

Layered Explanations of Software: A Methodology for Program Comprehension

Vaclav Rajlich, James Doran, Reddi.T.S.Gudla

Department of Computer Science
Wayne State University
Detroit, MI 48202, USA
vtr@cs.wayne.edu

Abstract

In dealing with the legacy systems, one often encounters poorly documented and heavily maintained software. Lack of understandability of these systems complicates the task of software maintenance, making it time consuming and limiting the possibilities of the evolution of the system. We present a methodology that helps the programmers to understand programs. Our approach is compatible with the "top-down theory" of software understanding, where the programmer creates a chain of hypotheses and subsidiary hypotheses, concerning the properties of the code. Then he/she looks for evidence (beacons) in the code. Our approach shortens the process of hypotheses creation and verification, and allows recording of successful hypotheses for the future maintenance. All information needed for understanding is recorded in layers of annotations. An experiment was conducted to investigate how the proposed methodology

helps in program understanding. A tool supporting the methodology, is presented.

1. Introduction

The first step in software maintenance, which is often the hardest, is simply understanding the code as it exists today. The software engineering field has a number of methodologies and tools that help programmers in this task, but as code complexity increases, so must a methodology's capabilities. This paper describes a methodology and an associated tool that facilitates the comprehension of programs. It is based on the cognitive theories known from the literature. The paper also describes an experiment conducted to investigate how the proposed methodology facilitates the ability of programmers' to understand programs.

SCORES	Cumulative score	mean	variance
<hr/>			
<i>Commented:</i>			
3 2 2 3 4 4 3 2 3 4 4 3 2	39	3.0	0.2248
<i>Non-Layered:</i>			
3 3 3 4 4 3 2 4 3 3 4 3 3	42	3.23	0.3313
<i>Layered:</i>			
3 4 4 4 5 4 4 4 3 4 4 4 4	51	3.92	0.6153
<hr/>			

Table 1

Before presenting our approach to program comprehension, we need to introduce some basic terminology and facts. First, program comprehension is the act of understanding a program on all levels; that is, understanding what it does conceptually as well as the fine details of the source code [1]. Some estimate the time spent on understanding a program alone to be 50 to 90 % of the total time to maintain it. Although significant progress has been made in this field, there is still a need for continued investigation.

Our work is related to the work presented in [4]. In [4], Brooks maintains that anyone who tries to comprehend a program makes certain assumptions or hypotheses based on both acquired and existing knowledge. The hypotheses are checked against the source code to prove their validity. The methodology presented in this paper, Layered explanation of software and the associated tool, the Tool for Layered Explanation of Software (or "TLES" for short), exploits the Brooks' theory of program comprehension. TLES identifies constructs of code that need to be explained; for example, a class in a C++ program is such construct. It organizes explanations of these constructs into layers. Each layer provides a summation of the decisions that the original analyst used when designing and developing the system.

In Section 2, we present a discussion of the top down, bottom up, and flat comprehension theories. Section 3 presents ideas and decisions behind TLES. Section 4 presents the experiment and the results. Section 5 describes the TLES tool. The appendices present details of the experiment.

2. Program comprehension theories

In [4], Brooks postulates that the programmer constructs a series of hypotheses to arrive at an understanding [4]. He argues that the original developer builds a program by initially gathering facts from a problem domain. After a firm understanding of the problem domain, the developer starts to apply or refine that domain towards a computer program by making a series of decisions which add multiple layers (e.g., algorithmic, representation etc.). In short, the software engineer takes a top-down approach starting with the most abstract concepts, refines them again and again, and transforms them into a computer program. The maintenance programmer performs a similar task. He reconstructs the domains and mappings as he tries to comprehend a program.

Brooks based his theory on a related research in the cognitive science. He puts forth the idea that people learn by creating hypotheses which shrink the domain

of the problem space (i.e., narrow down the problem). As an example, "sort numeric list" suggests a great deal about the structure of a program. From this requirement, we know that we must create a data structure that simulates a list (e.g. linked list), we know the type of list (numeric), and we make assumptions on the algorithm that could be used for the sort. We know if the list is not very long, a simple bubble sort would do. Longer lists suggest a binary sort or something more complex. Also, there should be an input and output of the list. All of these subsidiary hypotheses are based on the original requirements.

In the process of reconstructing the program, the maintenance programmer creates hypotheses based on what he or she has learned from the documentation and source code. These hypotheses lead to more and more refined hypotheses which create a tree-like structure where each node is an assumption. Naturally, the programmer will make incorrect hypotheses from time to time. The hypothesis tree is then traversed until the correct hypothesis branch is discovered. Hypotheses are verified through the use of segments of code known as beacons. Beacons are programming statements that help maintenance programmer prove hypothesis true or false. For example, if a programmer is tracing through a program which deals with a large database, he might assume the access to the database is indexed. Upon examining procedure names, the title "Read_Key_Field" is located. This procedure name serves as a beacon to the programmer by indicating to him that the access to the database is indexed as he thought. Beacons have been studied experimentally and proven beneficial when attempting to understand programs [10].

The concept of bottom-up comprehension is discussed in [9]. The idea presented is programmers learn by focusing on small pieces of code (perhaps as small as one line of source code) and later combine this information together. The result is the comprehension of a larger and larger part of code. For example, a maintenance programmer traces through a procedure named Binary_Sort and a procedure named Match_Item_In_List. He knows that another procedure calls Binary_Sort with a list and the output is passed to Match_Item_In_List. From this knowledge, he reasons that the procedure which calls the two subfunctions must perform a search on a sorted list.

The third type of comprehension (flat comprehension) does not attempt to organize itself in any direction (from top to bottom or vice-versa). Current implementations of hypertext, like Neptune [9], generally follow a flat comprehension structure. Neptune is a CASE tool that runs a Hypertext Abstract Machine (HAM). HAM provides analysts with

definition, modification, or retrieval of basic hypertext units, i.e. nodes and links. A node stores elements of information and links connect it to other nodes. For example, HAM could link a paragraph in a requirements specification to the module of source code where the specific requirements are implemented. The nodes that link all of the information together are not organized in a top-down or a bottom up fashion. Similarly, Horowitz and Williamsons' hypertext-like application SODOS also follows a flat structure of organization [8]. SODOS stores documents from each phase of the software life cycle in a database and links them together.

3. Layering principle

TLES provides an evolutionary history of the constructs of the program. Every programming construct, whether a variable, class, statement, an object, etc. has its origins in decisions of the programmer. A variable, for example, may represent an entity of the problem domain. As a variable progresses through successive refinements and design decisions to the source code, it undergoes a transformation: its scope is limited, its type is defined, it is expanded into a structure, etc. TLES allows the maintenance programmer to view this series of transformations (known as knowledge domains) by linking a number of layers of explanation to each construct in the program. Each of these layers relates to one of the domains used to define it.

In comprehending a program's source code, Brooks [4] points out that the maintenance programmer constructs a hypothesis tree with the series of decisions representing branches on the tree. TLES develops this idea further. It looks at the tree and horizontally groups the nodes into layers.

Even though most of the constructs undergo a transformation in every layer, some of the constructs will remain unchanged. TLES accommodates the fact that some variables and expressions in programs are created in layers closer to actual implementation or that certain constructs remain unchanged from one layer to another. As an example, say that a programmer created a variable for an application called "timeout_counter", that starts counting when the system is inactive. After it reaches a certain number, the system closes down intuiting that the execution is deadlocked. This particular variable would not be represented at a highly conceptual layer because it serves mostly as a security constraint on the system. If the application was a banking application, timeout_counter would not represent an actual entity of the financial world. Its concept is closer to the implementation layer; therefore, this

variable would not exist in the abstract (i.e., "higher") layers of TLES.

The layering principle allows maintenance programmers quick access to necessary information. Some programmers prefer to get a high-level view of the system before searching through the details of the code. Others want the details first hoping to join information together in comprehending the system. TLES can accommodate both lines of thought. TLES also allows a person to examine a program construct within the code and bring up its full history -- its reason for existence, its transformation of scope or definition, and finally its implementation.

In order to verify the concept of TLES, we conducted an experiment. The experiment and its results are described in the next section. The results supply an evidence that layered explanation of software is a helpful concept in understanding programs.

4. The experiment

The experiment was conducted in the winter of 1994 at Wayne State University, Detroit, Michigan. The subjects were 39 first-year graduate students, who were proficient in C++ programming and Software Engineering principles. The experiment was administered to all subjects in one 35-minute session. The subjects were divided into 3 groups (A, B, C) of equal ability. The basis for the division is the performance of the students in a course on Software Engineering, with all Grade A, Grade B and Grade C students equally distributed into the 3 groups. Each subject was given a C++ program listing and documentation (different for each group) and a question sheet (identical for each group) containing several questions. All the questions were multiple choice questions, with the subjects selecting the correct answer from 5 different choices (a,b,c,d,e). Only one correct choice exists for each question, and there was no ambiguity in the questions.

The program chosen for the experiment was an implementation of a tool for the OODG (Object Oriented Decomposition and Generalization) methodology for object oriented software development [11]. The program consists of four classes, with each class having 1-4 data members and 3-5 member functions (for a total of 17 member functions). The size of the program is 520 lines of code. Both the domain and the programming language (C++ language) were familiar to the subjects. Each of the three groups worked with a different documentation approach, namely Layered annotations, Non-layered annotations, and Embedded comments. They are described in the following way:

Layered annotations:

A complete description of the various components of the source code was presented in three different layers, namely : domain, algorithm, and representation. The components of the source code that are annotated include: classes, member functions of the classes, and selected local variables of the functions.

Problem domain layer contains description which is understood by an end-user and problem domain expert. Programming skills and knowledge of the implementation programming language are not required to understand the problem domain layer. Algorithm layer describes the underlying algorithms used. Representation layer depicts the constrains and assumptions made by the developer during the coding. All language dependent and machine dependent issues are presented in this layer.

A document was provided to the subjects, containing the code and the above mentioned three layers of explanations. An index listing all the software components and their explanations was provided. An example of the layered annotations of the experiment program is shown in Appendix B.

Non-layered annotations:

Same information is provided in this scheme, but in a single annotation without any distinction between the three different layers.

Commented version:

The description of each component of the source code is provided as a comment just preceding the program component in the source code. All information as provided by the layered and non-layered versions is embedded in the source code as comments.

Results and interpretation:

While conducting the experiment, we tried to remove all other influences on program comprehension. Therefore all variable names were changed to meaningless names. For each question, the subject is expected to identify the relevant portion of the source code and documentation.

The scores of the three groups, each group comprising of 13 subjects is shown in Table 1. It lists the number of correct answers for each subject. Also shown are the cumulative scores, means and variances of each group.

Notice that the group receiving the layered version scored consistently higher than the other groups. Surprisingly, a significant margin does not exist

between the scores for the non-layered and commented version.

In order to quantify the significance of these results, an analysis of statistical significance was conducted following the methodology of [6]. That is, we want to know whether the difference in understanding between the three groups of teams can be expected to be observed if we had access not only to our sample of three groups, but to all similar teams that we may encounter now and in the future.

Our null hypothesis is that in the population, the mean score for program understanding among the three groups (layered or nonlayered or commented) does not differ. To test this null hypothesis, we calculate the 't' statistic. A 't' statistic greater than 2.00 or less than -2.00 is a sufficiently compelling evidence to reject the null hypothesis [6]. The 't' statistic for the three different combinations of groups are:

t layered vs nonlayered	3.34
t nonlayered vs commented	0.8524
t layered vs commented	3.6187

Since from the experiment data, the t layered vs nonlayered and t layered vs commented are greater than 2.00, we can regard these data as sufficiently compelling evidence to cause us to reject the null hypothesis. In other words, we can conclude that there is a reliable difference between comprehension of programs with layered annotations as opposed to other styles of documentation (nonlayered or commented). On the other hand, the t nonlayered vs commented is only 0.8524, and hence, we cannot reject the null hypothesis. In other words, we cannot conclude, based on the experiment data, that there is a reliable difference between the comprehension of programs with non-layered annotation as opposed to programs with comments.

Based on the positive result of the experiment, a prototype tool for Layered Explanation of Software (called "TLES") has been developed at Wayne State University. A description of the tool follows.

5. TLES tool

TLES is an interactive implementation notebook. It allows a programmer to store and retrieve annotations describing program constructs. The layering feature of the tool supports records of an evolutionary history of the constructs of the source code. When a maintenance programmer selects a program construct (say a class in C++) a window pops up displaying the stored annotation, if any, for the class at the layer chosen.

An important feature of TLES is the ability to customize the layers. The definition and number of layers changes from system to system. For example, a program that backs up files for a customer is conceptually easy and would not require many layers of explanation. On the other hand, tax accounting software could benefit greatly from many layers because of all of the rules involved and the fact that the program is so apt to change from year to year. For this reason, TLES allows customization. The names of the layers as well as their number are specified by the user. The programmer using TLES is responsible for defining the intermediate domains the implementation passes through in order to become a program. This frees him from having to use any particular software process in order to use TLES.

Another feature of TLES is the ability to update/modify the documentation as a program is modified and constructs continue to develop. A full-length, detailed description of when and how a change was made may be added to the existing TLES documentation for future enhancements. These modifications could affect any layers and any program construct. For example, changing a variable from a string to an integer would effect only the layers closest to implementation. However, changing an account variable to an account structure to include savings accounts as well as IRAs would result in a change to the uppermost layers causing a ripple effect through all of the layers underneath it. If these changes are made consistently, the programmer reaps the full benefits of the tool.

TLES prototype was developed using AT&T C++ Language System Release 2.0 on SPARC workstations. The graphic user interface was developed using Xview toolkit release 3.1. The existing tool is used to document the C++ program constructs "class" in a C++ source code, at any number of layers.

When the tool is used, a menu appears allowing the user to select from two options: to view the documentation for an existing project, or to document a new project. When a new project is to be documented, the number of layers and their names at which the project is to be documented are to be specified first. For existing projects, class and layers have to be selected from menus. The particular annotation can be either read or updated, based on the circumstances.

6. Conclusions

In this paper we have presented a methodology that facilitates the understanding of programs. In this methodology, an evolutionary history of the basic constructs of the program is documented in a layered fashion, where decisions done on each layer are properly

recorded. This allows a user to examine a program construct within the code and bring up its full history.

The experiment was conducted on a group of graduate students, reading the same program with three different styles of documentation: layered annotations, non-layered annotations, and comments in the code. The result of the experiment was that the group receiving the layered version scored substantially higher than the other groups.

A tool to support the layered annotation scheme is also presented. The tool supports layered explanation of C++ programs at different layers. The number and the definition of the layers is configurable.

In future, a more powerful tool supporting layered explanation of source code is to be developed. It will be interfaced with a complete browser for the source code, and it will allow full browsing based on the dependencies in both the code and among the annotations.

Acknowledgements.

The authors want to thank to Jagadish K. Kadekar who implemented one of the the first versions of TLES. We also would like to thank Vasik Rajlich for statistical analysis of the results.

References.

- [1] Robson, D.J., Bennett, K.H., Cornelius, B.J., and Munro, M. "Approaches to Program Comprehension." *Journal of Systems Software* 14 (1991), 79-84.
- [2] Bennett, K.H. "The Software Maintenance of Large Software Systems." *Reliability Engineering and System Safety* 32 (1991), 135-154.
- [3] Corbi, T.A. "Program Understanding: Challenge for the 1990s." *IBM System Journal* Vol.28 No.2 (1989) 294-306.
- [4] Brooks, R. "Towards a Theory of the Comprehension of Computer Programs." *International Journal of Man-Machine Studies* 18 (1983) 543-554.
- [5] Brooks, R. "Towards a Theory of the Cognitive Processes in Computer Programming". *International Journal of Man-Machine Studies* 9 (1977) 737-751.
- [6] Judd, C.M., Smith, E.R., Kidder, L.H., "Research Methods in Social Relations", Sixth Edition, Holt, Rinehart and Winston, Inc., Fort Worth 1991, 396-405.

[7] Bigelow, J. "Hypertext and CASE." IEEE Software, Mar.1988: 23-27

[8] Horowitz, E. and Williamson, R.C., "SODOS: A Software Documentation Support Environment-- It's use." IEEE Transactions on Software Engineering 12 (11) (1986) 1076-1087.

[9] Basili, V.R. and Mills, H.D., "Understanding and Documenting Programs." IEEE Transactions on Software Engineering 8 (1982) 270-282.

[10] Wiedenbeck, S. "The Initial Stage of Program Comprehension." International Journal of Man-Machine Studies 35 (1991), 517-540.

[11] Rajlich, V., "Decomposition/Generalization Methodology for Object-Oriented Programming." Journal of Systems Software 1994; 24:181-186

Appendix A: Selected part of the source code:

```
void claa :: uildclas(classes& cmns, function& frmsn) {
    int i, j, k;
    int comum = 0;
    int claon, efolanc;
    FILE *fp;

    fp = fopen("memdoc","w");
    claon = cmns.couentc();
    efolanc = frmsn.etcno();
    for(i = 0;i<claon;i++) {
        fprintf(fp,"class %s {\n",cmns.etalacs(i));
        fprintf(fp,"\tpublic:\n");
        for(j = 0;j<efolanc;j++) {
            for(k = 0;k<3;k++)
                if(tramci[j][i].mmaetrin[k] == 'M')
                    fprintf(fp, "\t%s\n", frmsn.tfunge(j));
        }
        fprintf(fp,"\n\tprotected:\n");
        fprintf(fp,"\t...\n");
        fprintf(fp,"};\n\n");
    }
    fprintf(fp,"\n\nERRORS IN MEMBERSHIP\n");
    fprintf(fp,"_____");
    for(i = 0;i<efolanc;i++) {
        for(j = 0;j<claon;j++) {
            for(k = 0;k<3;k++)
                if(tramci[i][j].mmaetrin[k] == 'M')
                    comum++;
        }
        if(comum > 1) {
            fprintf(fp, "\n\n\t%s belongs to classes:",
```

```
frmsn.get_func(i));
for(j = 0;j<claon;j++) {
    for(k = 0;k<3;k++)
        if(tramci[i][j].mmaetrin[k] == 'M')
            fprintf(fp,"%s ",cmns.etalacs[j]);
    }
    fprintf(fp,"\n");
}
comum = 0;
}
fclose(fp);
}
```

Appendix B: Layered annotations.

function: claa::uildclas

Domain:

This function uses the relations between the deferred functions and classes, and builds the skeleton declarations of each of the classes. It also checks for errors in membership. If a deferred function has "Class Membership" relationship with more than one deferred class, an error is reported.

Algorithm:

```
Open the output file.
for (each deferred class in the list)
    write class name to output file.
for (each deferred function)
    if (function has a 'class membership' relation
        with the class)
        write the function as a public member function
        of the class.
for (each deferred function)
    if (function has a 'class membership' relation with
        more than one class)
        write error message.
```

Representation:

"memdoc" is the output file to which the skeleton declarations of the classes are written.

- cmns : Contains the list of all class names given as input by the user. The maximum number of classes is 5.
- frmsn : Contains the list of all function names given as input by the user. The maximum number of functions is 10.
- claon : Contains the number of class names in the list 'cmns'.
- efolanc : Contains the number of function names in the list 'frmsn'
- comum : counts the number of 'class membership' relations for a deferred function.

Appendix C: Commented version

```

void claa :: uildclas(classes& cmns, function& frmsn) {

// This function uses the relations between the deferred
// functions and classes, and builds the skeleton
// declarations of each of the classes. It also checks for
// errors in membership. If a deferred function has "Class
// Membership" relationship with more than one deferred
// class, an error is reported.
    int i, j, k;
    int comum = 0;
// comum : counts the number of 'class membership'
// relations for a deferred function.
    int claon, efolanc;
// claon : Contains the number of class names in the list
// 'cmns'.
// efolanc : Contains the number of function names in the
// list 'frmsn'
// cmns : Contains the list of all class names given as
// input by the user. The maximum number is 5.
// frmsn : Contains the list of all function names given
// as input by the user. The maximum number of
// functions is 10.
    FILE *fp;
// open the output file. "memdoc" is the output file to
// which the skeleton declarations of the classes are
// written.
    fp = fopen("memdoc","w");
//    number of classes
    claon = cmns.couentc();
//    number of deferred functions
    efolanc = frmsn.etcno();
    for(i = 0;i<claon;i++) {
        fprintf(fp,"class %s {\n",cmns.etlacs(i));
        fprintf(fp,"\tpublic:\n");
        for(j = 0;j<efolanc;j++) {
// Check for 'Class Membership' relationship
            for(k = 0;k<3;k++)
                if(tramci[j][i].mmaetrin[k] == 'M')
                    fprintf(fp, "\t%s\n", frmsn.tfunge(j));
        }
        fprintf(fp,"\n\tprotected:\n");
        fprintf(fp,"\t\t...\n");
        fprintf(fp,"};\n\n");
    }
    fprintf(fp,"\n\nERRORS IN MEMBERSHIP\n");
    fprintf(fp,"_____ \n");
    for(i = 0;i<efolanc;i++) {
// Check for more than one 'Class Membership'
// relationship
        for(j = 0;j<claon;j++) {
            for(k = 0;k<3;k++)
                if(tramci[i][j].mmaetrin[k] == 'M')
                    comum++;
        }
        if(comum > 1) {

```

```

// More than one 'Class Membership' ; Print error
// message
        fprintf(fp, "\n\n\t%s belongs to classes:",
            frmsn.get_func(i));
        for(j = 0;j<claon;j++) {
            for(k = 0;k<3;k++)
                if(tramci[i][j].mmaetrin[k] == 'M')
                    fprintf(fp,"%s ",cmns.etlacs[j]);
        }
        fprintf(fp,"\n");
    }
    comum = 0;
}
fclose(fp);
}

```

Appendix D: Experiment questions

- 1) A class/function pair has
 - a) Only one coupling relationship.
 - b) At least two coupling relationships.
 - c) A maximum of two coupling relationships.
 - d) Any number (0 thru 4) coupling relationships.
 - e) At least one coupling relationship
- 2) The length of a function name is
 - a) any number of characters.
 - b) must be less than or equal to 10 characters.
 - c) must be less than or equal to 5 characters.
 - d) must be less than or equal to 30 characters.
 - e) must be less than or equal to 35 characters.
- 3) The maximum number of deferred functions, that can be member functions of a class
 - a) 1
 - b) 2
 - c) 5
 - d) 8
 - e) none of the above.
- 4) What happens if a function has a 'Class Membership ('M')' relationship with two classes.
 - a) Prints an error message and quits.
 - b) Prints an error message and continues execution.
 - c) Makes the function a member of both the classes.
 - d) Makes the function a member of both the classes and then prints an error message.
 - e) none of the above.
- 5) How many times can a user input/change the coupling relationships between function/class pairs and view the results in the output file "memdoc".
 - a) 1
 - b) 5
 - c) 10
 - d) Any number of times.
 - e) none of the above.

