An experimental investigation of the impact of individual, program, and organizational characteristics on software maintenance effort

Sam Ramanujan a,1, Richard W. Scamell b,*, Jaymeen R. Shah c,2

a Computer Information Systems, College of Business Administration, Central Missouri State University, Warrensburg, MI 64015, USA
b Management Information Systems, Department of Decision and Information Sciences, College of Business Administration, University of Houston, College of Business Administration, 280J Melcher Hall, 4800 Calhoun Boulevard, Houston, TX 77204-6282, USA
c Information Systems, College of Business Administration, Prairie View A&M University, Prairie View, TX 77446-0638, USA

Abstract

Resources allocated to software maintenance constitute a major portion of the total lifecycle cost of a system and can effect the ability of an organization to react to dynamic environments. A major component of software maintenance resources is analyst and programmer labor. This paper is an experimental evaluation of how the Human Information Processing (HIP) model can serve as a framework for examining the interaction of an individual’s information processing capability and characteristics of the maintenance task. Independent variables investigated include program size, control flow complexity, variable name mnemonicity, time pressure, level of semantic knowledge and some of their interactions on maintenance effort. Data collection was done using the Program Maintenance Performance Testing System (PROMPTS) designed especially for the experiment. The results indicate that a HIP perspective on software maintenance may contribute to a decrease in maintenance cost and increase the responsiveness of maintenance to changing organizational needs. © 2000 Elsevier Science Inc. All rights reserved.

Keywords: Controlled laboratory experimentation; Software maintenance; Software maintenance effort; Control flow complexity; Variable name mnemonicity; Semantic knowledge

1. Introduction

Software maintenance is the “modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment” (ANSI/IEEE Standard 729, 1983; Schneidewind, 1987). For years it has been an important problem and currently constitutes a significant expense for organizations. For example, it is estimated that commercial software companies spend at least half of their resources on software maintenance (Kemerer, 1995), while Fortune 1000 companies spend over 75% of their information systems budget on software maintenance (Eastwood, 1993). This has led to a growing realization within organizations that productivity improvements in software maintenance would enable the redeployment of scarce resources (experienced software engineers) to other activities such as new systems development or as Dekleva (1992) observes more time may be spent on maintenance as a result of more requests for maintenance. Though there is evidence that senior managers are beginning to attach more importance to software maintenance (Lehner, 1990; Welsh, 1991; White and Cronan, 1997) especially in light of Year 2000 problems (Levin, 1997; Kirsner, 1997; Harris, 1998), as Swanson and Beath (1990) and Glass (1996) observe, software maintenance is more often misunderstood, misrepresented, and undervalued. Kemerer (1995) concludes from the review and analysis of literature on empirical research in software maintenance that the area of software maintenance is understudied relative to its practical import.

A major component of software maintenance resources is analyst and programmer labor. Since software maintenance is largely cognitive in nature, it is reasonable to employ concepts from cognitive psychology as a vehicle for understanding the psychological complexity of programs and program changes (Curtis, 1980). In the context of software maintenance, psychological
complexity refers to characteristics of software that make it difficult for individuals to understand and modify. For example, a program with many control paths is psychologically more complex than a program with fewer control paths. However, a program with many control paths becomes psychologically more simple as more regularity exists in its branching process (e.g., following a hierarchical structure in branching).

Psychological complexity and its implications for software maintenance can be studied in the context of Greeno’s (1973) adaptation of Atkinson and Shiffrin’s (1968) Human Information Processing (HIP) model (see Fig. 1). Employment of Greeno’s model is appropriate because, though relatively simple, it provides rich explanations for ‘why’ and ‘how’ certain factors (e.g., sensory registers, long-term memory (LTM), etc.) affect maintenance. In addition, unlike other prior maintenance studies which focus on software engineers’ cognitive processes (e.g., Detienne, 1990; Pennington, 1978), its application allows the entire HIP model to be considered rather than focusing on only a single factor (e.g., LTM).

2. Background

2.1. Greeno’s adaptation of the HIP model

When a person pays attention to a stimulus (e.g., specifications for program change), the information from the stimulus is gathered by sensory registers (e.g., eyes) through a process called pattern recognition. The information is then moved into short-term memory (STM).

STM has a limited capacity. Miller (1956) suggests that about 7 chunks (seven collections of related information) can be stored in STM at any given time. The process of collecting information and organizing it into chunks is referred to as chunking. The process of chunking gives the mind power that compensates for its limited capacity to store and process information. STM is used for storage purposes and does no manipulation of data. Information in STM is very volatile: it can be overlaid by switching attention and will fade away from consciousness if not rehearsed.

An individual’s (e.g., programmer’s) permanent knowledge is stored in LTM, which has an unlimited capacity for storing organized information. Information is moved from STM to LTM through a rehearsal process. LTM is comprised of two types: episodic and non-episodic (Tulving, 1972). Episodic memory consists of knowledge of a person’s past and is dependent on the time and place it is acquired. Non-episodic memory consists of knowledge that is independent of the time and place of its acquisition. This paper focuses on the non-episodic part of LTM, since the event-based episodic memory is not essential in explaining the maintenance task.

LTM has two underlying types of knowledge, semantic and syntactic (also referred to as conceptual and lexical) knowledge (Chomsky, 1965; Paivio, 1971; Atkinson et al., 1974; Collins and Loftus, 1975; Shneiderman, 1977). Semantic knowledge consists of general concepts (e.g., the notion of bubble sort rather than the individual steps of a bubble sort). Semantic knowledge is abstracted through experience and instruction, and is stored as general meaningful sets of information more or less independent of any particular context (e.g., independent of any programming language or operating system). An individual’s semantic knowledge can be seen as a network of related concepts gained mainly through ‘structured understanding’ (Werthheimer, 1959) rather than through ‘rote memorization’. Semantic knowledge is also generalizable over many syntactic representations.

Syntactic knowledge is the second kind of knowledge in LTM. It can be acquired through rote memorization and is more precise, detailed and arbitrary than semantic knowledge. In programming, syntactic knowledge involves details such as the names of library functions, valid character sets, etc. It is easier and less time consuming for human beings to learn a new syntactic construct than to acquire a completely new semantic structure. This explains the relative ease in learning a new programming language once the person knows a similar programming language.

Another type of memory used in the HIP model is the buffer or working memory (Feigenbaum, 1970). Buffer memory is more permanent than STM but less permanent than LTM. In the buffer information from STM and LTM are integrated to build new structures. During problem solving, new information from STM and the existing relevant information from LTM are integrated in the buffer and the result is used to generate the semantic structure for the solution. This semantic structure in the buffer is called internal semantics. Otherwise, the internal semantics and the relevant syntactics are moved from the buffer to the response generator in the
brain. The response generator converts these structures into appropriate activities that are needed to generate output, such as activation of vocal chords, movement of hands, etc.

2.2. An HIP perspective on software maintenance

The program maintenance task can be sub-divided into three sequential sub-tasks: (1) program comprehension – understanding the given program or its relevant parts and the change, (2) program modification – making changes to the logic of an existing program or (relevant parts) and (3) program composition – writing program code for a given specification. The following sections explain, from a HIP perspective, the process by which these sub-tasks are performed. (Hereafter when reference is made to a program we also mean the program’s relevant parts.)

Consider a hypothetical situation in which a payroll system must be modified due to a change in federal income tax rules. Also assume that this change request is given to the programmer and is expressed in unambiguous terms. One of the first activities for an analyst is to locate all the programs in the system affected by this change. Assume in this case that the tax-calculation program is the only program that needs to be modified. Once the program has been identified, the programmer must understand (i.e., comprehend) the existing program and the changes before any modifications can be made. The process of program comprehension is explained next.

Program comprehension. Program comprehension is probably the most important element of the software maintenance task. Programmers spend a significant amount of time in an effort to comprehend the source code of the program being maintained (Canfora and Lakhotia, 1999). In program comprehension, the individual maintaining the program creates a multi-level semantic structure in the buffer that represents the problem addressed by the program. This is accomplished using existing syntactic (e.g., programming language syntax) and semantic (e.g., programming structures) knowledge in LTM.

During program comprehension, the programmer first reads the program in groups of statements (chunks), instead of character by character, and moves it into the buffer through STM. The programmer then recognizes the functions of these groups of statements using pre-existing semantic and syntactic knowledge and further groups them into larger chunks until the semantic structure of the program is formed (i.e., the program is understood). The process by which one converts programs to internal semantics in the buffer is analogous to the chunking process used in STM. Outputs of this process are the semantic structures that represent the existing tax calculation program and the new tax rules. These semantic structures are stored in the buffer. Once program comprehension is completed, the next sub-task is modification.

Program modification. Program modification involves making requisite changes to the existing program’s internal semantic representation. This is done in the buffer by synthesizing the semantic structures of the existing program and the required change(s). The output is the semantic structure for the program specifications of the modified program. In the hypothetical case, the modification task results in the formation of the internal semantics of the program that incorporates the new tax rules. The output of this activity resembles program specifications represented in forms such as hierarchy charts, IPO charts, etc.

In the hypothetical case the modification activity was required because of the need to adapt a program based on a maintenance request. In addition, modifications may also be required as a result of errors during the maintenance process. For example, errors may occur due to incorrect mapping from internal semantics to program statements. These errors can be found by examining the output of the program and observing the deviation from the expected output. Also, errors may be introduced due to incorrect transformation from problem statement to internal semantics. These errors may be corrected by reevaluating the way in which the program was constructed.

Program composition. The process by which a program is composed in the context of Greeno’s adaptation of the HIP model is shown in Fig. 2. When a program composition task is presented to the software engineer, the program specifications arrive in the software engineer’s buffer through STM. Once in the buffer, the problem is analyzed and represented in terms of ‘given state’ and ‘desired state’ (Wickelgren, 1974). In the hypothetical case, the given state is the output of the modification task, present in the buffer, which includes just the semantic structure of the modified tax-calculation program. The desired state is the semantic and syntactic structure of the modified program in terms of program code.

To achieve the desired state, syntactic and semantic knowledge are transferred from the programmer’s LTM into the buffer for further analysis. This transfer plus the transfer of problem specifications into the buffer form the first stage in program composition.

In the second stage a general plan for writing a program is developed. This stage follows a pattern described by Wirth (1971) as stepwise refinement. Here the problem solution is conceived in terms of the internal semantic structure of the general plan for the program. Using a process referred to as funneling (Duncker, 1945), the internal semantics proceed from a general to a more specific and detailed plan resulting in the final program. In the hypothetical case, the result of
funneling is the conversion of the internal semantic structure of the modified specifications of the tax-calculation program to program code that implements the new rules for calculating income tax.

Ramanujan and Cooper (1994) examined the software maintenance empirical literature in an effort to study the interaction of an individual’s information processing capability and the characteristics of the maintenance task in light of the HIP model and found an accord between a variety of programmer, program, organizational, and request characteristics and maintenance effort. The studies they reviewed included various aspects of maintenance effort as the dependent variable and factors such as programmer experience, programmer quality, indentation, variable name mnemonicity, commenting or documentation, program modularity, program size, control flow complexity, loading, tightness of deadlines, and modification size as independent variables. Although seven lacked statistical significance, all 43 HIP predictions in these studies were supported.

3. Experimental environment and methodology

This research is an experimental evaluation of the Ramanujan and Cooper study (1994) and uses an operationalization of many of the same variables employed in the studies they reviewed. Students from C and object-oriented programming courses offered in the College of Business Administration and Department of Computer Science at a major southwestern university and professional programmers from local organizations served as subjects. The experimental tasks involved the maintenance of programs written in C. A list of these programs plus a brief description of each appears in Table 1.

Data in the experiment was collected using the Program Maintenance Performance Testing System (PROMPTS) designed especially for this research. A discussion of PROMPTS follows in Section 3.1. For each program shown to the subject, PROMPTS recorded the total of the time required to read the program, read the task and successfully maintain the program.

3.1. The Program Maintenance Performance Testing System

PROMPTS was the system designed to collect data in this study. PROMPTS began by asking the subject to enter his or her unique “User Identification Code.” Once a valid user identification code had been entered, the “Personal Details Screen” appeared and personal details such as the subject’s name and phone number were requested. The Personal Details Screen was displayed only when the subject invoked PROMPTS for the first time.

The “Introduction Screen” followed the Personal Details Screen and contained an explanation of the procedure for using PROMPTS to perform the maintenance task. This screen preceded each of the programs that required maintenance in the experiment. From the Introduction Screen the subject could choose one of three actions: (a) proceed to the task of maintaining the
next program; (b) suspend the experiment for a few minutes; or (c) suspend the experiment for an indefinite period of time.

For each program PROMPTS displayed a “Program Window” containing a program that required maintenance. After initially reviewing the entire program in the Program Window, the subject had to invoke the “Task Window” by moving the cursor to the last line and either clicking the mouse button or pressing the “Enter” key.

The Task Window contained the maintenance task for a program. At this point the output for both the program displayed in the Program Window and the program with the desired modifications (corrected or maintained) appeared in the Task Window. In conjunction with identifying the lines in the program in need of modification, the subject could scroll the text in the Task Window either by using the arrow keys or the scroll bar. After reading the task, the subject could switch back to the Program Window by clicking once in the Program Window or pressing the “Escape” key. The Task Window could be invoked from the Program Window at any moment.

The actual maintenance task began with the subject scrolling through the Program Window in an effort to identify the lines of code in need of modification. Upon identifying the lines which the subject felt needed to be modified in order to obtain the desired output, it was necessary to change those lines by choosing the corrected lines from the “Choice Window”. This window could be activated from any line in the program by either clicking the mouse button or pressing the Enter key. The Choice Window contained program lines that could be used to replace the lines in the program that required maintenance. The subject could click the mouse button or press the Enter key after moving the cursor to the appropriate choice in this window. In order to prevent someone from employing a “trial and error” method of maintenance (or debugging), PROMPTS displayed the same choices in the Choice Window when the window was invoked from different program lines and also generated a warning when too many erroneous choices had been made.

When the program had been correctly modified by replacing all erroneous program lines with correct program lines selected from the Choice Window, PROMPTS informed the subject and invoked the Introduction Screen for the next program.

For each program shown to the subject, the system recorded the time required to read the program, read the task and successfully maintain the program (i.e., choose the correct replacement lines from the Choice Window). If the subject was unable to maintain the program in a reasonable amount of time (this was determined in the pilot study), the subject was allowed to proceed to the next maintenance task. Subjects were also provided with two sample maintenance tasks for training in the operation of the PROMPTS software.

### 3.2. Dependent variable

The dependent variable used in this research is maintenance effort. There is little agreement in the software maintenance literature on the operationalization of this variable. For example, Gremillion (1984)

---

Some subjects were interviewed after they completed the experimental tasks in the pilot study. Their responses suggest that none of them used a “trial and error” method of debugging characterized by (a) invoking the Choice Window from program lines selected at random and then (b) randomly selecting from the choices in the Choice Window.
used number of repair requests while Vessey and Weber (1983) used number of repairs performed per production run. Vessey and Weber’s measure differed from Gremillion’s because they felt that the number of production runs affected the number of repairs and thus a ratio normalizing for production runs should be used. However, both measures fail to consider the magnitude of repairs; both simple and difficult repairs are assumed to have similar effects on maintenance effort.

A measure of maintenance effort that overcomes this deficiency is the total cost of maintenance (Bassili and Perricone, 1984). Cost would be expected to rise as the number and difficulty of repairs increase. A major component of this cost results from labor expenses (Oliver, 1979). Hence some studies have measured maintenance in terms of labor hours, assuming a constant quality of labor (e.g., Gremillion, 1984; Boehm-Davis et al., 1992).

Using an HIP perspective, maintenance effort can be measured in terms of labor hours, weighted by labor quality. This definition of maintenance effort is similar to that used by previous researchers (e.g., Banker et al., 1988; Banker and Slaughter, 1994). An advantage of this definition is that propositions which lead to reductions in labor apply to both the efficiency (cost) and effectiveness (ability to react quickly to required program changes) of program maintenance. In this study, however, the dependent variable is the time required to successfully maintain a program since one of the components of quality, semantic knowledge of programming structures, serves as an independent variable.

3.3. Independent variables

Unlike maintenance effort, there is more agreement in the literature with respect to the operationalization of various factors that affect maintenance effort. In general, many of the independent variables in this research have been measured dichotomously (e.g., high versus low, present versus absent) by previous researchers. However, use of an HIP perspective suggests changes in the measurement of some of these variables. The following paragraphs highlight the nature of these changes.

Since the maintenance tasks in this study did not require knowledge of a particular problem domain, semantic knowledge reflects only the level of programmer expertise. Programmers possess high or low semantic knowledge (i.e., are classified as expert or novice) on the basis of their ability to recognize various standard algorithms relevant to the applications being maintained. Some earlier studies (e.g., Banker et al., 1988; Vessey, 1989) have used number of years of experience to categorize programmers as expert or novice. From an HIP perspective, years of experience can be used as a surrogate for the amount of semantic knowledge, provided that components of relevant experience are also factored into this measure.

In this study, an instrument (see Appendix A) was used to classify programmers into experts and novices based on factors such as the size, the variations in control flow structures and the number of programs maintained. The characteristics of semantic knowledge, which to a large extent are experienced-based, were used to design this instrument. The programmer’s experience is captured by questions 1, 2, 17 and 19 of the instrument. Theory also suggests that the type of experience dictates the level of semantic knowledge. Questions that address the number and size of programs written or maintained (i.e., questions 3–13) along with the questions that deal with the type of training (i.e., questions 16 and 18) and knowledge of programming structures (question 14) capture the type of programming experience of a programmer. The 19 item questionnaire was expected to capture all facets of semantic knowledge for the C programming domain. A composite score was computed for each subject based on the factor weights and the subject’s response to the 16 questions in the instrument that remained in the factor solution.

From an HIP perspective, variable name mnemonicity can be operationalized in the same manner as in earlier studies (e.g., Shneiderman and McKay, 1976; Curtis et al., 1979a,b). If the variable name conveys the meaning and use of the variable in a program, it is said to have a high level of mnemonicity. On the other hand, a non-meaningful variable name is said to have a low level of mnemonicity. In this study, programs STACKMAN (prog12), PAYREPT (prog21), HISTGEN (prog32), and COUNTER (prog41) (see Table 1 and Fig. 3) possess a low level of variable name mnemonicity.

The existing maintenance literature suggests that program size should be measured as the number of program lines excluding comments (Gremillion, 1984). This approach is reasonable from the perspective of the HIP model as long as programs using the same programming language are being compared. The upper threshold for program/module size has been suggested to be between 50 lines (Fitos, 1982) and 70 lines (Woodfield et al., 1981). For experimental purposes, in this study the programs STACKMAN, PAYREPT, BILLING, and PAYROLL are classified as relatively large programs since each has over 70 lines of code (see Table 2).

McCabe’s (1976) cyclomatic number is a widely used measure of control flow complexity in maintenance research. Although there is a large amount of experimental evidence that demonstrates the ability of this measure to predict maintenance effort, it requires one modification from an HIP perspective. At present, McCabe’s cyclomatic number fails to account for the levels of nesting within various structures (e.g., two
loops in succession will result in metric values similar to those for two nested loops) and hence McCabe’s cyclomatic number does not completely capture the HIP notion of complexity. This weakness can be corrected by factoring Program Bandwidth (BW) into these measures. Program BW indicates the average level of nesting and is defined as:

\[ BW = \sum(i \times L(i)) \]

/ (number of nodes in the program control graph),

where \( L(i) \) denotes the number of nodes at level \( i \) (Belady, 1979). A program with a strictly sequential structure has a BW of 1 while a deeply nested program will have a much higher BW.

A measure of control flow complexity that factors in both the McCabe’s cyclomatic number and the BW would be ideal from an HIP perspective. In this research the product of both measures (McCabe’s cyclomatic number multiplied by Program BW) is used as the measure of complexity. As shown in Table 2, the programs used in this research have control flow complexities that range from 1 to 20.8.

In an effort to simulate an accelerated schedule (i.e., time pressure) for certain maintenance tasks, four programs were designated as ‘challenge programs’ (i.e., programs prog51, prog62, prog72, and prog81). These programs were displayed in a red window in PROMPTS. At the beginning of each experimental session, an announcement was made that each subject had been randomly assigned two challenge programs. To be eligible for the cash incentive (lottery tickets) accompanying these challenge programs, a subject’s performance had to be in the top 50% of their respective group (i.e., novice or expert). The cash incentive was awarded after all subjects had completed the experiment. Hwang (1994) has suggested that time pressure may be operationalized by varying task difficulty. The purpose of the combination of challenge programs and cash incentive was to force the subject to feel some time pressure and hopefully increase his or her concentration on certain maintenance tasks.

Table 2
Characteristics of the programs used in the research study

<table>
<thead>
<tr>
<th>Program name</th>
<th>Number of lines</th>
<th>McCabe’s cyclomatic number</th>
<th>Bandwidth</th>
<th>Composite measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>STACKMAN</td>
<td>97</td>
<td>13</td>
<td>1.6</td>
<td>20.8</td>
</tr>
<tr>
<td>PAYREPT</td>
<td>94</td>
<td>4</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>HISTGEN</td>
<td>16</td>
<td>3</td>
<td>1.5</td>
<td>4.5</td>
</tr>
<tr>
<td>COUNTER</td>
<td>16</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>BILLING</td>
<td>74</td>
<td>10</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>PAYROLL</td>
<td>79</td>
<td>11</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>16</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>TAX</td>
<td>16</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>PERSONAL</td>
<td>76</td>
<td>5</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

Fig. 3. Randomized block design employed in this research.

4 Each subject from the top 50 percentile (i.e., lowest time taken to maintain challenge programs) of their respective groups was included in the pool for winning lottery tickets. The distribution of tickets was as follows: 1 subject received 50 tickets, 2 subjects received 25 tickets each, and 15 subjects received 10 tickets each.
3.4. Research design and experimental procedure

The research design associated with the experiment in this study is a variation of the multi-group posttest design with multiple treatments (Huck et al., 1974). The treatments include variations in (a) program size, (b) variable name mnemonicity, (c) control flow complexity, (d) level of documentation, and (e) time pressure. The time taken to correctly maintain a program is the dependent variable and serves as the posttest measure. This kind of design provides a certain degree of protection to the research when a true experiment (random assignment of subjects to treatment groups) inadvertently degenerates into a quasi-experimental design when the randomization process is violated or contaminated by conditions in the research environment not under the control of the experimenter (Huck et al., 1974). Such a violation occurs in this experiment in order to test hypotheses that relate to semantic knowledge since randomization cannot be relied upon completely to assign subjects to different experimental groups. Since the experimental design employed involves one organismic (unmanipulated) variable (semantic knowledge with two levels), it may be more appropriate to label the design as a randomized block design (see Fig. 3) (Huck et al., 1974).

The multi-group posttest design with multiple treatments controls threats to internal validity such as history, maturation, instrumentation, statistical regression, mortality, and selection. However, there is still one possible uncontrolled extraneous variable (threat to internal validity), testing, which may confound the effect of the treatment on the dependent variable. Since the subjects in this study are exposed to nine different treatments (debugging tasks), it is possible that a response to a treatment may be affected by a prior treatment. For example, the subject may learn a new semantic structure while debugging a program and knowledge of this semantic structure may affect his or her performance when debugging subsequent programs. In order to control for this testing effect, the order of the treatments given to each subject was randomized.

Though the experimental design used in this research controls for the threats to internal validity, it does not control for all possible threats to external validity. External validity can be classified into two broad types: population validity and ecological validity. Population validity concerns the generalization of the results to other subjects; ecological validity concerns the generalization of the results to other settings or environmental conditions similar to the experimental setting or condition (Huck et al., 1974).

A major threat to population validity arises from the fact that the experimentally accessible population (i.e., mostly student subjects) differs from the target population (i.e., maintenance programmers). This study uses students from the MIS and computer science programs at a major southwestern university as subjects. In addition, professional programmers also serve as subjects. Nonetheless, this threat to population validity may be reduced to some extent since it is likely that many student subjects will become professional programmers in less than a year from the time they participate in the experiment. As entry level professional programmers, it is also likely that they will be involved in software maintenance during the early years of their professional careers (Swanson and Beath, 1990). Furthermore, nearly all student subjects would be expected to perform maintenance tasks while developing computer applications for various MIS and computer science courses. Thus it is reasonable to assume that the experimentally accessible population was at least similar to the target population.

The research design employed in this study controlled for some threats to ecological validity such as description of variables, the Hawthorne effect and the Rosenthal effect. In addition, as summarized in Table 3, other threats to ecological validity such as interaction of history and treatment effects, interaction of time of measurement and treatment effects, pretest and posttest sensitization, and novelty and disruption effects have no effect on the external validity.

In order to control for the threat to validity due to the description of the variables, an attempt was made to define each variable in unambiguous terms. For example, program size was defined as the number of lines in a program excluding comments. In addition, a detailed description of each variable was provided to enable other researchers to replicate the study.

The threat to ecological validity from the Hawthorne effect was decreased by not imposing a penalty for poor performance on the experimental task, thus reducing evaluation apprehension. All student subjects received extra credit for merely participating in the study and their performance in the experiment was not reported to their instructor. In addition, the Hawthorne effect was also controlled by making the experimental environment similar to the environment in which the subjects usually maintain programs. Finally, the threat to external validity due to the experimenter (or Rosenthal effect) was controlled by providing written instructions for the experiment. The experimenter just read these instructions to the subjects. In addition, the same researcher acted as the only experimenter during all experimental sessions.

3.5. Pilot studies

Two groups of students (an initial group of size 40 and a second group of size 30), taken from the introductory C programming language course offered in the College of Business Administration, participated in the pilot studies. The pilot studies were conducted for three
In order to train subjects to be productive users of software, many subjects had to ask for help from the experimenter in order to correctly maintain a program. Thus, additional training in using the PROMPTS software was required. During the first pilot study, it became apparent that subjects would require more training in using the PROMPTS software. For example, during the first pilot study, changes were made to the format of some of the PROMPTS screens. However, since the objective of the pilot studies did not include collecting data for testing the hypotheses, no attempt was made to solicit experienced C programmers as subjects.

Experience from the first pilot study also resulted in the streamlining of many administrative procedures associated with the experiment. For example, during the first pilot study it became apparent that subjects would require additional training in using the PROMPTS software; many subjects had to ask for help from the experimenter in order to correctly maintain a program. In order to train subjects to be productive users of PROMPTS, the number of sample sessions provided before the experimental tasks was increased from one to two. This change was successful in handling the training problem.

Although data was collected during both pilot studies, it was not used for testing hypotheses. Data in the first pilot study may have contained noise since some subjects were not sufficiently trained in using the PROMPTS software. In addition, since the objective of the pilot studies did not include collecting data for testing the hypotheses, no attempt was made to solicit experienced C programmers as subjects.

4. Hypotheses development

Graphical representations that show the independent variables and hypothesized relationships studied in each hypothesis appear in Fig. 4. This section contains a formal statement of each hypothesis preceded by a discussion of its rationale from a HIP perspective.
The volatility of STM along with the limitation of its size makes chunking an important aspect in storing and manipulating information in STM. Existence of regular control flow patterns in a program helps the process of chunking. Therefore, even if the program is large, program design techniques that enforce a regular control flow structure such as a hierarchical structure will result in the formation of fewer chunks and should reduce the effort to understand and maintain the program. Such a reduction in maintenance effort may not be observable for small programs that result in formation of less than 7 chunks of information in STM since STM can manipulate these chunks without resorting to rehearsal. This leads to the first hypothesis.

Fig. 4. Graphical representation of hypotheses.
Hypothesis 1. The difference in time taken to maintain a large program with a high level of control flow complexity and a large program with a low level of control flow complexity will be significantly greater than the difference in time taken to maintain a small program with a high level of control flow complexity and a small program with a low level of control flow complexity.

Since a higher amount of relevant semantic knowledge presumes a larger number of familiar control flow structures, the effect of control flow complexity on maintenance effort will be reduced. This interaction of the amount of relevant semantic knowledge (measured in terms of relevant experience) and control flow complexity on maintenance effort leads to Hypothesis 4.

Hypothesis 4. The difference in time taken to maintain a program with a high level of control flow complexity and a program with a low level of control flow complexity will be significantly greater for subjects with a low level of semantic knowledge when compared to subjects with a high level of semantic knowledge.

Hypothesis 4 suggests that control flow complexity has an effect on the amount of time required for programmers with low semantic knowledge to maintain a program. In an effort to study whether the effect hypothesized in Hypothesis 4 involves the time required by programmers with low semantic knowledge to maintain large programs of both high and low control flow complexity, Hypothesis 5 investigates this interaction in the context of small programs and studies the maintenance effort of programmers with low semantic knowledge when maintaining small programs of varied control flow complexity.

Hypothesis 5. For subjects with a low level of semantic knowledge, the time taken to maintain small programs with a high level of control flow complexity will not be significantly greater than the time taken to maintain small programs with a low level of control flow complexity.

Greater concentration on the maintenance task is expected to reduce maintenance effort since it helps in chunking by increasing the number of discriminations a person can perform for forming chunks. The program characteristics (documentation and logic characteristics) help determine the number of chunks formed in STM and buffer for a given program. The interaction of the effect of degree of concentration and documentation characteristics on maintenance effort leads to the sixth hypothesis.

Hypothesis 6. The difference in time taken to maintain a program under a low level of time pressure and the same program under a high level of time pressure will be

The formation of chunks in the STM is mainly aided by the semantic knowledge present in the LTM (de Groot, 1965). This semantic knowledge – especially in the form of different control flow structures – is gained mainly through relevant experience. Hence a person with relevant experience recognizes more regular patterns in a program than a person with less experience. Thus a person with greater relevant semantic knowledge will understand a program faster and will be able to maintain it in less time. This leads to Hypothesis 3.

Hypothesis 3. Subjects with a high level of semantic knowledge will maintain a program in significantly less time when compared to subjects with a low level of semantic knowledge.

Hypothesis 2. The difference in time taken to maintain a large program with a low level of variable name mnemonicity and a large program with a high level of variable name mnemonicity will be significantly greater than the difference in time taken to maintain a small program with a low level of variable name mnemonicity and a small program with a high level of variable name mnemonicity.

The lack of significance in some studies (Curtis et al., 1979a,b) examining the effect of variable name mnemonicity on software maintenance effort may be due to the difference in the size of the programs used in the studies. The interaction of program size with variable name mnemonicity leads to the second hypothesis.

Chunks in STM and semantic structures in the buffer are created by comparing and discriminating parts of a problem and grouping them in some logical way to form meaningful bits of information. Non-meaningful variable names place an extra burden on the programmer by forcing an encoding of the variable’s meaning. This increases psychological complexity because such variable names must be converted to their meaning for storage and manipulation. For example, in the program composition task, semantic structures in the buffer focus on the meaning and use of the variable, not on the particular variable name. A variable name that conveys the variable’s function simplifies this task. Thus the level of variable name mnemonicity should inversely affect maintenance effort. The interaction of the amount of relevant semantic knowledge (measured in terms of relevant experience) and control flow complexity leads to the sixth hypothesis.

For subjects with a low level of semantic knowledge, the time taken to maintain small programs with a high level of control flow complexity will not be significantly greater than the time taken to maintain small programs with a low level of control flow complexity.

The difference in time taken to maintain a program under a low level of time pressure and the same program under a high level of time pressure will be

5 Concentration can be manipulated by varying the time pressure for a given task. Up to a certain point, increased time pressure will lead to increased concentration and hence increase the ability to form chunks (Stroud, 1966).
significantly greater for programs with bad documentation characteristics when compared with programs with good documentation characteristics.

The number of chunks formed in STM is also a function of the relevant experience of a maintenance programmer. For maintaining a given program, an experienced maintenance programmer will form fewer STM chunks than a novice programmer. If the number of chunks to be manipulated in the memory is small (e.g., around seven), then the effect of increased degree of concentration on maintenance effort may not be observable. This interaction of degree of concentration and experience leads to Hypothesis 7.

Hypothesis 7. The difference in time taken to maintain a program under a low level of time pressure and the same program under a high level of time pressure will be significantly greater for subjects with a low level of semantic knowledge when compared to subjects with a high level of semantic knowledge.

5. Data analysis and results

5.1. Sample characteristics

The sample consisted of 100 subjects. One group of 50 subjects came from an introductory C language course offered in the College of Business Administration at a major southwestern university whereas a second group of 50 subjects were drawn from local software development organizations and from a senior-level object-oriented programming course in the Computer Science program at the same university.

The subjects in the sample were categorized into two groups: those with high semantic knowledge versus those with low semantic knowledge. An instrument was used to classify subjects into the two groups (see Appendix A). This instrument classifies subjects into those with low semantic and high semantic knowledge based on responses to questions that address the programmer’s programming experience (questions 1–4, 7, 9, 11, 13), program maintenance experience (questions 5, 6, 8, 10, 12), knowledge of programming structures (question 14), formal training in programming and systems development (questions 16, 18) and level of programming experience (questions 17, 19). A composite score was computed for each subject based on the factor weights and the subject’s response to the 16 questions in the instrument that were found to be significant in the factor solution. The median score was used to split the 100 subjects into two groups. The average composite score for the low semantic knowledge group was 55.3 with scores ranging between 45.53 and 62.98. On the other hand, the high semantic knowledge group had composite scores ranging from 13.70 to 35.44 with an average of 28.28. A t-test confirmed that these two groups were significantly different with a p-value of 0.0001.

The students from the introductory C language course formed the bulk of the low semantic knowledge group. As shown in Table 4, most subjects in this group (44 of 50) had less than 6 months of C programming experience. Further, a majority of the subjects (>40) in this group had written and maintained fewer than 25 C language programs, and none of these programs were more than 300 lines long. In addition, most subjects in this group recognized around 5 out of the 15 C programming concepts/algorithms identified in the instrument (question 14) and had a low level of formal education and experience in developing systems using C.

The high semantic knowledge group, on the other hand, consisted of programming professionals from local organizations and senior students enrolled in the Department of Computer Science. As shown in Table 4, the vast majority of the members of this group had C programming experience that varied from 2 to 5 yr and on an average they had written and maintained over 50 C programs that were more than 300 lines long. In addition, they had used over 10 of the 15 C programming concepts/algorithms that were identified in the instrument and had a high level of experience and formal training in developing applications using C.

5.2. Instrument reliability and validity

As indicated by its Cronbach’s alpha of 0.98, the 19 item instrument, of which 16 were found to be significant in the factor solution, used to classify subjects into high or low semantic knowledge is highly reliable (Nunnaly, 1978 recommends a lower limit of Cronbach’s alpha of 0.80). Furthermore, the instrument does not warrant rigorous validation since its items are based on the definition of semantic knowledge. As described, this instrument measures the type of programming knowledge and the experience through which such knowledge can be gained. Using the definition as a basis for developing the instrument accounts for its content validity. In addition, the significant loading (ranging between 0.79 and 0.94) of all the items on one factor along with the scree plot indicates that these items are all measuring a single construct – semantic knowledge. This factor accounts for 72.42% of the total variation. According to Hair et al. (1992), it is not uncommon to consider a solution that accounts for 60% of the total variance (and in some instances even less) as a satisfactory solution.

5.3. Statistical analysis of assumptions

The data gathered in the study were tested using the Analysis of Variance (ANOVA) procedure and satisfied
the two fundamental assumptions required by the test: (a) a normal distribution of scores in the population and (b) the homogeneity of variance in the treatment groups. Use of the Shapiro–Wilk’s test revealed the sample data to be normally distributed while studentized residuals between −2 and +2 satisfied the requirement that the groups have equal variance.

5.4. Results of the study

The evaluation of the seven hypotheses using the ANOVA procedure resulted in the support for all but Hypothesis 5. This section contains a discussion of the results that appear in Table 5.

Hypothesis 1 investigates the difference in time taken to maintain programs that have different sizes and control flow complexity. As shown by the analysis of the data in Table 5, the difference in the reduction in maintenance effort due to reduction of control flow complexity is greater in larger programs when compared to smaller programs (449.21 s versus 276.83 s for larger programs as opposed to 281.88 versus 163.06 s for smaller programs). Likewise, an analysis of the results associated with Hypothesis 2 reveals a similar reduction in maintenance effort for larger versus smaller programs when variable name mnemonicity is increased.

Hypothesis 3 is the first hypothesis that investigates semantic knowledge. As shown by the results in Table 5, programmers with a high level of semantic knowledge take significantly less time to maintain a program when compared to programmers with a lower level of semantic knowledge. An analysis of Hypothesis 4 reveals that the decrease in maintenance effort due to the decrease in the level of control flow complexity is greater in programmers with low semantic knowledge when compared to programmers with high semantic knowledge (419.11 s versus 192.82 s as opposed to 311.99 s versus 145.07 s).

While Hypothesis 4 suggested that control flow complexity has an effect on the time taken to maintain a program for programmers with low semantic knowledge, Hypothesis 5 asserted that such an effect would not be observable when maintaining smaller programs (i.e., meaning that support for this hypothesis could be obtained by failing to reject the hypothesis). However, as shown in Table 5, there is a significant difference in the time required to maintain a small program with a high level of control flow complexity when compared to a small program with a lower level of control flow complexity.

Rejection of Hypothesis 5 suggests that maintenance effort is more sensitive to changes in control flow complexity.

Table 4
Characteristics of the sample

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Responses</th>
<th>Number of the subjects in the</th>
<th>Low semantic knowledge group</th>
<th>High semantic knowledge group</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Programming experience</td>
<td>&gt;60 months</td>
<td>0</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td></td>
<td>24–60 months</td>
<td>3</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td></td>
<td>12–24 months</td>
<td>11</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6–12 months</td>
<td>13</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0–6 months</td>
<td>23</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2. C language programming experience</td>
<td>&gt;60 months</td>
<td>0</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td></td>
<td>24–60 months</td>
<td>2</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td></td>
<td>12–24 months</td>
<td>2</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6–12 months</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0–6 months</td>
<td>44</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3. Total number of C language programs written</td>
<td>&gt;100</td>
<td>1</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td></td>
<td>50–100</td>
<td>1</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td></td>
<td>25–50</td>
<td>7</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0–25</td>
<td>42</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4. Total number of large (&gt;300 lines) C language programs written</td>
<td>&gt;100</td>
<td>0</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>25–100</td>
<td>0</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1–25</td>
<td>7</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td></td>
<td>None</td>
<td>43</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>5. Total number of C language programs maintained</td>
<td>&gt;100</td>
<td>0</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td></td>
<td>50–100</td>
<td>0</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td></td>
<td>25–50</td>
<td>9</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0–25</td>
<td>41</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>6. Total number of complete C applications written</td>
<td>&gt;20</td>
<td>0</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10–20</td>
<td>3</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5–10</td>
<td>5</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0–5</td>
<td>42</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
### Table 5
Results for Hypotheses 1–7

<table>
<thead>
<tr>
<th>Hypotheses</th>
<th>n</th>
<th>Test statistic</th>
<th>p-Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANOVA/contrast results</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hypothesis 1: $\mu_{\text{prog11} + \text{prog12}} - \mu_{\text{prog21} + \text{prog22}} &gt; 0$</td>
<td>400</td>
<td>$t = 14.56$</td>
<td>0.0001</td>
</tr>
<tr>
<td>Hypothesis 2: $\mu_{\text{prog12} + \text{prog21}} - \mu_{\text{prog11} + \text{prog22}} &gt; 0$</td>
<td>400</td>
<td>$T = 12.84$</td>
<td>0.0001</td>
</tr>
<tr>
<td>Hypothesis 3: $\mu_{\text{prog11} + \text{prog12} + \text{prog31} + \text{prog32}} - \mu_{\text{prog21} + \text{prog22} + \text{prog41} + \text{prog42}} &gt; 0$</td>
<td>100</td>
<td>$F = 27.01$</td>
<td>0.0002</td>
</tr>
<tr>
<td>Hypothesis 4: $\mu_{\text{prog11} + \text{prog12} + \text{prog31} + \text{prog32}} - \mu_{\text{prog21} + \text{prog22} + \text{prog41} + \text{prog42}} &gt; 0$</td>
<td>200</td>
<td>$F = 2.13$</td>
<td>0.0399</td>
</tr>
<tr>
<td>Hypothesis 5: $\mu_{\text{prog51} + \text{prog52}} - \mu_{\text{prog61} + \text{prog62}} &gt; 0$</td>
<td>50</td>
<td>$F = 224.40$</td>
<td>0.0001</td>
</tr>
<tr>
<td>Hypothesis 6: $\mu_{\text{prog51} + \text{prog52}} - \mu_{\text{prog61} + \text{prog62}} &gt; 0$</td>
<td>50</td>
<td>$F = 3.26$</td>
<td>0.0012</td>
</tr>
</tbody>
</table>

**Descriptive statistics**

<table>
<thead>
<tr>
<th>Program identifiers</th>
<th>Program characteristics</th>
<th>n</th>
<th>Mean time to maintain (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>prog11 + prog12</td>
<td>Size – Large Complexity – High</td>
<td>100</td>
<td>449.21</td>
</tr>
<tr>
<td>prog21 + prog22</td>
<td>Size – Large Complexity – Low</td>
<td>100</td>
<td>276.83</td>
</tr>
<tr>
<td>prog31 + prog32</td>
<td>Size – Small Complexity – Low</td>
<td>100</td>
<td>161.31</td>
</tr>
<tr>
<td>prog41 + prog42</td>
<td>Size – Small Complexity – High</td>
<td>100</td>
<td>313.22</td>
</tr>
<tr>
<td>prog51 + prog52</td>
<td>Size – Small Mnemonicity – High</td>
<td>100</td>
<td>259.77</td>
</tr>
<tr>
<td>prog51 + prog61</td>
<td>Time Pressure – Low</td>
<td>50</td>
<td>341.44</td>
</tr>
<tr>
<td>prog52 + prog52</td>
<td>Time Pressure – High</td>
<td>50</td>
<td>309.52</td>
</tr>
<tr>
<td>prog61 + prog62</td>
<td>Time Pressure – Low</td>
<td>50</td>
<td>309.52</td>
</tr>
<tr>
<td>prog62 + prog62</td>
<td>Time Pressure – High</td>
<td>50</td>
<td>335.91</td>
</tr>
</tbody>
</table>

**Note:** Data from the ninth maintenance tasks was used to examine a variable (repair request detail) not included in this study. Hence the analysis of Hypothesis 3 is based on data from only eight maintenance tasks (i.e., programs).
complexity than expected. This hypothesis compares the time taken to maintain two 16 line programs, COUNTER and HISTGEN, that have control flow complexity measures of 1.0 and 4.5, respectively. The large magnitude of difference (4.5 versus 1.0) in control flow complexity may have led to the difference in time taken to maintain these programs. It is possible that maintaining a program with control flow complexity of 4.5 led to formation of more than 7 chunks in the STM, thereby leading to a significantly higher maintenance effort when compared with maintaining a program with control flow complexity of 1.0 which leads to formation of fewer than 7 chunks. This was not expected given the current practice of classifying all programs with control flow complexity (McCabe’s number) of 10 and above as complex programs. The results of this study suggest that even small programs with composite control flow complexity measure of 4.5 (see Table 2) can lead to formation of more than 7 chunks in the STM and thus be classified as complex programs.

Hypotheses 6 and 7 deal with operationalizations of time pressure. As suggested by Hypothesis 6, the difference in time taken to maintain programs under low time pressure versus high time pressure is greater for programs with poor documentation characteristics (309.54 s for low time pressure versus 386.31 s for high time pressure) when compared to programs with good documentation characteristics (341.44 s for low time pressure versus 336.08 s for high time pressure). Hypothesis 7 also considers time pressure by investigating the effect of programmer’s semantic knowledge along with time pressure on maintenance effort. Consistent with the hypothesis, the results shown in Table 5 indicate that increased time pressure reduces maintenance effort to a greater extent in programmers with lower semantic knowledge (398.02 s versus 335.91 s as opposed to 329.72 s versus 309.52 s).

6. Discussion

The results of the study, summarized in Table 6, reveal several interesting relationships between maintenance effort and various programmer, program, and organizational characteristics. Program characteristics (control flow complexity and variable name mnemonicity) were found to have a significant effect on maintenance effort. The results of Hypotheses 1 and 2 indicate this effect was stronger for larger programs. In particular, in Hypothesis 1 an increase in control flow complexity increased maintenance effort to a greater extent in larger programs when compared to smaller programs. This encourages the use of program design techniques that recommend strict control flow structures, especially for designing larger programs. Consistent with the findings for Hypothesis 1, the results for Hypothesis 2 indicate that an increase in variable name mnemonicity reduced maintenance effort in larger programs but not for smaller programs. This encourages the presence of standards for variable naming for larger programs.

Semantic knowledge also had an effect on maintenance effort. As expected programmers with high level semantic knowledge were found to require less maintenance effort when compared to programmers with low level semantic knowledge. This finding suggests the use of experienced programmers for maintenance, especially if the experience is gained by maintaining or developing many programs with varied control flow structures. It encourages training of programmers by focusing on different programming structures rather than on the syntax of a programming language. In addition, the results of Hypothesis 4 imply that a programmer who has gained experience by working with programs of varied control flow structures should be assigned to maintain complex programs.

Table 6
Summary of the results of this study

<table>
<thead>
<tr>
<th>Hypotheses</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. The marginal increase in maintenance effort from an increase in control flow complexity will be greater in larger programs or larger modifications.</td>
<td>Supported</td>
</tr>
<tr>
<td>2. The marginal decrease in maintenance effort because of higher variable name mnemonicity will be smaller for smaller programs.</td>
<td>Supported</td>
</tr>
<tr>
<td>3. An increase in relevant semantic knowledge will decrease maintenance effort.</td>
<td>Supported</td>
</tr>
<tr>
<td>4. The marginal increase in maintenance effort from an increase in the control flow complexity will be less for a programmer with higher relevant semantic knowledge.</td>
<td>Supported</td>
</tr>
<tr>
<td>5. Irrespective of the amount of relevant semantic knowledge, the marginal increase in maintenance effort from an increase in the control flow complexity will not be significantly different for small programs.</td>
<td>Not Supported</td>
</tr>
<tr>
<td>6. The reduction of maintenance effort from increased programmer concentration (or increased time pressure) will be less while maintaining programs with good characteristics.</td>
<td>Supported</td>
</tr>
<tr>
<td>7. The reduction of maintenance effort from increased programmer concentration will be lower for programmers with a higher level of semantic knowledge.</td>
<td>Supported</td>
</tr>
</tbody>
</table>
The results of this study contradict Hypothesis 5. It was found that even in small programs, the increased control flow complexity resulted in higher maintenance effort for programmers with low level semantic knowledge. This suggests the use of more experienced programmers for maintaining programs with complex control flow structures even if the programs are small.

Collectively, the results associated with Hypotheses 6 and 7 indicate that increased time pressure decreased maintenance effort while maintaining programs with poor documentation characteristics. This effect was observed to be stronger for programmers with low level semantic knowledge. Hypothesis 6 implies that if a program has good documentation characteristics then the effect of organizational characteristics that increase concentration may not have large impacts. Hence, the setting of deadlines in order to increase the degree of concentration will help reduce maintenance effort to a greater extent if the programs have poor documentation characteristics. Since relevant semantic knowledge can be gained through relevant experience, Hypothesis 7 implies that setting of targets and deadlines in order to increase the degree of concentration may not be very effective in reducing the maintenance effort if the programmer/analyst has a large amount of relevant experience. On the other hand, organizational characteristics such as setting deadlines, and providing incentives that serve to increase the degree of concentration may help reduce the maintenance effort for novice programmers/analysts, at least to a certain extent. This qualification is based on studies that examine the effect of an individual’s concentration on their ability to chunk information (Stroud, 1966; Schroeder et al., 1967; Halstead, 1977). These studies conclude that, depending on the level of concentration, a human brain can perform between 5 and 20 discriminations per second when grouping related information to form chunks. However, when a person works on more than one task at a time, the number of discriminations decreases to around 5 per second. Further, when distractions exist, the volatile nature of STM can cause its contents to be erased or over-written. Conversely, when time pressure forces an individual to fully concentrate on a single task, he or she is capable of up to 20 discriminations per second with minimal loss of STM information. Thus a higher degree of concentration resulting from time pressure can reduce maintenance effort as a result of (a) both the enhanced ability to form chunks and develop internal semantics, and (b) a reduction in the loss of STM information. It is important to note that increases in time pressure will not reduce maintenance effort beyond a certain point since it has been shown that an average person is not capable of more than 20 discriminations per second (Stroud, 1966).

Based on the results of their study, Peters et al. (1984) have proposed a theory which suggests an inverted U-shaped relationship between time pressure and performance (i.e., increasing time pressure can result in a better performance up to a certain point; beyond which an increase in time pressure results in performance deterioration).

6.1. Contributions to practitioners

This study offers several contributions to practitioners. Even though additional research is necessary to affirm or negate the intensity of program size, level of semantic knowledge, time pressure, and level of control flow complexity on maintenance effort, the results of this research suggest that these factors play an important role in determining the magnitude of maintenance effort.

At a macro level, the results serve as a basis for explaining why the cost of software maintenance can be high. For example, support for Hypotheses 3 and 4 suggests that the practice of using novice programmers for software maintenance (Swanson and Beath, 1990) may contribute to the high cost of maintenance. Novice programmers take significantly more time to maintain programs leading to higher cost and lower efficiency of software maintenance.

At a micro level, the results provide several ways to reduce maintenance effort. In particular, they suggest changes to certain maintenance practices. Support for Hypothesis 1 indicates that maintenance effort can be reduced by using program design techniques that utilize concise control flow structures, especially when developing larger programs. Results of the study, in particular the support for Hypothesis 2, also suggest the use of standards for variable naming when designing large programs. Absence of the use of variable naming standards or structured program design methods may result in programs with a high level of complexity. Banker et al. (1993) and Curtis et al. (1979b) suggest that maintaining such programs will increase the cognitive load on the person trying to comprehend the program when maintaining it. This could result in an increase in the effort required to maintain the program and may decrease the productivity of the person doing the maintenance. These results suggest that software managers may want to consider promoting the use of structured program design methods, and established variable naming standards for new system development in order to reduce the overall system lifecycle cost.

In addition to making suggestions for program design, the results of this study also have implications for the management of software maintenance. Support for Hypotheses 3 and 4 suggests the use of programmers who have gained experience by working with programs of varied control flow structures for maintaining complex programs. According to this study, novice programmers should not be assigned to maintain complex programs.
One of the methods that can be used to reduce maintenance effort is to increase the time pressure on the programmers by setting realistic deadlines or by providing incentives. This practice is especially effective when the programs have poor documentation characteristics or when they are maintained by novice programmers.

The results also offer suggestions for assigning programmers to projects characterized by varied time pressure, program size, and control flow complexity. For example, while the results of the Hypothesis 4 suggest the use of high semantic knowledge persons for maintaining complex programs, organizational constraints may make it impossible to assign experienced programmers to maintenance. This may be compounded by the results of Hypotheses 4, 5 and 7 which suggest that assigning a low semantic knowledge person to a maintenance project with high time pressure, high control flow complexity, and low size would delay the completion of the maintenance task. In such situations, a manager might consider the implications of Hypotheses 5 and 7 and assign smaller modules of complex programs to a low semantic knowledge person along with tight deadlines. This would allow a novice programmer to receive training in maintaining complex programs, reduce maintenance effort and thus the maintenance cost, and permit experienced programmers to be used for new development tasks.

6.2. Limitations of the study

There are several limitations to this study. First, the investigations comprising this research are essentially restricted to laboratory experimentation thereby limiting the external validity (generalization) of the results. Even though conclusions drawn from experimental research (performed in laboratory settings) are empirically stronger than those drawn from non-experimental research conducted in the field, field studies are preferred for their generality and testing of real world phenomena. Chapanis (1983) points out that laboratory experiments are, at best, rough and approximate models of real-life situations and can select only a few independent variables for testing.

Second, while psychometric properties of the instrument used to group subjects based on their semantic knowledge have been validated, a systematic evaluation of the psychometric properties of the semantic knowledge instrument through several replications is required before the validity of this instrument can be unequivocally established. The factor analytic method used in deriving and validating the semantic knowledge instrument is not as comprehensive or acceptable as other methods such as Multi-Trait Multi Method (MTMM) for validating instruments.

Third, although a concerted effort was made to secure a sample representative of the target population of 100, certain groups (especially students from the C programming class offered in the College of Business Administration) tended to be disproportionately represented in the sample due to practical considerations in gathering a critical mass of experimental subjects. Any claim of external validity must be tempered with the recognition of the fact that the sample, strictly speaking, was not randomly selected from the target population.

The next limitation of this study is the limited manipulation of the independent variables. All independent variables in this study are classified dichotomously as either high or low. Such dichotomous measurement allows only for relative analysis and thus the applicability of the results of the study to real-world situations. For example, the results of the study indicate that presence of time pressure reduces software maintenance effort for programs with poor documentation characteristics. However, no conclusions can be drawn on how much or what level of time pressure is required to induce such reduction in maintenance effort.

Finally, the fifth limitation involves the nature of the tasks used in this study. The size of the tasks was dictated by the limited availability of the subjects for the experiments. Tasks were designed so that all subjects would be able to complete them within a reasonable amount of time (i.e., 2 h). This led to using C programs whose size ranged from 16 to 97 lines of code. Though these programs were larger than those used in earlier studies on software maintenance, they may not be representative of the size of programs in industry and thus may restrict the external validity of this study.

6.3. Suggestions for future research

Future research in software maintenance might take three forms. First, the application of the HIP model to software maintenance effort can be used as a basis for generating additional research questions that focus on the reduction of maintenance costs. For example, a possible question from the HIP perspective would be “Modularization decreases maintenance effort. However, beyond a certain point, increased modularization will increase maintenance effort for experts.” Second, field experiments could be conducted in an effort to better support the external validity of the results of this research. Visaggio’s (1999) experiment evaluating the Quick Fix versus the Iterative Enhancement maintenance

---

6 In this study, time pressure is manipulated by providing incentives in the form of lottery tickets to quickly complete a maintenance task.
paradigms is a recent example of a study that included professional information systems people whose activities largely involved software maintenance. Finally, the impact of the use of debugging environments that accompany today’s modern application development environments represents another interesting area to investigate.

6.4. Concluding remarks

This study can best be described as a precursor to a program of research in the area of software maintenance effort and provides a theoretical basis for a more comprehensive program of research. Research methods that provide for greater external validity such as field experiments and case studies should be used to evaluate the results of this study. Such triangulation will provide confidence for practitioners to incorporate the results of this study into their maintenance practices.

Once again, it should be emphasized that this study is a small step in the direction toward building a cognitive psychology based theory of software maintenance effort that not only predicts “what” factors affect software maintenance effort but also explains “why” and “how” these factors influence software maintenance effort.

The HIP model focuses on the individual. However, software maintenance often is a function of team behavior and interaction with clients, especially during the phase in which the programmer tries to understand the client’s maintenance requests. Therefore we also recognize that theories of software maintenance effort can be based on other theoretical perspectives from fields such as sociology and social psychology.

Appendix A

Personal Information
Login id: NDIS71
Name:

Programming experience:

(Check the most appropriate choice)

1. How many months of programming experience do you have?
   1. more than 60 months
   2. between 24 and 60 months
   3. between 12 and 24 months
   4. between 6 and 12 months
   5. less than 6 months

2. How many months of C language programming experience do you have?
   1. more than 60 months
   2. between 24 and 60 months
   3. between 12 and 24 months
   4. between 6 and 12 months
   5. less than 6 months

3. How many C programs have you written:
   1. more than 100
   2. between 50 and 100
   3. between 25 and 50
   4. fewer than 25

4. How many C programs did you write in the last 12 months:
   1. more than 100
   2. between 50 and 100
   3. between 25 and 50
   4. fewer than 25

5. How many C programs have you debugged or maintained:
   1. more than 100
   2. between 50 and 100
   3. between 25 and 50
   4. fewer than 25

6. How many C programs have you debugged or maintained in the last 12 months:
   1. more than 100
   2. between 50 and 100
   3. between 25 and 50
   4. fewer than 25

7. How many C programs you wrote were more than 300 lines long (excluding comments)?
   1. more than 100
   2. between 50 and 100
   3. between 25 and 3
   4. fewer than 3

8. How many C programs you debugged or maintained were more than 300 lines long?
   1. more than 100
   2. between 50 and 100
   3. between 25 and 3
   4. fewer than 3

9. How many C programs you wrote were more than 200 lines long?
   1. more than 100
   2. between 50 and 100
   3. between 3 and 25
   4. fewer than 3

10. How many C programs you debugged or maintained were more than 200 lines long?
    1. more than 100
    2. between 50 and 100
    3. between 3 and 25
    4. fewer than 3
11. How many C programs you wrote were more than 100 lines long?
   1. more than 100
   2. between 25 and 100
   3. between 3 and 25
   4. fewer than 3

12. How many C programs you debugged or maintained were more than 100 lines long?
   1. more than 100
   2. between 25 and 100
   3. between 3 and 25
   4. fewer than 3

13. How many complete applications have you built using C programs?
   1. more than 20
   2. between 10 and 20
   3. between 5 and 10
   4. less than 5

14. Which of the following concepts/algorithms have you used in your C programs? (Mark all appropriate choices)
   a) Stack manipulation
   b) Recursion
   c) Branching
   d) Looping
   e) Linked list
   f) Pointers
   g) Arrays
   h) Heap sort
   i) Bi-directional list
   j) Circular list
   k) B Tree (Traversing, Deletion, Insertion)
   l) Balance B Trees (Traversing, Deletion, Insertion)
   m) Quick sort (Procedures, Distributive partitioning, etc.)
   n) Stackless traversal (Threaded trees, Link inversion, etc.)

14a. Which of the following variable names make sense to you?
   a) Net_pay
   b) Gross_pay
   c) Counter
   d) Stack_node_ptr
   e) Average
   f) Maxnum
   g) Choice
   h) Current_ptr
   i) Pagenum
   j) Linenum
   k) Cost
   l) Price

14b. Which of the following variable names do not make any sense?
   a) SPPXN
   b) QREW
   c) X
   d) isempty
   e) tidsa
   f) iuyds
   g) wxy

Educational background:

15. What is your highest level of education completed?

   For each of the following questions, two adjectives are provided that might describe your level of training and experiences with programming. Please circle the number that best describes you for each question.

16. How would you describe your level of formal training in C language?
   1 None
   2 Extensive

17. How would you describe your level of experience with developing programs using C language?
   1 None
   2 Extensive

18. How would you describe your level of formal training on methods and procedures for developing complete computer applications using C?
   1 None
   2 Extensive

19. How would you describe your level of experience in developing complete computer applications using C?
   1 None
   2 Extensive

References


Miller, G., 1956. The magical number seven, plus or minus two: some limits in our capacity for processing information. Psychology Review 63, 81–97.


Sam Ramanujan is an Assistant Professor of Computer Information Systems at Central Missouri State University. Prior to this he taught information systems courses at the University of Houston while working on his Ph.D. His current research interests include software engineering, Internet/Intranet technologies, global information systems, and distributed systems architecture. He has published papers in the Journal of Global Information Systems and OMEGA.

Richard W. Scamell is a Professor in the Department of Decision and Information Sciences at the University of Houston. His current teaching and research interests focus on database management systems and behavioral implications surrounding the adoption of information technologies. He has published papers in journals such as Management Science, Decision Sciences, MIS Quarterly, Communications of the ACM, Academy of Management Journal, IEEE Transactions on Software Engineering, ACM Transactions on Office Information Systems, and OMEGA.

Jaymeen R. Shah is an Assistant Professor of Information Systems in the Department of Accounting, Finance and Information Systems at Prairie View A&M University. His research interests are in the areas of emerging technologies, neural network applications, computer-mediated communication systems, production planning and scheduling, and engineering management. He has published articles in journals such as European Journal of Operational Research, OMEGA, Engineering Management Journal, and Journal of Information and Optimization Sciences. In addition, he has published articles in national and international conference proceedings.