Software Engineering", Proceedings of Fifth International Conference on Software Engineering, San Diego, March, 1981, pages 224-233.

(19) Sheppard, S. B., E. Kruesi and B. Curtis, "The Effects of Symbology and Special Arrangement on the Comprehension of Software Specifications", Proceedings of Fifth International Conference on Software Engineering, San Diego, March, 1981, pages 207-214.

(20) Brooks, R., "Using a Behavioral Theory of Program Comprehension in Software Engineering", Proceedings of Third International Conference on Software Engineering, Atlanta, May, 1978, pages 196-201.