العنوان:	Measuring software complexity for software engineering quality assurance
المؤلف الرئيسي:	Wohaishi, Ahmad Muhammad Ali
مؤلفين آخرين:	Fahmy, Maged A.، Al Sultanny, Yas A.(Co-Advisor، Advisor)
التاريخ الميلادي:	2009
موقع:	المنامة
الصفحات:	1 - 155
رقم MD:	736039
نوع المحتوى:	رسائل جامعية
اللغة:	English
الدرجة العلمية:	رسالة ماجستير
الجامعة:	جامعة الخليج العربي
الكلية:	كلية الدراسات العليا
الدولة:	البحرين
قواعد المعلومات:	Dissertations
مواضيع:	برامج الحاسوب، هندسة البرمجيات، تكنولوجيا المعلومات
رابط:	https://search.mandumah.com/Record/736039

Chapter 1

Introduction and Overview

As software systems importance has grown, the software community has continually attempted to develop technologies that will make it easier, and less expensive to build and maintain high-quality computer programs. Some of these technologies are targeted at a specific application domain (e.g., web-site design and implementation), other focus on a technology domain e.g. objected-oriented systems or aspect-oriented programming (Deitel H. et. al. 2009). This chapter presents an overview of the thesis, it describes the problem statement and continues with the thesis contribution. Then, it presents thesis outline.

1.1 Software Engineering

Several software systems have been developed over the past few years. However, the absence of a standard regulatory mechanism in terms of quality control/quality assurance with respect to implementation and managing projects, particularly in the industrial sector has lead to an inconsistency among the various software systems. The complexity of each project and uniqueness make the task even more difficult. With an eye on new methodologies and tools relevant to the entire life cycle, from conceptualization to implementation, the quality assurance of software has to be visualized. Development of software for managing projects is an extremely complex affair. Usually, the evolution of the software is the result of team work or rather several groups of specialists, who individually are experts in their respective disciplines, but probably may now have enough expertise in other disciplines. The work is quite tasking and time consuming. Whatever be the technique for final testing of the software, however organized be the methodology, however systematic be the documentation involved as well as control of the final configuration, it would be a fruitless exercise if proper management of quality assurance is not in place (Pressman R. 2005).

Software can also be seen as an interface between the problem domain and the computer as shown in Figure 1.1. Software Engineering, as defined in IEEE Standard 610.12, is: "The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software".

Virtually all countries now depend on complex computer-based systems (Somerville I. 2006). More and more products incorporate computers and controlling software in some form. The software in these systems represents a large and increasing proportion of the total system costs. Therefore, producing software in a cost-effective way is essential for the functioning of national and international economies software engineering is an engineering discipline whose goal is the cost effective development of software systems. In some ways, this simplifies software engineering as there are no physical limitations on the potential of software. In other ways, however, this lack of natural constraints means that software can easily become extremely complex and hence very difficult to understand (Nasib S. 2005).

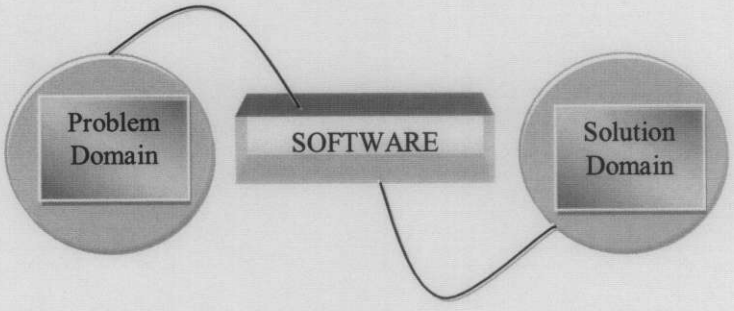


Figure 1.1: Contemporary View

Figure 1.2 shows short history of software. During the early years of the computer era, general-purpose hardware became commonplace. Product software (i.e., programs developed to be sold to one or more customers) was in its infancy. The second era of computer system spanned the decade from the mid-1960s to the late 1970s. Multiprogramming and multi-user systems introduced new concepts of human-machine interaction. Real-time systems could collect, analyze, and transform data from multiple sources. The second era was also characterized by the use of product software and the advent of "software houses." The third era of computer system evolution began in mid-1970s and spanned more than a full decade. The distributed system greatly increased the complexity of computer-based systems. The conclusion of third era was characterized by

the advent and widespread use of microprocessors. In less than a decade, computers became readily accessible to the public at large. The fourth era of computer system evolution moves us away from individual computers and computer programs and toward the collective impact of computers and software. Powerful desk-top machines controlled by sophisticated operating systems, networked locally and globally, and coupled with advanced software applications have become the norm. The Internet exploded and changed the way of life and business (Pressman R. 2005).

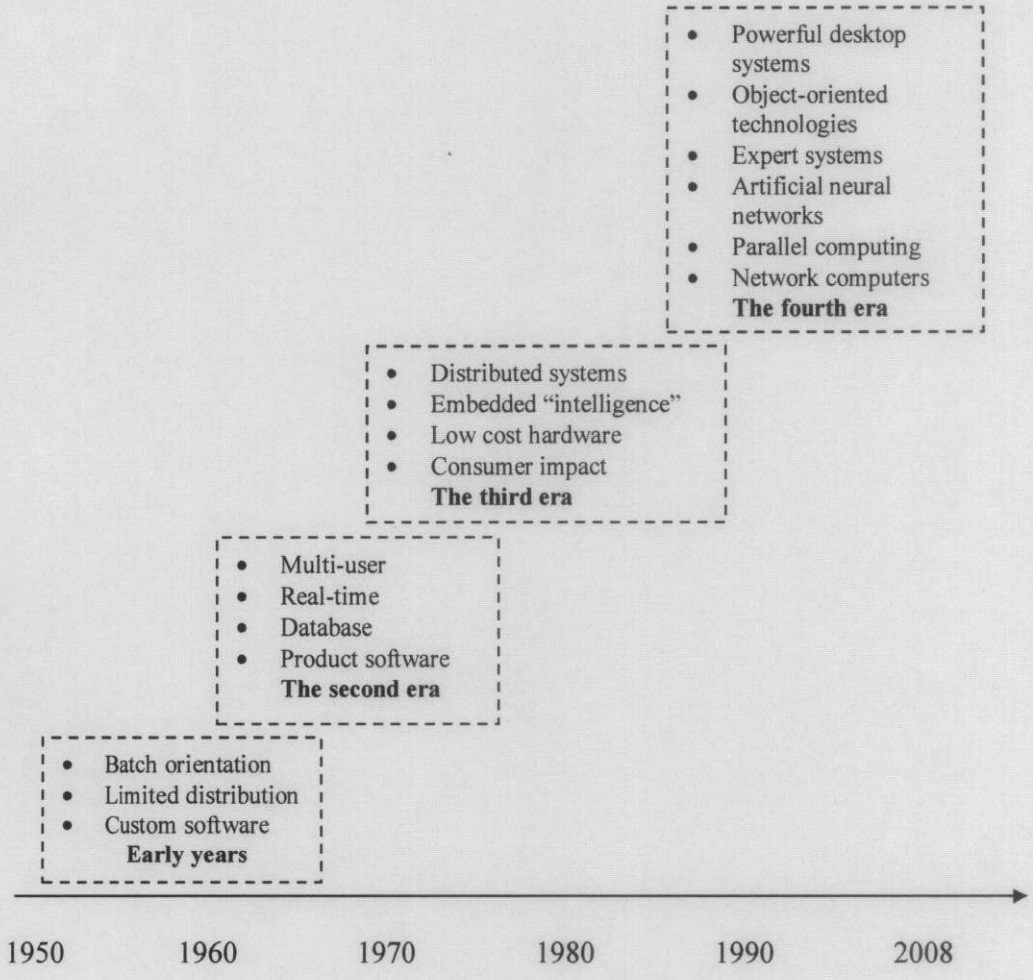


Figure 1.2: A short history of software

The series of steps that software undergoes, from concept exploration through final retirement, is termed its life cycle as shown in Figure 1.3. During this time, the product

goes through a series of phases: requirements, specification (analysis), planning, design, implementation, integration, maintenance (which is the highest cost among all these phases), and retirement (Schach S. 1997).

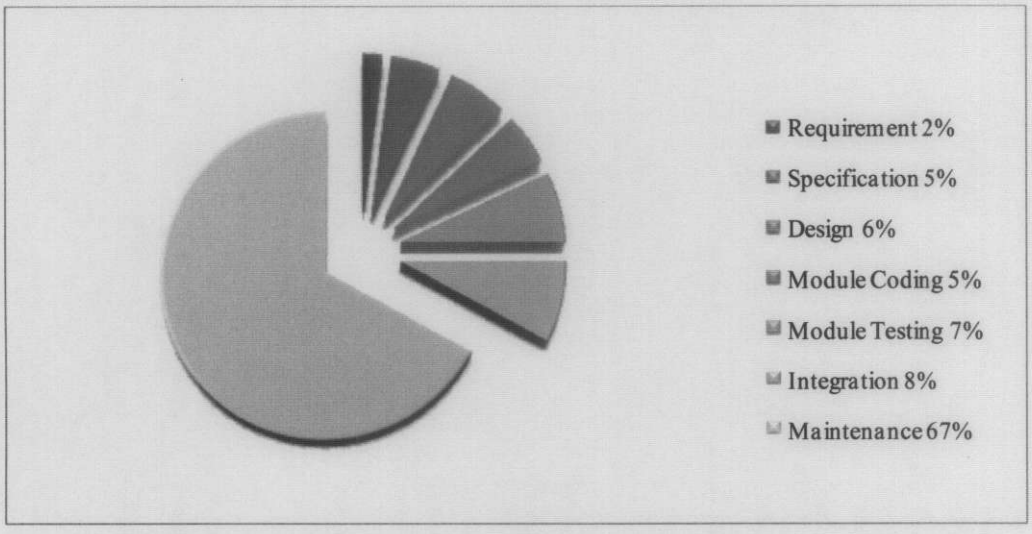


Figure 1.3: The phases of the software life cycle/software process

1. *Requirements phase*: The concept is explored and refined, and the client's requirements are elicited.
2. *Specifications (analysis) phase*: The client's requirements are analyzed and presented in the form of the specification document ("what the product is supposed to do"). Sometimes it is called the specification phase since a plan is drawn up for the software project management and the proposed development is described in detail.
3. *Design phase*: The specifications undergo two consecutive processes of architectural design (the product as a whole is broken down into modules) and detailed design (each module is designed).
4. *Implementation phase*: The various components undergo coding and testing.
5. *Integration phase*: The components are combined and tested as a whole (integration) when the developers are satisfied that the product functions correctly,

it is tested by the client (acceptance testing) implementation phase ends when the product is accepted by the client and installed on the client's computer.

6. *Maintenance phase*: All changes to the product once the product has been delivered and installed on the client's computer. It includes corrective maintenance (software repair) which consists of the removal of residual faults while leaving the specifications unchanged and enhancements (software updates) which consists of changes to the specifications and the implementation of those changes. The two types of enhancements are perfective (changes the client thinks will improve the effectiveness of the product, such as additional functionality or decreased response time) and adaptive (changes made in response to changes in the environment, such as new hardware/operating system or new government regulations).
7. *Retirement phase*: The product is removed from service. Provided functionality is no longer of use to the client.

1.2 Software Quality Assurance

The quality of software has improved significantly over the last few years and the reason for this is that companies have new techniques and technology such as the use of object-oriented development and associated Computer-Aided Software Engineering (CASE) support. In addition, however, there has a greater awareness of the importance of software quality management and the adoption of quality management techniques from manufacturing in the software industry. However, software quality is a complex concept that is not directly comparable with quality in manufacturing. In manufacturing, the notion of quality means that the developed product should meet its specification. In an ideal world this definition should be applied to all products but for software system the problems with this is as the following (Moores B 1994):

- A specification should be oriented toward the characteristics of the product that the customer wants. However, the development organization may also have equipments (such as maintainability requirements) that are not included in the specification.
- Unknown how to specify certain quality characteristics (e.g., maintain ability) in an unambiguous way.

Quality assurance (QA) is the process to define how the software quality can be achieved and how the development organization knows that the software has the required level of quality. The QA process is primarily concerned with defining or selecting standards that should be applied to the software development process or software product. As the part of QA process tools and methods to support these standards are selected and procured. The two types of standards that may be established as part of the quality assurance process are (Sommerville I. 2006):

1. Product standards: these standards are applied to the software product being developed. These include document standards, such as the structure of requirements documents; documentation standards.
2 Process standards: these standards define the process that should be followed during software development. It include definitions of specification design and validation process and a description of the documents that should be written in the path of these processes.

SQA must plan what checks to do early in the project. The most important selection criterion for software quality assurance planning is risk. Common risk areas in software development are novelty, complexity, staff capability, staff experience, manual procedures and organizational maturity. SQA staff should concentrate on those items that have a strong influence on product quality. They should check as early as possible the following (Jones M. et. al. 1997):

- Project is properly organized, with an appropriate life cycle;
- development team members have defined tasks and responsibilities;
- documentation plans are implemented;
- documentation contains what it should contain;
- documentation and coding standards are followed;
- standards, practices and conventions are adhered to;
- metric data is collected and used to improve products and processes;
- reviews and audits take place and are properly conducted;
- tests are specified and rigorously carried out;

- problems are recorded and tracked;
- projects use appropriate tools, techniques and methods;
- software is stored in controlled libraries;
- software is stored safely and securely;
- software from external suppliers meets applicable standards;
- proper records are kept of all activities;
- staff are properly trained;
- risks to the project are minimized.

Project management is responsible for the organization of SQA activities, the definition of SQA roles and the allocation of staff to those roles (Jones M. et. al. 1997).

1.3 Software Quality Control

Good quality managers aim to develop a 'quality culture' where everyone responsible for product developments is committed to achieving a high level of product quality as shown in Figure 1.4. They encourage teams to take responsibility for the quality for their work and to develop new approaches to quality improvement, while standards and procedures are the basis of quality management, experienced quality managers recognize that there are intangible aspects to software quality (elegance, readability, etc.) that cannot be embodied in standards. They support people who are interested in these intangible aspects of quality and encourage professional behavior in all team members (Daniel G. 2003).

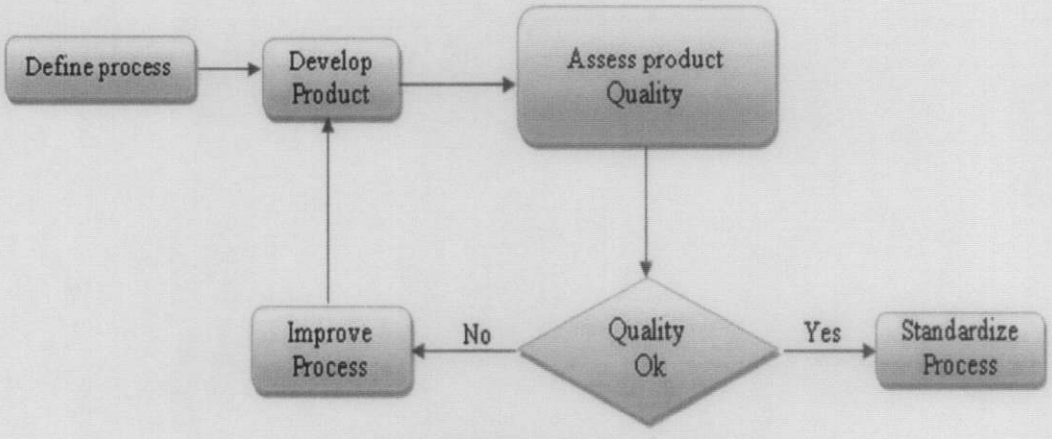


Figure 1.4: Process based quality

Variation control may be equated to quality control as shown in Figure 1.5. But how do we achieve quality control?. Quality control involves the series of inspection, reviews, and tests used throughout the software process to ensure each work product meets the requirements placed upon it. Quality control includes a feedback loop to the process that created the work product. The combination of measurement and feedback allows us to tune the process when the work products created fail to meet their specifications. A key concept of quality control is that all work products have defined, measurable specifications to which we may compare the output of each process. The feedback loop is essential to minimize the defects product (Vigder M. et. al. 1994).

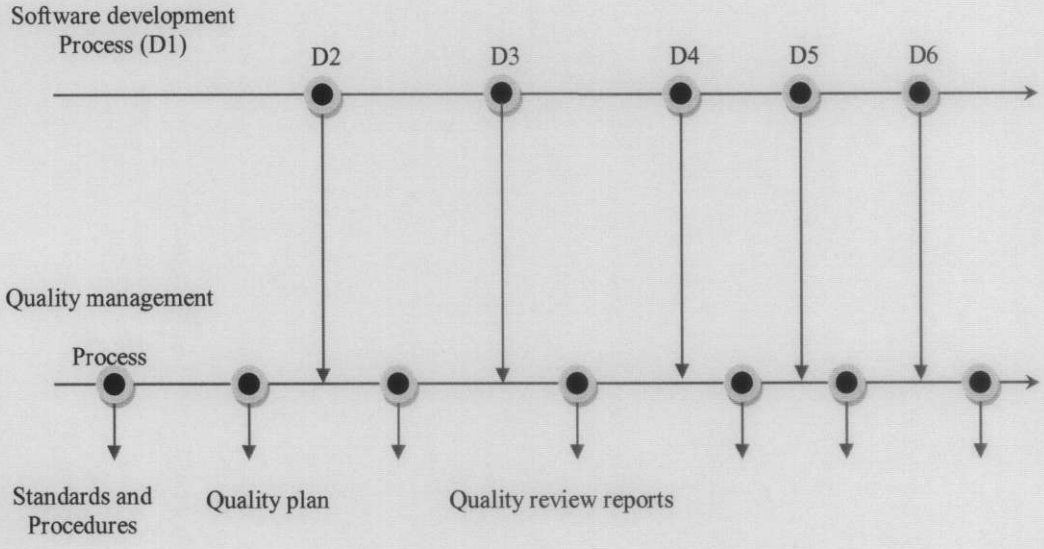


Figure 1.5: Quality control process

Testing presents an interesting anomaly for the software engineers, who by their nature are constructive people. Testing requires that the developer discard preconceived notions of the "correctness" of software just developed and then work hard to design test cases to "break" the software. Software testability is simply how easily can be tested. The following characteristics lead to testable software (Gray M. 1999):

Operability: "The better it works, the more efficiently it can be tested." If a system is designed and implemented with quality in mind, relatively few bugs will block the execution of tests, allowing testing to progress without fits and starts.

Observability: "What you see is what you test." Inputs provided as part of testing produce distinct outputs. System states and variable are visible during execution. Incorrect output is easily identified. Internal errors are automatically detected and reported. Source code is accessible.

Controllability: "The better we can control the software. The more the testing can be automated and optimized." software and hardware states and variables can be controlled directly by the test engineer. Tests can be conveniently specified automated, and reproduced.

Decomposability: "By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting." the software is built from independent modules that can be tested independently.

Simplicity: "The less there is to test, the more quickly we can test it." The program should exhibit functional simplicity (e.g., the feature set is the minimum necessary to meet requirements), structural simplicity (e.g., architecture is modularized to limit the propagation of faults), and code simplicity (e.g., a coding standard is adopted for ease of inspection and maintenance).

Stability: "The fewer the changes, the fewer the disruption to testing." changes to the software are infrequent, controlled when they do occur, and do not invalidate existing tests. The software recovers well from failure.

Understandability: "The more information we have, the smarter we will test." The architectural design and the dependencies between internal, external and shared components are well understood. Technical documentation is instantly accessible, well organized specific and detailed, and accurate. Changes to the design are communicated to testers.

Most systems eventually reach a point when questions arise about their maintainability and supportability. Some systems are supportable for years, while others have supportability problems from initial deployment. Many of these problems are indicative of insufficient resources being applied to system support. The key to having cost-effective systems is to have applied the correct quality controls during initial development and implemented good recovery strategies to existing systems. Quality controls used during maintenance may need to be different than those used when the software was created. There are a number of important issues when considering improving the quality of existing software systems. Some are (Brenda C. et. al. 2002):

- Most likely the system is being managed in a different environment than it was developed.
- Customers or users should be involved and their expectations need to be carefully considered, particularly in terms of failures or errors and availability.
- The people involved in support of the system may not be the same ones that developed it.
- How the system was developed or constructed may not necessarily be obvious to current maintainers.
- Documentation and a change management process may not be adequate.
- Planning for adequate resources and identification of their sources needs to be done.
- Integration into information architectures or modernization plans needs to be considered.

There are good reasons to consider improving the quality of existing systems. If a system is becoming difficult to support, it may well hinder an organization's ability to achieve business success. Support costs can be up to 80% of the system's overall life cycle cost; quality improvements can provide a clear return on investment (Chatzigeorgiou A. et. al. 2003).

To provide a methodology for designing, applying, and validating software quality guidelines, we recommend and briefly summarize IEEE standard 1061 (IEEE Std 1061 1998). This standard gives a process for constructing and implementing a software quality metrics framework that can be tailor-made to meet quality requirements for a particular

project and/or organization. Since the introduction of source code metrics into the discipline of software engineering, controversy has surrounded attempts to validate their usefulness as indicators and discriminators of quality. Traditional metrics such as McCabe's cyclomatic complexity, Halstead's software science metrics, and source lines-of-code, while persisting as commonly used metrics for indicating quality, unfortunately still lack conclusive evidence to support this practice. The reasons why traditional code metrics have been inadequate as measures of quality and complexity include the following(Crutchfield et. al. 1994):

- They are narrow in the scope of software characteristics they measure, and they are defined to be language independent.
- Their conception was strongly influenced by the relative simplicity of programming languages.
- They are based primarily on lexical and syntactic features of code, rather than on the semantic and structural relationships that exist among program units and smaller elements within units.
- They focus on executable statements and generally neglect the influence on complexity of data definitions and the interaction between data and computation flow.
- They have often been designed without regard to the programmers task, problem domain, or environment and ignore the stylistic characteristics of a program.

All testing activities should include processes to uncover defects during the complete life cycle of the software. The focus on full life cycle testing (as opposed to system integration testing only) is important because the cost of defects rises exponentially the later the phase of the cycle. The testing activities include, but are not limited to (Bussieck M. et. al. 2004):

Unit testing: Testing of the individual component using both black-box (input-output only) and white-box (known internal code structure) type tests.

Regression testing: Testing to determine if changes to the software or fixing a defect cause any problems to other components in the system.

System Integration Testing: Testing done to ensure that the entire product functions as intended and to specifications. The emphasis of all testing activities is again on automation and reproducibility.

Metrics: Most engineering processes involve measurements to make accurate assessments of the attributes of the product.

1.4 Fundamental of Software Testing

Software engineering is an engineering discipline which is concerned with all aspects of software production. Engineering discipline Engineers make things controlled and working under their authority. They apply theories, methods and tools where these are appropriated but they use them selectively and always try to discover solutions to problems even when there are no applicable theories and methods. Engineers also recognize that they must work to organizational and financial constrains, so they look for solutions within these constrains. Depending upon the type of process and project being implemented, the testing and evaluation of the associated software also becomes proportionally tougher to handle (Kaner C. 2004).

In most cases, the effectiveness of the testing meets short of requirement leading to downtime and project delays. Historically, software testing has been based only on technical grounds. In recent times however, a lot of improvement in the ways and means of testing software has developed. The fundamentals of software testing comprises of the following (Ali K. et. al. 2004):

- A well-documented plan, test case
- Ways and means, tools, techniques
- Human aspect involving people and the company

For any software testing to be effective and efficient, a well-documented plan, test case, tools and techniques are all important elements for quality assurance. The present testing practices suffer broadly from three main deficiencies, which are more subjective rather than objective, for example (Markku O. 1999):

- The types of test methods

- The way the method is viewed by the users
- Work culture of the organization

Present day testing practices suffer due to various reasons such as:

- Not following written procedure
- Bypassing set rules
- Lack of knowledge of the importance of QA
- 'Take it easy' attitude
- Attitude of 'Supply now, bugs in program can be fixed later'
- Poor pay and poor staffing

Measurement of the reliability of a software is facilitated by using past development data. Similar to reliability programs in industrial processes, the reliability of a software process can be defined as the probability of a failure-free program in a given environment and time period. Software errors can be classified according to the programming and hardware characteristics of the error, such as an incorrect memory-to-memory transfer caused by an error in addressing. In addition, errors can be classified according to their effect on mission success. Both types of classification are useful. However, the latter classification is much more difficult to make than the former because it is necessary to associate three items of information (Schneidewind 1975):

1) the manifestation of the error (incorrect target symbol on the display console);
2) the effect on the mission (failure to identify a hostile target); and
3) the programming cause of the error (incorrect interpretation by the program of an operator hostile target input).

Although the variability among error counts which are used to measure software quality could probably be decreased by classifying and counting errors by category, the resulting sample sizes would be significantly reduced. Consequently, as a practical matter, only a limited number of categories can be used for error classification. In Navel Tactical Data System (NTDS) software error reporting, errors are classified according to the effect on the mission as follows (Pan J. 1999):

- *High*: The program will be halted if this type of error occurs. Example: attempt to address data outside the memory address range of the computer.
- *Medium*: This type of error will cause a degradation in system performance. Example: target position is not updated with sufficient frequency.
- *Low*: This type of error will be distracting or annoying but will not normally result in degraded performance, although lowered performance could result if the operator is unable to cope with the problem. Example: the programmed refresh rate of the display console is low and causes fading of symbol displays.

1.5 Standards

ISO9000 talks only in general terms without being specific about the service or the product. It treats an organization as comprising of interconnected processes. These individual processes must satisfy the requirements of their areas mentioned in the ISO document. The development of software in a controlled manner is facilitated by ISO9000/ISO14000 along with proper procedures for project specifications, design and evaluation. Effective implementation of the software in a controlled manner should result in its development in an economical way and should include confidence among users and clients that it will meet the requirements of the project. ISO9000/ISO14000 does not say how the system has to be implemented. This task is left for software developers to establish their own rules and regulations depending upon their project, organization, etc., so as to achieve the requirements of the standard. The bottom line is that the standard mentions the basic requirements for quality management of software, to be developed with the objective of meeting the requirements of individual projects, which vary from one project to another. The development of software is not a one-man show, but rather done by a group of people. Each and every person associated with the software development is actually a contributor to quality management in one way or the other. Hence, all the responsibilities involved must be clearly documented and applied. Each group of software developers must have a clear understanding of the requirements of the project and must have smooth coordination within and outside their sub-groups. The set of rules, regulations and procedures so as to meet the

contractual requirement of the project must be well established and adhered to strictly, so as to have total control over quality assurance (Ashrafi 2003).

A quality assurance system may be defined as the organizational structure, responsibilities, procedures, and resource for implement. QA systems are created to help organization ensure their products and services satisfy customer expectations by meeting their specifications. ISO 9000 describes a quality assurance system in generic terms that can be applied to any business regardless of the products or services offered. To become registered to one of the quality assurance system models contained in ISO 9000, a company's quality system and operations are scrutinized by third-party auditors for compliance to the standards and for effective operation. Upon successful registration, a company is issued a certificate from a registration body represented by the auditors. Semiannual surveillance audits ensure continued compliance to the standard. Figure 1.6 shows the areas covered in ISO 9001. ISO 9001:2000 is the quality assurance standard that applies to software engineering. The standard contains 20 requirements that must be present for an effective quality assurance system. Because the ISO 9001:2000 standard is applicable to all engineering disciplines, a special set of ISO guidelines (ISO9000-3) have been developed to help interpret the standard for use in the software process (Sommerville I. 2006).

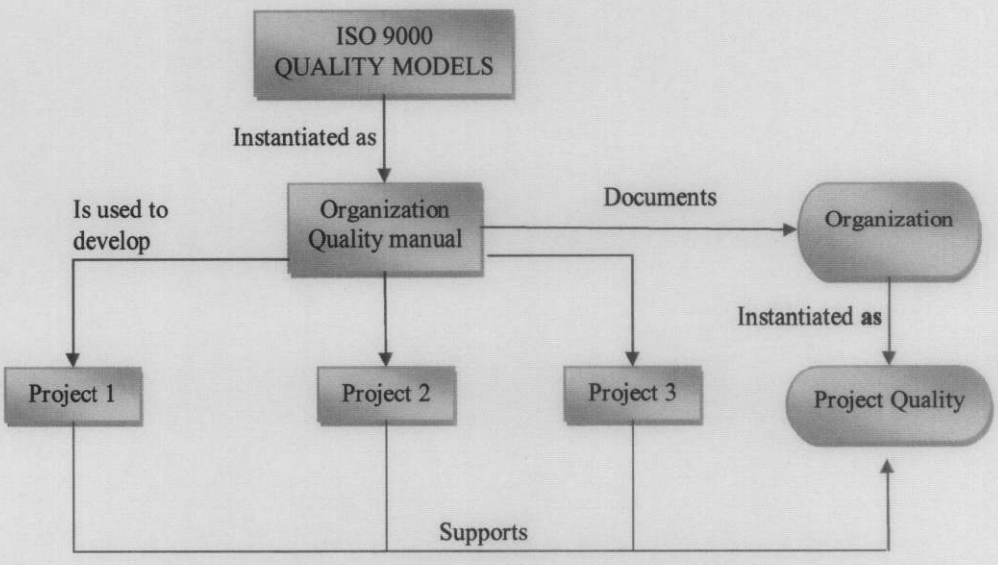


Figure 1.6: ISO 9000 and quality management

1.6 Problem Statement:

In designing there is a need to know the complexity especially for existing many programming languages such as C++, Java, and Visual Basic that are used for producing software. In this thesis, different methods for measuring software complexity specially McCabe and Halstead methods are used to measure the complexity of the algorithms written in different languages such as C++, Java, and Visual Basic. The guideline of software quality assessment based on the analysis of McCabe and Halstead metrics will help the researcher on calculating and measuring the software complexity. The conception was strongly influenced by the relative simplicity of programming languages in popular use. The focus is on executable statements and generally neglect the influence on complexity of data definitions and the interaction between data and computation flow. A software quality model which is effective in testing and improving the software quality and software houses through transitions to higher software culture is suggested. Software testing techniques, methodologies, tools, and standards can only aid in testing. Testing need to focus on maximizing 'customer satisfaction' rather than just detecting and correcting errors involved in delivered software.

1.7 Research Significance

Complexity has undesirable effects on, among others, the correctness, maintainability, and understandability of business process models. Yet, measuring complexity of business process models is a rather new area of research with only a small number of contributions. In our research, survey findings from neighboring disciplines on how complexity can be measured will be carried out. In particular, gathering insight from software engineering, and graph theory, and discuss in how far analogous metrics can be defined on business process models. The results of this research will be quite helpful to the program designers and researchers in quantifying the specific measuring tools for software quality attributes and it will lead into new techniques for measuring the complexity for software engineering. These shortcomings have led to investigate design-oriented or structural metrics; which is, metrics either can be applied to artifacts of design, or that measure only design related software characteristics, such as structural relationships.

1.8 Research Objective

The main objective of the thesis is to find out an appropriate method for software complexity measurement to improve software engineering quality, in order to make the software simple and easy to understand the logic, maintain, review and use.

The sub objectives are:

1. To obtain the specific measuring tools for software quality attributes through Cyclomatic Complexity and Halstead metrics.
2. To find out a methodology for designing, applying and validating software quality guidelines.

1.9 Research Methodology

The research divided into three parts: one theoretical and one practical part, that together result in a third concluding part. In the theory part will present what has been done in the area of software complexity and investigate the metrics that are to be evaluated. In the second part will collect tools that measure the selected metrics and apply them to a code base with well-known design quality. In the third concluding part will discuss how the results from the practical experiment together with the theoretical part are to be interpreted. The validity of the metrics and suggest strategies on how they can be used in different stages in a software development process will be discussed.

The steps of research are:

1. An extensive literature review of the subject.
2. Proposing own algorithms and techniques.
3. Specifying measurement of complexity for software.
4. Gathering information through extensive field studies in order to understand the functions and activities to improve the quality and effectiveness of the software engineering.
5. Investigating related work and study.
6. Adequate comparisons, analysis, conclusions and recommendation will accompany my work.

1.10 Research Variables

The rate of complexity suggested by McCabe and Halstead, which suitable with the new languages such as the object oriented languages need to be investigated. Perceptions of current software engineering practices were gathered by measurement of success and complexity in software projects and will study the factors influencing the success of software engineering practices.

1.11 Thesis Outline:

This thesis consists of seven chapters:

Chapter 1 introduces the software engineering and quality issues, for example software quality assurance and software quality control. Then, it gives small brief about the fundamental of software testing and the problem of statement.

Chapter 2 discusses the structure to measure the software complexity such as control-flow structure, data structure, and data-flow structure. Subsequently discusses several types of software complexity measurement such as lines of code methods, information flow metric, complexity measure (McCabe methods), Halstead methods, and cognitive weights methods.

Chapter 3 introduces the object-oriented programming and history of object-oriented programming language. Discussed different types of object-oriented languages such as JAVA, C++, C#.NET, and Visual Basic.NET. Then it compares between Microsoft's C# programming language to SUN Microsystems's JAVA programming language keyword. Then the complexity of linear search code and binary search code algorithms written in C++, Visual Basic, and Java is measured and compared.

Chapter 4 applies Halstead mathematics equation complexity through several estimated examples taken in consideration different value of operators and operands. Then, discusses the affect of execution time for three types of microprocessor such as 80286, 80486, and Pentium. Then discusses the combinational logic.

Chapter 5 introduces and discusses software complexity based on cognitive weights to analyze the robustness of the new measurement of cognitive functional size and its relationship with the physical size of software.

Chapter 6 proposes a model of software complexity that will form the basis for software quality assurance.

Chapter 7 is conclusion and future works.

العنوان:	Measuring software complexity for software engineering quality assurance
المؤلف الرئيسي:	Wohaishi, Ahmad Muhammad Ali
مؤلفين آخرين:	Fahmy, Maged A.، Al Sultanny, Yas A.(Co-Advisor، Advisor)
التاريخ الميلادي:	2009
موقع:	المنامة
الصفحات:	1 - 155
رقم MD:	736039
نوع المحتوى:	رسائل جامعية
اللغة:	English
الدرجة العلمية:	رسالة ماجستير
الجامعة:	جامعة الخليج العربي
الكلية:	كلية الدراسات العليا
الدولة:	البحرين
قواعد المعلومات:	Dissertations
مواضيع:	برامج الحاسوب، هندسة البرمجيات، تكنولوجيا المعلومات
رابط:	https://search.mandumah.com/Record/736039

Chapter 2

Literature Survey on Software Complexity

Measurement is needed for assessing the status of products, processes and resources. Every measurement activity must be in the way of achieving clearly defined goals. Understanding measurement helps to understand what is happening during development and maintenance phases of software development. Measurement should be done in order to derive models of processes and examine relationships among the process parameters. It leads to better understanding and improving software projects (Ali M. 2006).

There are many of software complexity measures, ranging from the simple, such as source lines of code, to the obscure, such as number of variable definition/usage associations. It is important to select a good subset of these measures for implementation. An important criterion for metrics selection is uniformity of application. The key idea here is "open reengineering." The reason "open systems" are so popular for commercial software applications is that the user is guaranteed a certain level of interoperability and the applications work together in a common framework. These applications can be ported across hardware platforms with minimal impact. The open reengineering concept is similar, in that the abstract models used to represent software systems should be as independent as possible of implementation characteristics such as source code formatting and programming language. Complexity measurement is a fundamental application, but open reengineering extends to other modeling techniques such as flow graphs, structure charts, and structure-based testing (Kan S. 2003). This Chapter presents the software complexity theories and summarizes the literature survey on this issue.

2.1 Structure measures of software complexity

Over the last three decades many measures have been proposed by researchers to analyze software complexity, understandability, and maintenance. Metrics were designed to analyze software such as imperative, procedural, and object-oriented programs. Software measurement is concerned with deriving a numeric value for an attribute of a software product, i.e. a measurement is a mapping from the empirical world to the formal world. From the several software metrics available there are particularly interests in studying

complexity metrics and find out how it can be used to evaluate the complexity of business processes. Software metrics are often used to give a quantitative indication of a program's complexity. However, it is not confused with computational complexity measures, whose aim is to compare the performance of algorithms. Software metrics have been found to be useful in reducing software maintenance costs by assigning a numeric value to reflect the ease or difficulty with which a program module may be understood. There are hundreds of software complexity measures that have been described and published. For example, the most basic complexity measure is the number of lines of code (LOC), simply counts the lines of executable code, data declarations, comments, and so on. While this measure is extremely simple, it has been shown to be very useful and correlates well with the number of errors in programs (Cardoso et. al. 2006).

Software Metrics are standards to determine the size of an attribute of a software product and a way to evaluate it. They can also be applied to the software process. Classification of software metrics is as the following (Klasky H. 2003):

a) *Software product metrics*: These metrics measure the software product at any stage of its development. They are often classified according to the size, complexity, quality and data dependency.
b) *Software process metrics*: These metrics measure the process regarding to the time that the project will take, cost, methodology followed and how the experience of the team members can affect these values. They can be classified as empirical, statistical, theory base and composite models.

The size of the product offers a great deal of information about the effort that went into producing it. A large module takes longer to be specified, designed, and coded than that of a smaller one. However, this view is not true; the structure of the product has a role, not only in terms of the development effort but also maintenance. We must examine the features of product structure to determine how they influence the required outcomes. Although the structure of a module is often called complexity, there are various elements of structure or complexity each having a diverse role. Structure can be seen to have three main parts (Banker R. 1989):

1. Control-flow structure
2. Data-flow structure
3. Data structure

The control flow deals with the sequence by which instructions are performed in a program. Data flow follows the trail of a data item as it is produced or handled by the program. Data structure is the organization of the data itself, independent of the program (Sofia N. 1999).

In our daily life, people always believe that simple things are reliable and easy to be repaired. So does it in the design of hardware and software reliabilities. Software measurement should be applied to guiding the process of testing, because intricate and involved software may be especially difficult to code, debug, test, and maintain (Yanming C. et. al. 2007).

Software complexity is one branch of software metrics that is focused on direct measurement of software attributes, as opposed to indirect software measures such as project milestone status and reported system failures. Ideally, complexity measures should have both descriptive and prescriptive components. Descriptive measures identify software that is error-prone, hard to understand, hard to modify, hard to test, and so on. Prescriptive measures identify operational steps to help control software, for example splitting complex modules into several simpler ones, or indicating the amount of testing that should be performed on given modules (Watson et. al. 1996).

There are two independent approaches to develop a control flow complexity metric to analyze business process: a top-down and a bottom-up approach. The top-down approach starts by formulating a set of general metrics common to various business process languages such as Business Process Execution Language (BPEL). These metrics are then applied to specific business process languages to evaluate their applicability and if necessary missing control flow elements can be added to general metric. In the second approach, the bottom-up approach start by analyzing specific business process languages and formulate specific control flow complexity metrics. Once a reasonable set of business process languages have been analyzed it is then possible to advise general metrics that can be suitably applied to the business process languages analyzed (Cardoso 2006).

Growing maintenance costs have become a major concern for developers and users of software systems. Changeability is an important aspect of maintainability, especially in environments where software changes are frequently required. The assumption that high-level design has an influence on maintainability is carried over to changeability and investigated for that quality characteristics. The approach taken to assess the changeability of an object-oriented (OO) system is to compute the impact of changes made to classes of the system. A change impact model is defined at the conceptual level and mapped on the C++ language. In order to assess the practicality of the model on large industrial software systems, an experiment involving the impact of one change is carried out on a telecommunications system (Chaumun et. al. 2002).

There are five classic approaches being used for software measurement. Each of these approaches will be discussed in the following Sections:

1. LOC (lines of code) which stand for software's length.
2. IF (information flow) which regards amount of information flow.
3. McCabe which concerns loops.
4. Halstead which concerns with numbers of operators and operands.
5. Cognitive weights method.

2.2 Line of Code Method

The predominant definition for lines of code (LOC) is 'a line of code is line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line'. LOC is used as an indicator of software reliability and maintenance. Empirical data from research done by various researchers show that LOC metric has a high correlation to system reliability and maintenance in spite of its simplicity. It has been used extensively in spite of all the criticisms. However, LOC is highly language specific, therefore it can be used for comparison of code in the same language. Also, is not applicable for visual languages because the notion of LOC may not be meaningful. It is not a suitable indicator at the design phase when the code has not been developed because it is not a good indicator of structural complexity. Possibly the most commonly used software measurement is the LOC (lines of code) metric. It is easy to calculate and can be interpreted

in a variety of ways to describe source code. Many programmers use the LOC metric to depict a program's size, complexity, or programming effort. There are few different ways of calculating the LOC metric. Some involvs counting only non white spaces lines, while others exclude comments (Garcia 2008).

The most commonly used measure for the length of a code source of a program is the number of lines of code (LOC). The abbreviation NCLOC is used to represent a noncommented source line of code. NCLOC is also sometimes referred to as effective lines of code (ELOC). NCLOC is therefore a measure of the uncommented length. The commented length is also a valid measure, depending on whether or not line documentation is considered to be a part of programming effort. The abbreviation CLOC is used to represent a commented source line of code. By measuring NCLOC and CLOC separately we can define:

$$total\ length(LOC) = NCLOC + CLOC \quad ...(2.1)$$

It is useful to separate comment lines and other lines (NCLOC). KLOC is used to denote thousands of lines of code. Generally, is better to address how the followings are handled:

1) blank lines
2) comment lines (CLOC)
3) data declarations or other commands
4) lines that contains several separate instructions
5) program lines generated by a tool

The entity CLOC/LOC is then a measure of the density of comments in a program. The purpose of software is to provide certain functionality for solving some specific problems or to perform certain tasks. Efficient design provides the functionality with lower implementation effort and fewer LOCs. Therefore, using LOC data to measure software productivity is like using the weight of an airplane to measure its speed and capability. In addition to the level of languages issue, LOC data do not reflect no coding work such as the creation of requirements, specifications, and user manuals (Chis 2008).

The most basic complexity measure, the number of lines of code LOC counting the size of software is usually applied to executable sentence. The major concern in the original

method is the use of software size as the only complexity factor, due to the limitation of detailed information available during the early stage of a program. It is likelihood that the larger the size of a program is, the more the defects there could be. The LOC metrics does represent some aspects of software complexity, but it is a rough method which does not provide the necessary fidelity for a software reliability assessment. LOC should be a referenced factor of software complexity (Yanming C. et. al. 2007).

The number of lines of code, does not meet the open reengineering criterion, since it is extremely sensitive to programming language, coding style, and textual formatting of the source code (Waston et. al. 1996).

2.3 Information Flow

The information flow metric is a structure metric that measures the sources (fan-in) and destinations (fan-out) of all data related to a given software component. These factors are used to compute the communication complexity of the component, which is taken as a measure of the strength of its communication relationship with other components. The fan-in consists of all function parameters and global data structures from which the function retrieves its information, while the fan-out consists of the return values from function calls and the global data structures that the function updates. Another important structure metric draws attention to the "ripple effect" that occurs when a change to one component causes a need for change in other components. In this stability measure, the flow of data through parameters and global variables is used to identify the components which could be affected by a change in a particular component both of these metrics reflect a quality risk. If program components are designed to be too dependent on each other, flexibility is lost. This means the program is more difficult to change in the future. Other metrics that can be taken once the analysis and design phases are complete are the number of documents produced, the size of these documents, and several metrics that analyze the contents of the documents. Using these, a project could determine whether the analysis and design quality is sufficient, or if more work is needed (Fagerholm 2007).

Measures of control-flow structure have been the interest of software metrics work since the beginning of the 1970's. This approach defines a connection between two components

to exist if there is a possible flow of information from one to the other. The information flow metrics are based on the structure created by the flow of information within a system. This metric provides a method for quantitatively assessing the balance between structure and efficiency. It captures the intuition that a large but well-structured system may be less complex than a smaller, but poorly structured system. Where interface is complex, there exists a far greater potential for a simple maintenance change to ripple through the system and impact other components. The procedure for calculating the metrics is as follows (Areejit P. et. al. 2005):

1. Identify the information flows in a system:
   - *Local Direct Flow*: if a module invokes a second module and passes information to it or if the invoked module returns a result to the caller
   - *Local Indirect Flow*: if the invoked module returns information, which is subsequently passed to another invoked module.
   - *Global Flow*: if there is flow of information from one module to another, via a global data structure.
2. Calculate fan-in and fan-out:
   - *Fan-in*: number of local flows that terminate at a module, plus the number of data structures from which information is retrieved.
   - *Fan-out*: number of local flows that emanate from a module plus the number of data structures that are updated by that module
3. Calculate complexity of each module:

The most commonly used method is fan-in and fan-out metrics. Henry and Kafura in 1981 defined the structure complexity as

$$C=(\text{fan-in} * \text{fan-out})^2 \quad \ldots(2.2)$$

where C stands for the amount of information flow in a module, fan-in is a module's inputs, and fan-out is the amount of return values and variables changed (Yanming C. et. al. 2007).

The intra-module complexity is given by the term length which can be defined as the number of lines of text in the source code for the module while the inter-module complexity is given by the term fan-in * fan-out. The inter-module complexity term represents the total possible number of combinations of an input source to an output destination, i.e. the total possible combinative paths of information flow. The metric thus takes into account both the intra-module complexity and the inter-module complexity. Raising the equation to the power of 2 is done to express the complexity as a nonlinear function, which means that the information flows contribute to the physical complexity of an entity in a nonlinear relation with length. It also reflects the fact that it is more difficult to understand the interactions among the entities than the length of the code. The control flow measures are typically modeled using directed graphs as shown in Figure 2.1, with each node representing a program statement and each arc the flow of one statement to the next. These directed graphs are known as control graphs or flow graphs.

However, instead of using the typical length measure, which involves counting the number of statements in the module, McCabe's cyclomatic complexity and data complexity have been used. This is because McCabe's cyclomatic complexity has shown greater correlations to errors and programming time than the simple length metric. McCabe method will be discussed in the following Section.

```
10    INPUT P
20    Div = 2
30    Lim=INT(SQR(P))
40    Flag = P/Div- INT(P/Div)
50    If Flag = 0 OR Div = Lim THEN 80
60    Div = Div +1
70    GO TO 40
80    IF Flag <> 0 OR P>4 THEN 110
90    Print Div
100   GO TO 120
110   Print P
120   End
```

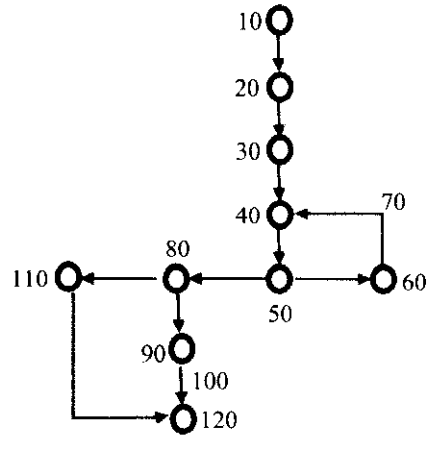


Figure 2.1: A program and its associated flow graph

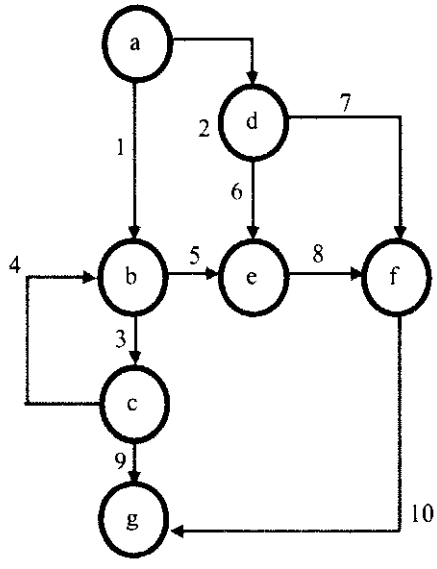
2.4 McCabe Method

Thomas McCabe introduced his graph-theoretic approach for measuring program complexity. His complexity measurement calculates total number of *possible* control paths through a program, using a control graph (Garcia 2008).

McCabe (1976) proposed "A complexity measure" that brought forward the idea of cyclomatic complexity for the first time, and it basically measures decision points or loops of the program. Cyclomatic complexity utilizes a graph, which is derived from code. The formula defined as:

$$MC=V(G)=e-n+2p \quad \ldots(2.3)$$

where e is the number of edges, n is the number of nodes, and p is the number of connected components as shown in Figure 2.2. McCabe's cyclomatic complexity metrics measures software complexity in program's structure, but it neglects the fact that length of a program is a factor of complexity (Yanming C. et. al. 2007).



McCabe cyclomatic complexity (MC) = edges - nodes + 2 = 10 – 7 + 2 = 5

Figure 2.2: Example for McCabe's cyclomatic number

The complexity measure approach is to measure and control the number of paths through a program. This approach, however, immediately raises the following nasty problem: "Any program with a backward branch potentially has an infinite number of paths." Although it is possible to define a set of algebraic expressions that give the total number of possible paths through a (structured) program using the total number of paths has been found to be impractical. Because of this the complexity measure developed here is defined in terms of basic paths that when taken in combination will generate every possible path (McCabe 1976).

McCabe (1976) did not provide a clear definition (characterization and meta-model) of the complexity attribute: this does not facilitate the understanding of this measure and it is challenging to analyze McCabe interpretation of software complexity. Similarly while defining a numerical assignment rule, McCabe did not provide the properties of the numerical representation. Understanding of related measurement concepts, and in particular of the cyclomatic number in graph theory, is therefore important to understand the input to the design of the number proposed by McCabe (1976). A key contribution of McCabe has been his attempt at transposing into software the cyclomatic number from graph theory. McCabe, while using the term 'complexity', does not provide his definition of complexity, nor of the attribute itself, nor of his direct characterization. His approach is basically to mapping of the concepts that selected from graph theory into his view of software as a control flow graph (Abran A. et. al., 2004).

The McCabe cyclomatic complexity measure is so versatile and widely used that it is often referred to simply as "complexity," and recommend it as the foundation of any software complexity program. Since it is based purely on the decision structure of the code, it is uniformly applicable across projects and languages and is completely insensitive to cosmetic changes in code. Many studies have established its correlation with errors, so it can be used to predict reliability. More significantly, studies have shown that the risk of errors jumps for functions with a cyclomatic complexity over 10, so there's a validated threshold for reliability screening. Also, this assessment can be performed incrementally during development and can even be estimated from a detailed design. For an individual

software module, the programmer can easily calculate cyclomatic complexity manually by counting the decision constructs in the code. This allows continuous control during a project, so that unreliable code is prevented at the unit development stage. Compliance can be verified at any stage of the project using automated tools. A final benefit of cyclomatic complexity is that it gives a precise verifiable testing prescription that the more complex therefore error-prone a piece of software is, the more testing it requires (Thomas J. 1994).

Although no one would argue that the number of control paths relates to code complexity, some argue that this number is only part of the complexity picture. According to McCabe, a 3,000-line program with five IF/THEN statements is less complex than a 200-line program with six IF/THEN statements. Many programmers find this assertion hard to swallow. McCabe and others found a high correlation between programs with high failure rates and high cyclomatic complexity (Lou M. 1997).

2.5 Halstead's Method

In 1976 Maurice Halstead made an attempt to capture notions on size and complexity beyond the counting number of operators and number of operands of the program (Magnus A. et. al. 2004). Halstead's metrics are related to the areas that were being advanced in the seventies, mainly psychology literature. Although his metrics are often referenced to in software engineering studies and they have had a great impact in the area, the metrics have been a subject of criticisms over the years. Although his work has had a lasting impact, Halstead's methods, or measures, provide an example of confused and inadequate measurement. However, other studies have empirically investigated the metrics and found them (or parts of them) to be good maintainability predictors. One metric that uses parts of Halstead's metrics is the maintainability index metric. Before defining the metrics, Halstead began defining a program P as a collection of tokens, classified as either operators or operands. The basic metrics for these tokens where:

- μ_1 = number of unique operators
- μ_2 = number of unique operands

- N_1 = total occurrences of operators
- N_2 = total occurrences of operands

For example the statement: f(x) = h(y) + 1, has two unique operators (= and +) and two unique operands (f(x) and h(y)).

For a program P, Halstead defined the following metrics:

The length N of P: $N = N_1 + N_2$ ...(2.4)

The vocabulary μ of P: $\mu = \mu_1 + \mu_2$ ...(2.5)

The volume V of P: $V = N * \log_2 \mu$ ...(2.6)

The program difficulty D of P: $D = (\mu_1 \div 2) * (N_2 \div \mu_2)$ ...(2.7)

The effort E to generate P is calculated as: $E = D * V$. ...(2.8)

The volume, vocabulary and length value can be viewed as different size measures. Take the vocabulary metric for instance: It calculates the size of a program very different from the LOC metric and in some cases it may be a much better idea to look at how many unique operands and operators a module has than just looking at lines of code. If, let's say a class, consists of four methods (operands): A, B, C and D. The LOC and vocabulary metrics will yield similar results, but if all four operands where implemented as A (four identical methods) the LOC metric would stay unchanged while the vocabulary metric is divided by four. The vocabulary metric shows the fact that: it is easier to understand a class with four identical methods than one with four different. When it comes to the program difficulty and effort measure, we will find them to be invalidated prediction measures, and that the theory behind them has been questioned repeatedly (Magnus A. et. al. 2004).

The Halstead (1976) Software Science metrics are a significant step up in value. By counting the number of total and unique operators and operands in the program, measures are derived for program size, programming effort, and estimated number of defects. Halstead metrics are independent of source code format, so they measure intrinsic attributes of the software. Since different languages have different sets of operators, it isn't

immediately obvious that these measures can be applied across languages, but there's a "language level" measure that can help with conversion. Halstead metrics are a bit controversial, especially in terms of the psychological theory behind them, but they have been used productively on many projects. The main drawback is that the mathematical formulas of the major Halstead metrics are significantly removed from the code, so there isn't a strong prescriptive component. Halstead metrics are very useful for identifying computationally-intensive code with many dense formulas, which represent potential sources of error that other complexity measures are likely to miss (Waston et. al. 1994).
Halstead's metrics are most often used during code development in large projects in order to track complexity trends. A spike in Halstead metrics can signify a highly error-prone module, for example. Use metrics to compare two high-level representations of a program, both with the same semantics. Halstead's metrics do not lend themselves well to this problem. Halstead's program volume metric is a measure of the minimum number of bits required for coding a program. In the case of Java, non-local variables (either class fields or static fields) and method names are preserved in the compiled byte code. A common Java obfuscation technique is to rename these identifiers, often with shorter and more incomprehensible names. This reduces the program volume but also reduces the ability of a decompile to recover the full cognitive representation of the original program. Indeed, many metrics are designed to compare large software projects in a very abstract way in order to predict maintainability, reliability and/or programming effort (Naeem N. 2006).

2.6 Cognitive Weights Method

The cognitive weight of a software component without nested control structures is defined as the sum of the cognitive weights of its control structures according to Table 2.1. It seems to be a promising approach to use the ideas to define cognitive weights for Business Process Models (BPMs). Table 2.1 should be tailored to the needs of BPMs: While recursion does not play any role in business process modeling, it is necessary to consider other concepts like cancellation or the multi-choice-pattern. The idea behind cognitive weights is to regard basic control structures as patterns that can be understood by the reader as a whole. This approach is based on automatically finding well-known architectural

patterns in a Unified Modeling Language (UML) model. The idea behind this approach is that well-documented patterns have been found highly mature and using them helps to improve code quality, understandability and maintainability. Obviously, this assertion should be regarded with care: Architectural patterns are only useful if they are used by experienced programmers in the right way, and an extensive use of patterns does not have to mean anything for the quality of the code. So, if the approach is used, a deep knowledge about the patterns and their correct usage is necessary. However, does not only discuss the use of "good" design patterns, it also recognizes so called anti-patterns, i.e. commonly occurring solutions to a problem that are known to have negative consequences. If such an anti-pattern is found in the code, this can be regarded as a sign of bad programming (Gruhn V. et. al. 2006).

Table 2.1: Basic control structures and its cognitive weights

Control Structures	***Wi***
Sequence (an arbitrary number of statements in a sequence without branching)	1
Call of user-defined function	2
Branching with if-then or if-then-else	2
Branching with case (with an arbitrary number of selectable cases)	3
Iteration (for-do, repeat-until, while-do)	3
Recursive function call	3
Execution of control flows in parallel	4
Interrupt	4

The existing measures for software complexity can be classified into two categories: the macro and the micro measures of software complexity. Major macro complexity measures of software have been proposed by Basili (1980) and by Kearney J. et al (1986). The former considered software complexity as "the resources expended"(Basili V. 1980). The latter viewed the complexity in terms of the degree of difficulty in programming (Kearney J. et. al. 1986). The micro measures are based on program code, disregarding comments and stylistic attributes. This type of measure typically depends on program size, program

flow graphs, or module interfaces such as Halstead's software science metrics and the most widely known cyclomatic complexity measure developed by McCabe. However, Halstead's software metrics merely calculate the number of operators and operands; they do not consider the internal structures of software components; while McCabe's cyclomatic measure does not consider I/Os of software systems. In cognitive informatics, it is found that the functional complexity of software in design and comprehension is dependent on three fundamental factors: internal processing, input and output. Cognitive complexity takes into account both internal structures of software and the I/Os it processes (Shao J. 2003).

Complexity measure based on weighted information count of a software and cognitive weights has been developed by (Kushwaha D. et. al. 2005). Basic control structures (BCS) such as sequence, branch and iteration are the basic logic building blocks of any software and the cognitive weights (W_c) of a software is the extent of difficulty or relative time and effort for comprehending a given software modeled by a number of BCS's. These cognitive weights for BCS's measure the complexity of logical structures of the software. Either all the BCS's are in a linear layout or some BCS's are embedded in others. For the former case, sum the weights of all the BCS's and for the latter, cognitive weights of inner BCS's are multiplied with the weight of external BCS's (Kushwaha D. et. al. 2006).

Shao and Wang (2003) defined the cognitive weight as a metric to measure the effort required for comprehending a piece of software. Based on empirical studies, they defined cognitive weights for basic control structures. Table 2.1 shows the basic control structures and its cognitive weights which are a measure for the difficulty to understand a control structure. The cognitive weight of a basic control structure is a measure for the difficulty to understand this control structure.

العنوان:	Measuring software complexity for software engineering quality assurance
المؤلف الرئيسي:	Wohaishi, Ahmad Muhammad Ali
مؤلفين آخرين:	Fahmy, Maged A.، Al Sultanny, Yas A.(Co-Advisor، Advisor)
التاريخ الميلادي:	2009
موقع:	المنامة
الصفحات:	1 - 155
رقم MD:	736039
نوع المحتوى:	رسائل جامعية
اللغة:	English
الدرجة العلمية:	رسالة ماجستير
الجامعة:	جامعة الخليج العربي
الكلية:	كلية الدراسات العليا
الدولة:	البحرين
قواعد المعلومات:	Dissertations
مواضيع:	برامج الحاسوب، هندسة البرمجيات، تكنولوجيا المعلومات
رابط:	https://search.mandumah.com/Record/736039

Chapter 3

Implementation of Measuring Object-Oriented Programming Complexity

3.1 Introduction

In this chapter, a brief description on programming languages & Object-oriented programming (OOP) and the development history is given. Then, two case studies to measure software complexity are implemented and discussed. These two cases are linear search and binary search algorithms. Each of them are implemented through writing programs using C++, Java, and Visual Basic object-oriented programming languages. Methods of measuring software complexity are used and evaluated. These methods include: Counting LOC without comments, measured LOC+ comments, McCabe complexity, Halstead method, and file size of program.

Object-oriented programming is a programming paradigm that uses "objects" and their interactions to design applications and computer programs. OOP may be seen as a collection of cooperating *objects*, as opposed to a traditional view in which a program may be seen as a group of tasks to compute ("subroutines"). In OOP, each object is capable of receiving messages, processing data, and sending messages to other objects. Each object can be viewed as an independent little machine with a distinct role or responsibility. The actions or "operators" on the objects are closely associated with the object. For example, in OOP, the data structures tend to carry their own operators around with them (or at least "inherit" them from a similar object or "class"). The traditional approach tends to view and consider data and behavior separately (Woodmanl M. 1997). Programming techniques may include features such as, encapsulation, modularity, polymorphism, and inheritance. It was not commonly used in mainstream software application development until the early 1990s. Many modern programming languages now support OOP. Object-oriented programming can trace its roots to the 1960s. As hardware and software became increasingly complex, researchers studied ways in which software quality could be maintained. (Schach S. 2002).

3.2 Most Common Used Programming Languages and History

The concept of objects and instances in computing had its first major breakthrough with the PDP-1 system at MIT which was probably the earliest example of capability based architecture. Another early example was Sketchpad made by Ivan Sutherland in 1963; however, this was an application and not a programming paradigm. Objects as programming entities were introduced in the 1960s in Simula 67, a programming language designed for making simulations. The idea occurred to group the different types of ships into different classes of objects, each class of objects being responsible for defining its own data and behavior. Such an approach was a simple extrapolation of concepts earlier used in analog programming (Benussi L. 1995).

The Smalltalk language, which was developed at Xerox PARC in the 1970s, introduced the term Object-oriented programming to represent the pervasive use of objects and messages as the basis for computation. Smalltalk creators were influenced by the ideas introduced in Simula 67, but Smalltalk was designed to be a fully dynamic system in which classes could be created and modified dynamically rather than statically as in Simula 67 (Jorgensen P. 2002).

Object-oriented programming developed as the dominant programming methodology during the mid-1990s, largely due to the influence of C++. Its dominance was further cemented by the rising popularity of graphical user interfaces, for which object-oriented programming is well-suited. An example of a closely related dynamic GUI library and OOP language can be found in the Cocoa frameworks on Mac OS X, written in Objective C, an object-oriented, dynamic messaging extension to C based on Smalltalk. Object-oriented features have been added to many existing languages during that time, including Ada, BASIC, Lisp, Fortran, Pascal, and others. Adding these features to languages that were not initially designed for them often led to problems with compatibility and maintainability of code (Fateman R. 2000).

In the past decade Java has emerged in wide use partially because of its similarity to C and to C++, but perhaps more importantly because of its implementation using a virtual machine that is intended to run code unchanged on many different platforms. This last feature has made it very attractive to larger development shops with heterogeneous

environments. Microsoft's .NET initiative has a similar objective and includes/supports several new languages, or variants of older ones (Gopal N. 2004).

3.2.1 C and C# Language

C is high-level programming language, which was developed in the 1970s in the Bell Laboratory. C is the third try after two other similar programming languages, BCPL (Basic Combined Programming Language) and B, did not meet expectations. C was used to write the UNIX Operating System and works well within that operating system. C, which is easier than assembly language but harder than other third-generation programming languages, is still widely used by a large number of programmers.

C# (pronounced C Sharp) is a multi-paradigm programming language that encompasses functional, imperative, generic, object-oriented (class-based), and component-oriented programming disciplines. It was developed by Microsoft as part of the .NET initiative and later approved as a standard by ECMA (ECMA-334) and ISO (ISO/IEC 23270) (Schildt H. 1998).

3.2.2 Visual Basic, Early Versions

Although Visual Basic was not the first programming language designed to develop a Windows application, it is perhaps the first Visual programming environment that has been introduced to programmers. Since Visual Basic syntax is based on the early version of BASIC, BASIC-A was a version that was developed for IBM™PCs. Later on, Microsoft developed GW-BASIC, which was bundled with MS-DOS Operating System for IBM compatible machines. GW-BASIC was replaced by QBASIC (stands for Quick Beginner's All-purpose Symbolic Instruction Codes). QBASIC accompanied the Windows 95 Operating System package. Along with the development of Windows 3.0 in 1991, the first official version of Visual Basic was developed. Visual Basic's drag and drop features that ran under the Windows operating systems created an easier development environment. Although Visual Basic 6.0 became a very popular programming language, it was often criticized for not being a fully object-oriented language. The .NET version of this

programming language has corrected all of the object-oriented shortcomings found in previous versions of Visual Basic (River C. 2006).

3.2.3 HTML Language

The *Internet* is a collection of computer networks that connects millions of computers around the world. The *World Wide Web* is a client/server technology used to access a vast variety of digital information from the Internet. Using a software client called a *Web browser*, such as Microsoft® Internet Explorer, and a modem or other connection to an Internet Service Provider (ISP), we can easily access text, graphics, sound, and other digital information from practically any computer in the world that is running the appropriate server software on the Internet.

The traditional stand-alone programming paradigms couldn't meet the demands of this new communication media. As a result, a new language was developed to facilitate communications among users. The developers of this new language intended to keep it simple and easy to learn. HTML, which stands for *Hyper Text Markup Language*, was developed in the 1990s. This language uses tags for specific instructions, which are translated into meaningful Web pages by special software called browsers (River C. 2006).

3.2.4 JavaScript

Although JAVA and JavaScript share a lot of similarities in both name and structures, they are two different programming languages. JAVA is a stand-alone programming language that can run on any platform. JavaScript, on the other hand, is a scripting language that must be included within HTML codes and be processed by browsers. JavaScript was developed by Netscape® in the 1990s to allow Web developers to add more interactive features to their Web sites (Campione M. et. al. 2001).

3.2.5 VBScript

VBScript is a Microsoft Scripting language that is based on the Visual Basic programming language and shares some similarities. Like JavaScript, VBScript was designed to help

Web developers add more interactive functions to their Web pages. VBScripts can be viewed by the Internet Explorer browser, but Netscape doesn't support it. Due to such limitations, VBScript is not as commonly used as JavaScript.

3.2.6 Object-Oriented Programming

The terms that are associated with the object-oriented programming are (Forouzan B. et. al. 2001):

i. Object

Object-oriented programming refers to a program that uses *objects*, such as Textboxes, Buttons, and Labels. Tools or controls are used to create these objects.

ii. Attributes or Properties

Attributes or properties are used to identify objects. Each object must have a name, color, size, and other characteristics. Each object has several attributes or properties. A Textbox, for example, has properties such as Name, BackColor, Size, Font, BorderStyle, and Text.

iii. Methods

Each object is designed to perform specific operations or actions. These actions are called *methods*. Objects have standard methods that can be used by the programmers to do specific tasks. For example, an object such as a button can have methods such as Refresh, Hide, Show, Focus, and so on.

iv. Events

Objects can react to the user's input according to the *event* that the developer has programmed for. For example, Click is one of Button's many events that can be used by the user to perform a task. The programmer has placed a set of codes within the event subprocedure of the Exit button to fulfill certain request.

v. Classes and Instances

One of the important elements of object-oriented programming is *class*. Think of a class as a rubber stamp you have made for your name, address, and zip code. Any time you use this rubber stamp, an identical instance of the information will be printed on paper. The class is like a rubber stamp or template that maintains the properties and methods needed for each object.

vi. Encapsulation

The objects communicate with the users through the interface that is designed by the programmer. The user shouldn't know what properties of the object have been used or what kind of processing takes place behind the object. *Encapsulation* is used to hide the properties and methods of the object.

vii. Inheritance

Inheritance will enable you to reuse an existing class. The terms *parent class* and *child class* refer to the original and duplicated classes.

viii. Polymorphism

Poly in Greek means "many," and *morph* means "forms." This means different objects that have methods and properties with the same name react to them differently. For example, two different objects could have similar methods. Although the methods' names are the same, they function differently when you execute the program.

3.2.6.1 Why Object-Oriented?

The industry is moving toward object-oriented design. Although the move was slow in the beginning, it has received positive acceptance in recent years. However, with the popularity of object-oriented programming languages such as JAVA, C++, C#.NET, and Visual Basic.NET, object-oriented programming languages prove to be faster to learn and easier to maintain. In addition, object-oriented programming languages provide a way to break large

and sometimes difficult-to-manage programming projects into smaller modules that can be managed easily (Dale N. 2000).

3.2.6.2 Object Oriented Programming Terminology

Class: A class is a definition for a combination of data and procedures which operate on those procedures. Instances of other classes can only access that data or those procedures via specified interfaces. A class acts as a template when creating new instances. That is, a class does not hold any data, the data is held in the instance. However, the class specifies what data will be held. The relationship between a class, its superclass and any subclasses is illustrated in Figure 3.1.

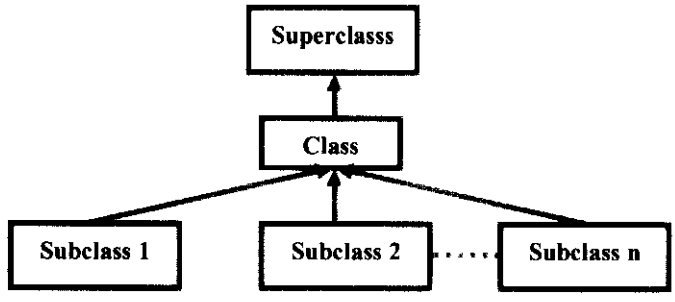


Figure 3.1: The relationship between Class, Superclass and Subclass

Subclass: A subclass is a class which inherits from another class. For example, Undergraduate-Student was a subclass of Student. Subclasses are of course classes in their own right. The term subclass merely indicates what is inherited by what. Any class can have any number of subclasses.

Superclass: A superclass is the parent of a class. It is the class from which the current class inherits. For example, Student class was the superclass of Undergraduate-Student.

Instance / Object: An instance is an example of a class. All instances of a class possess the same data variables but have their own data in these data variables. Each instance of a class will also respond to the same set of requests.

Instance variable: This is the special name given to the data which is held by an object. The "state" of an object at any particular moment relates to the current values held by its instance variables. Figure 3.2 illustrates a definition for a class in pseudo code.

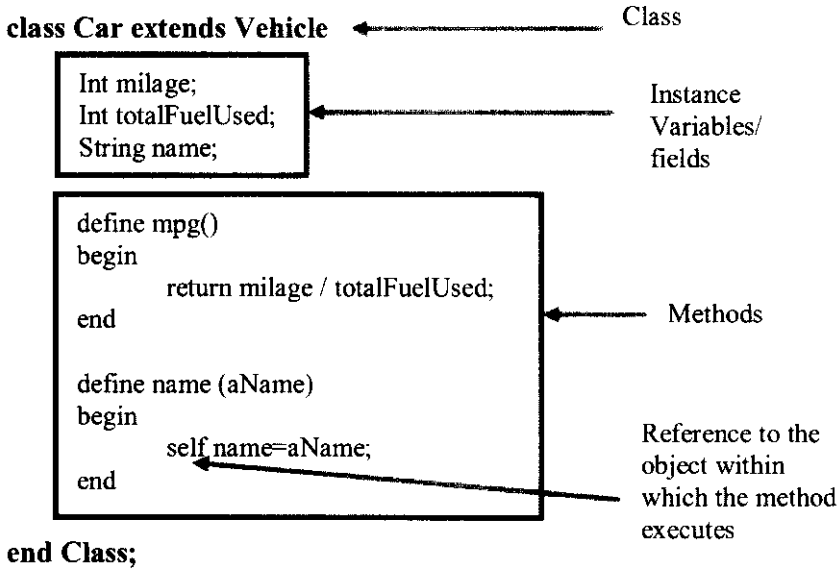


Figure 3.2: A pseudo code definition of a class

Method: Method is the name given to a procedure defined within an object. The name stems from its use in early versions of Smalltalk where it was a method used to get an object to do something\or return something.

Message: This is a request from one to another requesting some operation or order.

Self: This is reference to the object within which the method is executing.

Single/Multiple inheritance: Single and multiple inheritance refer to the number of superclasses that a class can inherit from.

3.2.7 Comparison Between C# and JAVA Keywords

In this section keywords in C# and Java languages are compared as an example to show if the number of keywords affect programming complexity. There are a large number of syntactic similarities between Java and C#. Almost every Java keyword has a C# equivalent. Table 3.1 shows the comparison between Java and C# keywords (Obasanjo D. 2007):

Table 3.1: Java and C# keywords

C# keyword	Java keyword	C# keyword	Java keyword	C# keyword	Java keyword	C# keyword	Java keyword
abstract	abstract	Explicit	N/A	object	N/A	this	This
as	N/A	Extern	native	operator	N/A	throw	Throw
base	Super	Finally	finally	out	N/A	true	True
bool	boolean	Fixed	N/A	override	N/A	try	try
break	break	Float	float	params	N/A	typeof	N/A
byte	N/A	For	for	private	private	unit	N/A
case	case	Foreach	N/A	protected	N/A	ulong	N/A
catch	catch	Get	N/A	public	public	unchecked	N/A
char	char	Goto	goto	readonly	N/A	unsafe	N/A
checked	N/A	If	if	ref	N/A	ushort	N/A
class	class	Implicit	N/A	return	return	using	import
const	const	In	N/A	sbyte	byte	value	N/A
continue	continue	Int	int	sealed	final	virtual	N/A
decimal	N/A	Interface	interface	set	N/A	void	void
default	default	Internal	protected	short	short	volatile	volatile
delegate	N/A	Is	instanceof	sizeof	N/A	while	while
do	do	Lock	synchronized	stackalloc	N/A	:	extends
double	double	Long	long	static	static	:	implements
else	else	namespace	package	string	N/A	N/A	strictfp
enum	N/A	New	new	struct	N/A	N/A	throws
event	N/A	Null	null	switch	switch	N/A	transient

From Table 3.1 found that there are 35 keywords with similar identification in both languages C# and Java. Nevertheless, there are 15 keywords having different identification from both languages for example *base* in C# language is represented with *super* in Java language. Just like Java, C# has a single rooted class hierarchy where all classes in C# are subclasses of System.Object the same way all Java classes are subclasses of java.lang.Object. The methods of the two languages Object classes share some similarities (e.g. System.Object's ToString() to java.lang.Object's toString()) and differences (System.Object does not have analogs to wait(), notify() or notifyAll() in java.lang.Object).

In C#, the object class can either be written as object or Object. The lower case "object" is a C# keyword which is replaced with the class name "System.Object" during compilation. From Table 3.2, one can predicate that one of the reasons of software complexity's the absence of some keywords from the languages. From the program view it is easier to programming when the keywords less. Regarding program execution, it will be more complex with longer access time.

Table 3.2: Brief comparison between C# and Java languages keywords

	C# keywords	Java keywords
The same keywords	35	35
The similar in function and different in definition	15	15
Keywords found in C# and not in Java	34	N/A
Keywords found in Java and not in C#	N/A	5
Total Keywords	79	50

3.3 Examples of Measuring Complexity for Software Program

Two case studies of linear search algorithm and binary search algorithm written through object-oriented programming languages such as C++, Java, and Visual Basic were selected to measure the complexity of each languages. The measurements were based on line of program, LOC without comments, LOC+ comments, McCabe method, the program difficulty using Halstead method and file size.

3.3.1 Case Study I (Linear Search Complexity)

Linear search algorithm is used as an example to compare the complexity of a program if written using C++, Visual Basic, and Java programming languages. The listing of the main program (test program) of the linear search was written in C++ is as shown below. The method LinearSearch is not included in the comparison for simplicity. From Deitel's book How to program C++: Figure 3.3 shows the flow chart of the algorithm, Figure 3.4 shows the flow graph of the algorithm.

```
1   // C++
2   // Linear search of an array.
3   #include <iostream>
4   using std::cout;
5   using std::cin;
6   using std::endl;
7
8   int linearSearch( const int [], int, int ); // prototype
9
10  int main()
11  {
12    const int arraySize = 100; // size of array a
13    int a[ arraySize ]; // create array a
14    int searchKey; // value to locate in array a
15
16    for ( int i = 0; i < arraySize; i++ )
17      a[ i ] = 2 * i; // create some data
18
19    cout << "Enter integer search key: ";
20    cin >> searchKey;
21
22    // attempt to locate searchKey in array a
23    int element = linearSearch( a, searchKey, arraySize );
24
25    // display results
26    if ( element != -1 )
27      cout << "Found value in element " << element << endl;
28    else
29      cout << "Value not found" << endl;
30
31    return 0; // indicates successful termination
32  } // end main
```

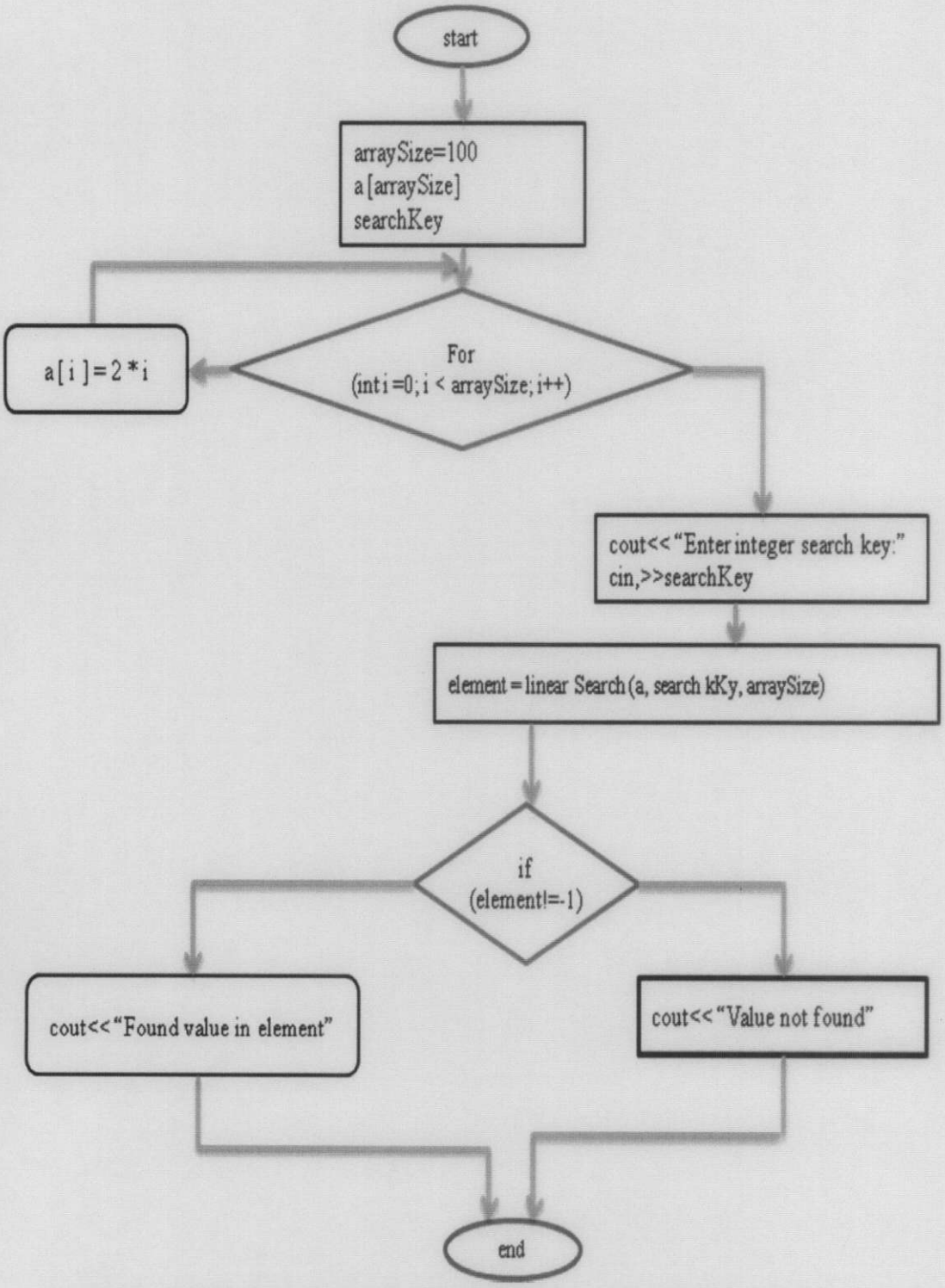


Figure 3.3: Main program- linear search flow chart written in C++ language

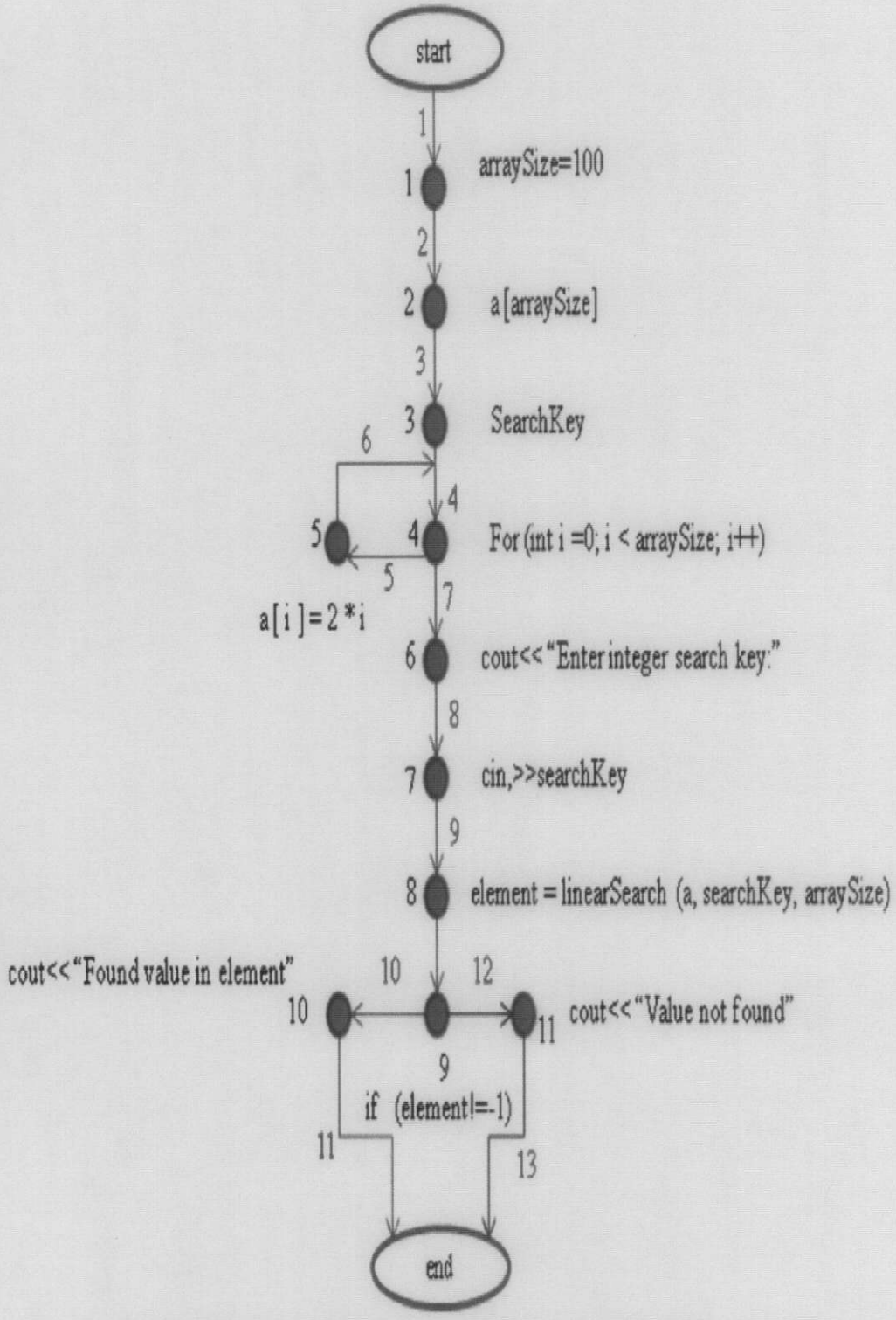


Figure 3.4: Main program- Linear search flow graph written in C++ language

Table 3.3 shows the complexity of the main program of the linear search written in C++ language. The table shows that the length in lines of the program as 32, and LOC without comments as 20. The measured LOC+ comments is 7 where LOC+ comments, is that mixed line of code with comments on the same line. The McCabe complexity is measured as 4 while the program difficulty by Halstead method is 9.3. While the file size of this program is 954 bytes.

Table 3.3: The complexity of the main program for linear search written in C++

Complexity Type	Value
Length (in lines)	32
LOC without comments	20
LOC + comments	7
McCabe complexity	4
The Program difficulty D (As per Halstead method)	9.3
File Size	954 bytes

The list of the main program (test program) of linear search written in Visual Basic is shown below. The method LinearSearch is not used in the comparison for simplicity. Figure 3.5 shows the flow chart of linear search algorithm from Deitel book "Visual Basic 2005 for Programmers". Figure 3.6 shows the flow graph of the algorithm.

```
1     ' Visual Basic
2     ' Linear search of an array.
3     Public Class FrmLinearSearchTest
4       Dim array1 As Integer() = New Integer(19) {}
5
6       ' create random data
7       Private Sub btnCreate_Click(ByVal sender As System.Object, _
8         ByVal e As System.EventArgs) Handles btnCreate.Click
9
10        Dim randomNumber As Random = New Random()
11        Dim output As String = ("Index" & vbTab & "Value" & vbCrLf)
12
13        ' create string containing 20 random numbers
14        For i As Integer = 0 To array1.GetUpperBound(0)
15          array1(i) = randomNumber.Next(1000)
16          output &= (i & vbTab & array1(i) & vbCrLf)
17        Next
18
19        txtData.Text = output ' display numbers
20        txtInput.Text = "" ' clear search key text box
21        btnSearch.Enabled = True ' enable search button
22      End Sub ' btnCreate_Click
23
24      ' search array for search key
```

```
25        Private Sub btnSearch_Click(ByVal sender As System.Object, _
26          ByVal e As System.EventArgs) Handles btnSearch.Click
27
28          ' if search key text box is empty, display
29          ' message and exit method
30          If txtInput.Text = "" Then
31            MessageBox.Show("You must enter a search key.", "Error", _
32              MessageBoxButtons.OK, MessageBoxIcon.Error)
33            Exit Sub
34          End If
35
36          Dim searchKey As Integer = Convert.ToInt32(txtInput.Text)
37          Dim element As Integer = LinearSearch.Search(searchKey, array1)
38
39          If element <> -1 Then
40            lblResult.Text = "Found value in index " & element
41          Else
42            lblResult.Text = "Value not found"
43          End If
44        End Sub ' btnSearch_Click
45      End Class ' FrmLinearSearchTest
```

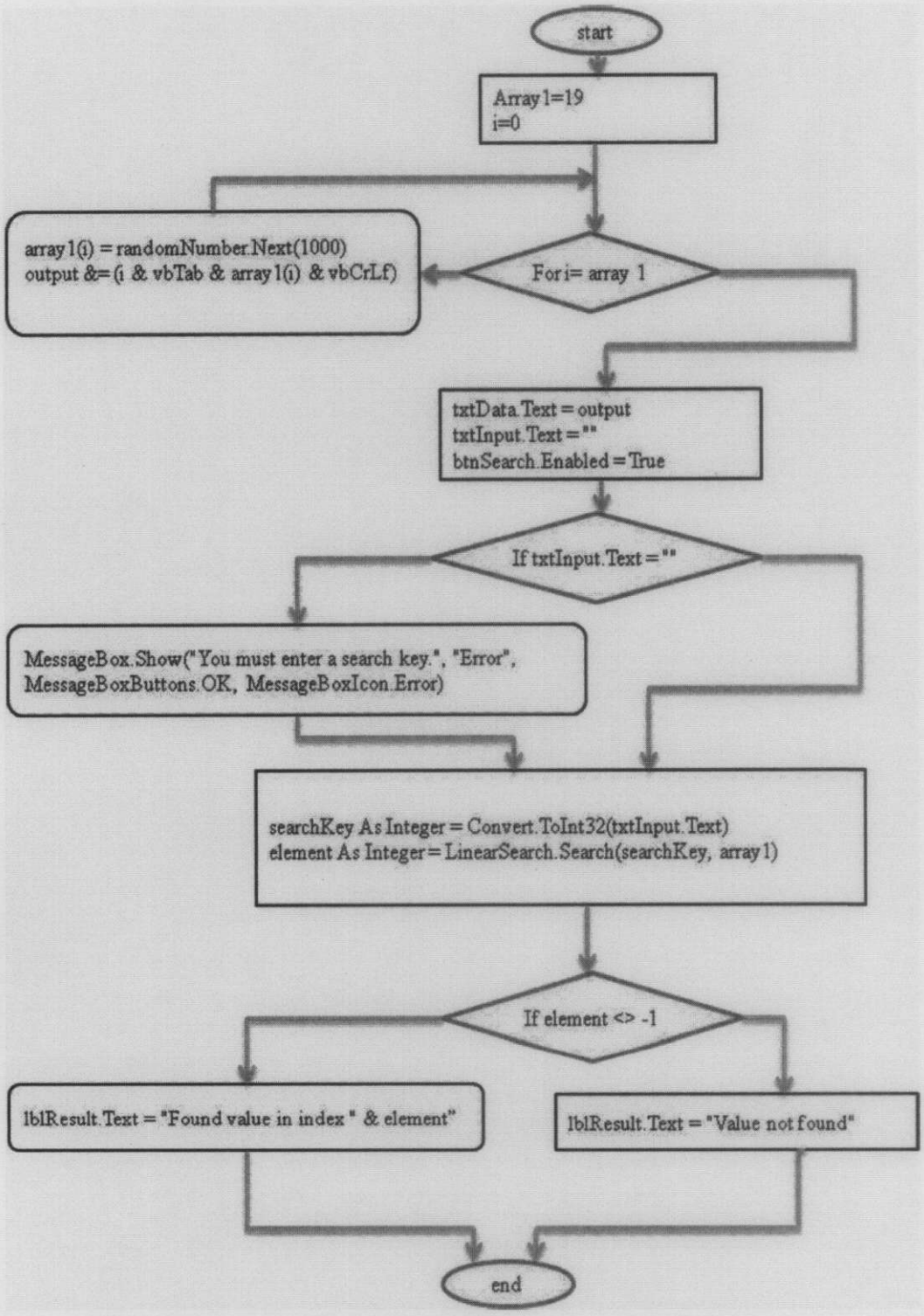


Figure 3.5: Main program- Linear search flow chart written in Visual Basic language

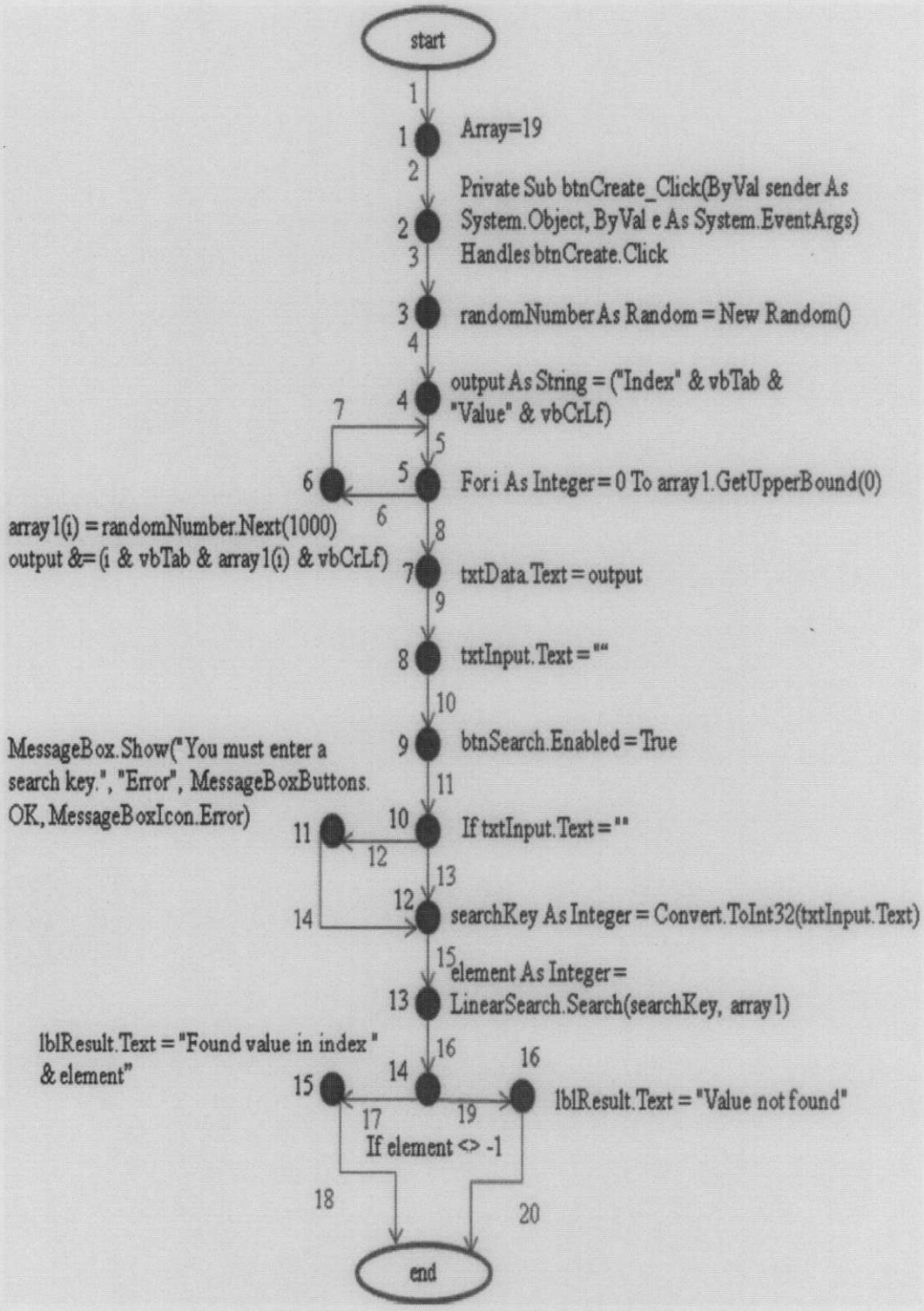


Figure 3.6: Main program- linear search flow graph written in Visual Basic languages

Table 3.4 shows the complexity of the main program linear search written in Visual Basic language. The table shows that the length in lines of the program are 45, LOC without comments are 27, and LOC + comments is 6. The McCabe complexity is 6 while the program difficulty by Halstead method is 11.6. The file size of this program is 1789 bytes.

Table 3.4: The complexity of the main program linear search written in Visual Basic

Complexity Type	Value
Length (in lines)	45
LOC without comments	27
LOC + comments	6
McCabe complexity	6
The Program difficulty D (As per Halstead method)	11.6
Size Memory	1,789 bytes

The list of the main program (test program) of linear search written in Java language is shown below. The method LinearSearch is not used in the comparison for simplicity. Figure 3.7 shows the flow chart of linear search algorithm from Deitel book "How to program Java". Figure 3.8 shows the flow graph of the algorithm.

```
1   // Java
2   // Sequentially Linear search an array for an item.
3   Import java.util.Scanner;
4
5   public class LinearSearchTest
6   {
7   public static void main( String args[] )
8   {
9   // create Scanner object to input data
10   Scanner input = new Scanner( System.in );

11
12   int searchInt; // search
13   int position; // location of search key in array
14
15   // create array and output it
16   LinearArray searchArray = new LinearArray( 10 );
17   System.out.println( searchArray );
18
19   // get input from user
20   System.out.print(
```

```
21     "Please enter an integer value (-1 to quit): " );
22   searchInt = input.nextInt(); // read an int from user
23
24   // repeatedly input an integer; -1 will quit the program
25   while ( searchInt != -1 )
26   {
27     // search array linearly
28     position = searchArray.linearSearch( searchInt );
29
30     // return value of -1 indicates integer was not found
31     if ( position == -1 )
32       System.out.println( "The integer " + searchInt +
33         " was not found.\n" );
34     else
35       System.out.println( "The integer " + searchInt +
36         " was found in position " + position + ".\n" );
37
38     // get input from user
39     System.out.print(
40       "Please enter an integer value (-1 to quit): " );
41     searchInt = input.nextInt();
42   } // end while
43   } // end main
44   } // end class LinearSearchTest
```

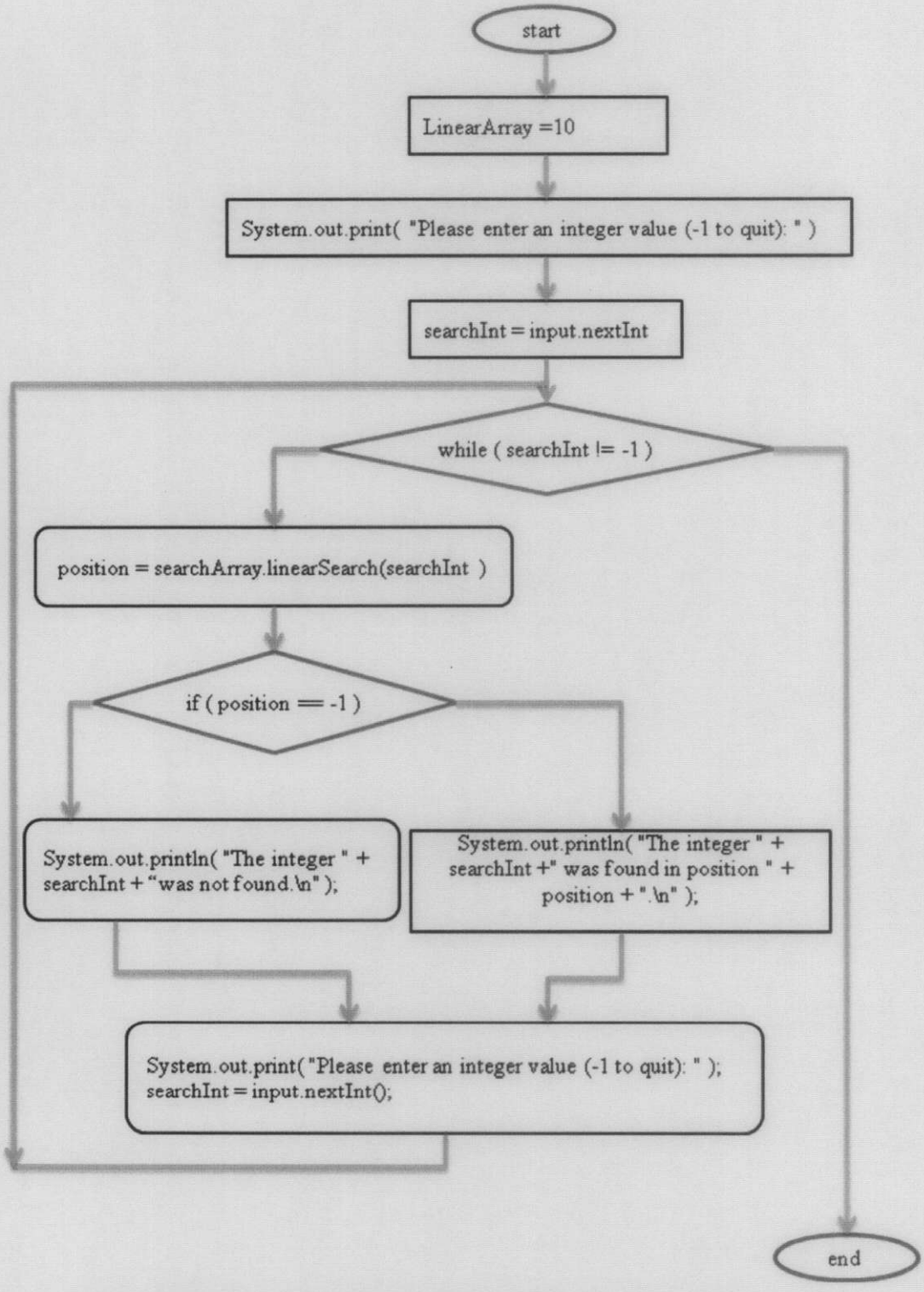
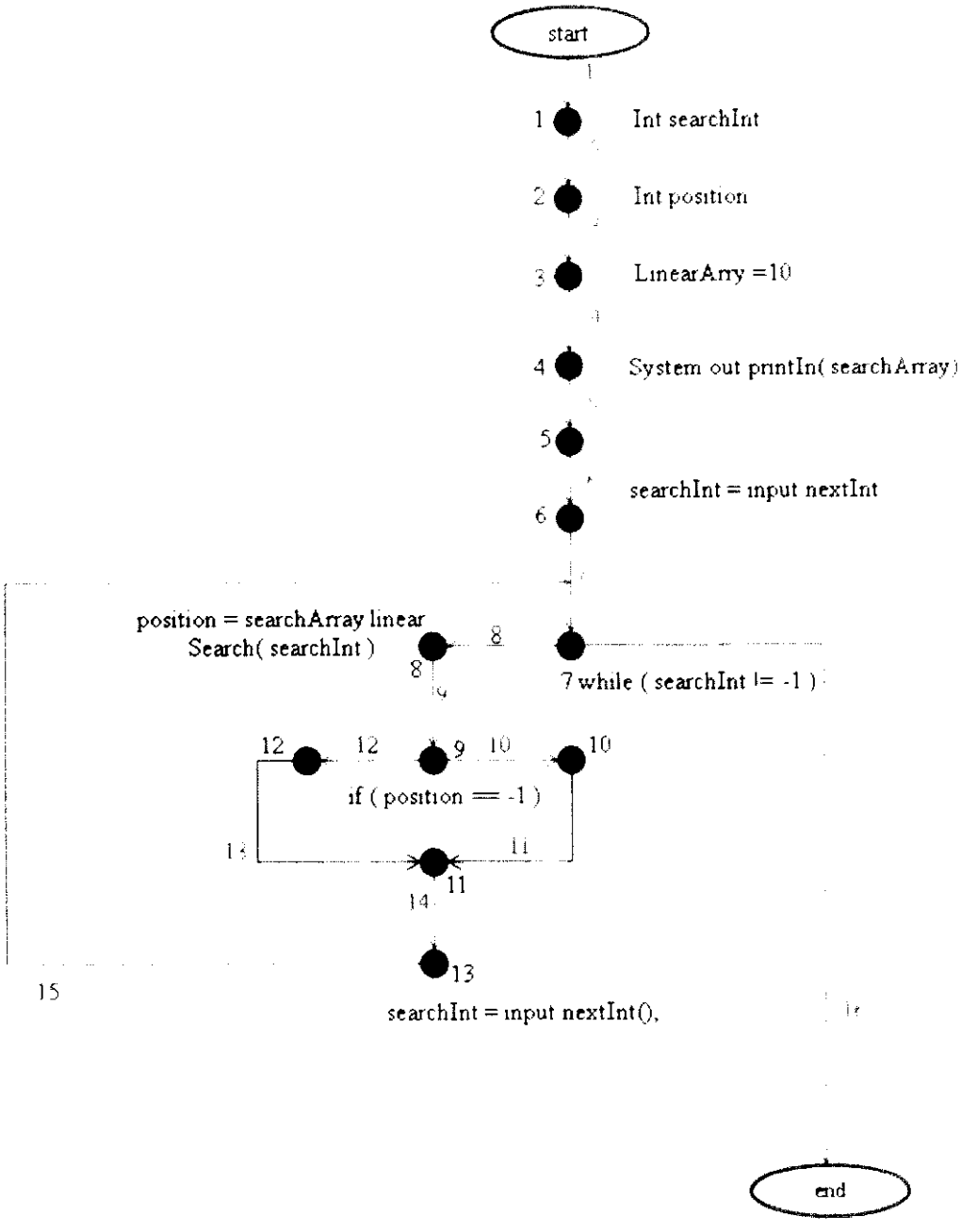


Figure 3.7: Main program- linear search flow chart written in Java language



5 System out print("Please enter an integer value (-1 to quit) ")

10 System Out Print In("The integer " + search Int +" was found in position " + position + " \n").

11 System out print("Please enter an integer value (-1 to quit) ").

12 System out println("The integer " + searchInt + "was not found \n").

Figure 3.8: Main program- linear search flow graph written in Java languages

Table 3.5 shows the complexity of the main program linear search written in Java language. The table shows that the length in lines of the program as 44, LOC without comments as 21, and LOC + comments as 6. The McCabe complexity is measured as 7 while the program difficulty as per Halstead method is 5.8. While the size memory of this program is 1524 bytes.

Table 3.5: The complexity of the main program linear search written in Java

Complexity Type	Value
Length (in lines)	44
LOC without comments	21
LOC + comments	6
McCabe complexity	7
The Program difficulty D (As per Halstead method)	5.8
File size	1,524 bytes

Figure 3.9 shows the comparison between the object oriented language C++, Visual Basic, and Java by using the main program of the linear search algorithm as a case study for comparison.

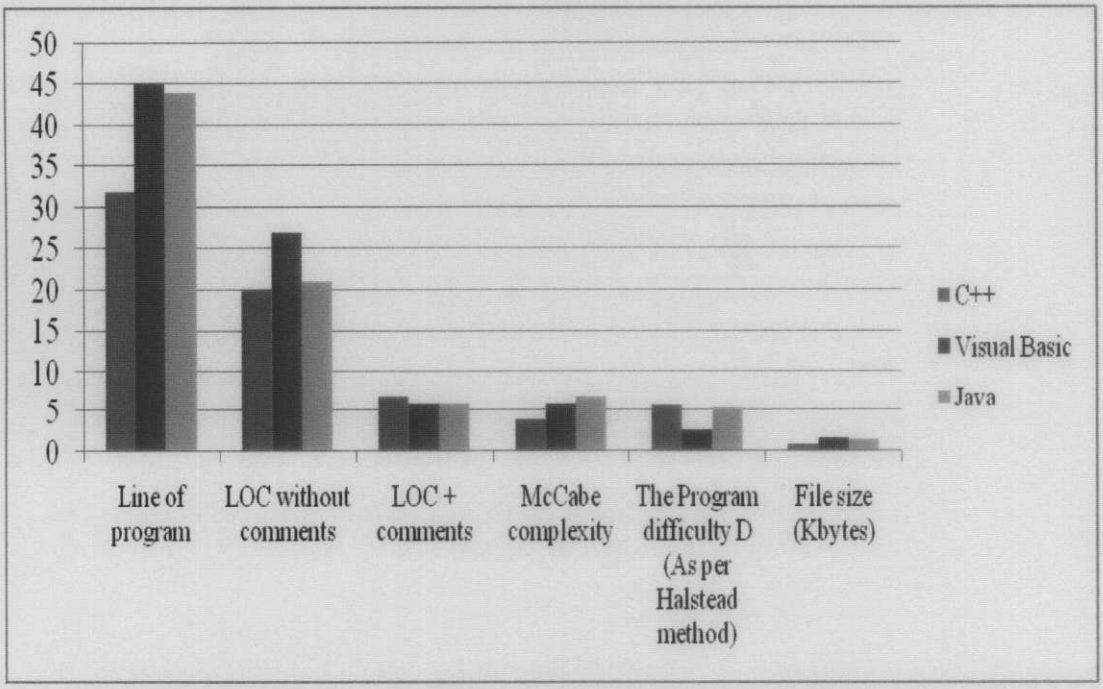


Figure 3.9: Comparison the complexity between the object oriented language C++, Visual Basic, and Java for linear search algorithm

3.3.2 Case Study II (Binary Search Complexity)

The binary search is the second case study taken to measure the complexity written in C++, Visual Basic, and Java languages. The listing of the main program (test program) of the binary search algorithm was written in C++ is shown below while the method binarySearch is not used in the comparison for simplicity. Figure 3.10 shows the flow chart of the algorithm from Deitel book "How to program C++". Figure 3.11 shows the flow graph of the algorithm.

```
1       // C++
2       // BinarySearch test program.
3       #include <iostream>
4       using std::cin;
5       using std::cout;
6       using std::endl;
7
8       #include "BinarySearch.h" // class BinarySearch definition
9
10      int main()

11      {
12        int searchInt; // search key
13        int position; // location of search key in vector
14
15        // create vector and output it
16        BinarySearch searchVector ( 15 );
17        searchVector.displayElements();
18
19        // get input from user
20        cout << "\nPlease enter an integer value (-1 to quit): ";
21        cin >> searchInt; // read an int from user
22        cout << endl;
23
24        // repeatedly input an integer; -1 terminates the program
25        while ( searchInt != -1 )
26        {
27          // use binary search to try to find integer
28          position = searchVector.binarySearch( searchInt );
29
30          // return value of -1 indicates integer was not found
31          if ( position == -1 )
32            cout << "The integer " << searchInt << " was not found.\n";
33          else
34            cout << "The integer " << searchInt
```

```
35              << " was found in position " << position << ".\n";
36
37          // get input from user
38          cout << "\n\nPlease enter an integer value (-1 to quit): ";
39          cin >> searchInt; // read an int from user
40          cout << endl;
41       } // end while
42
43       return 0;
44    } // end main
```

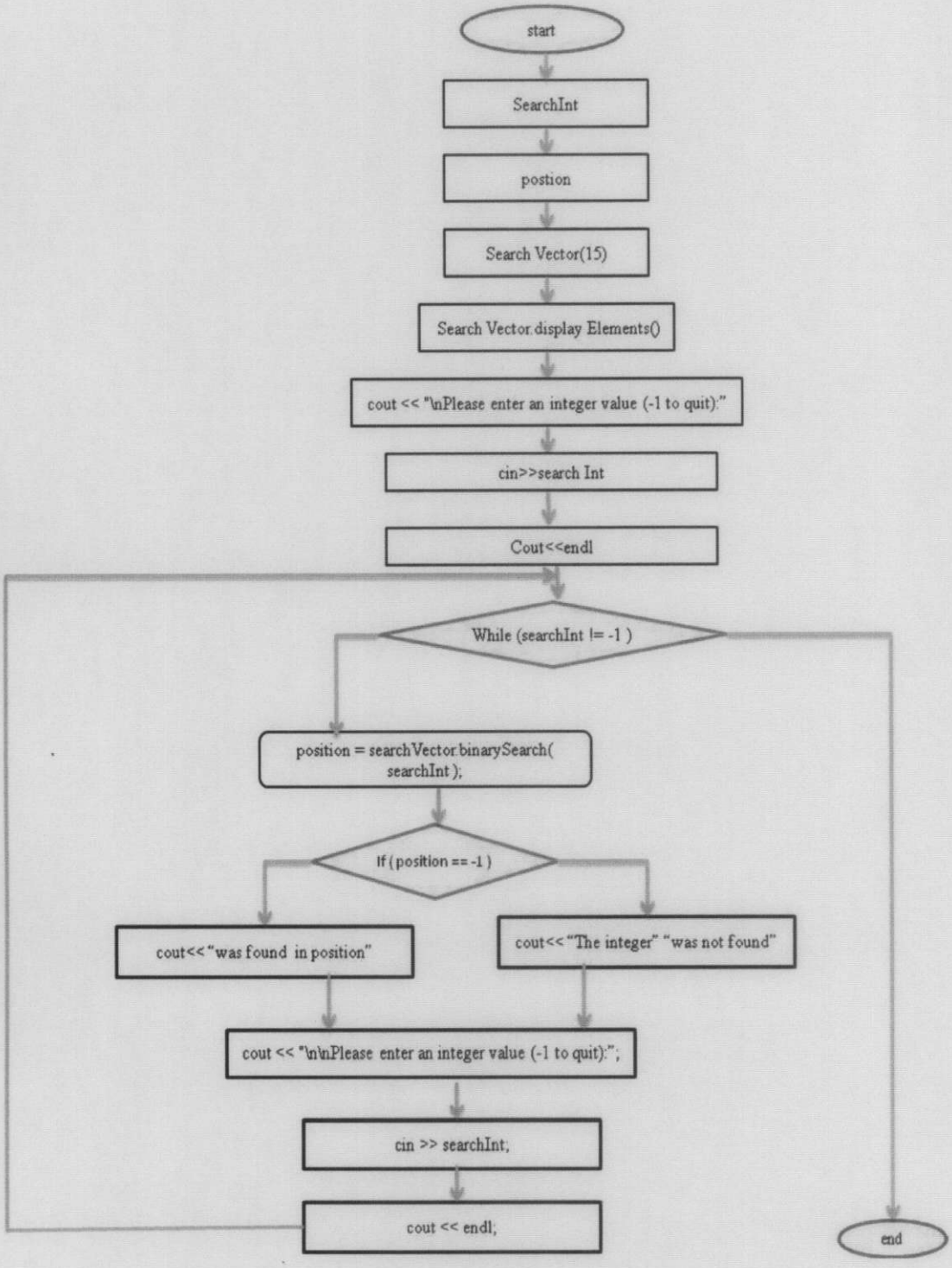
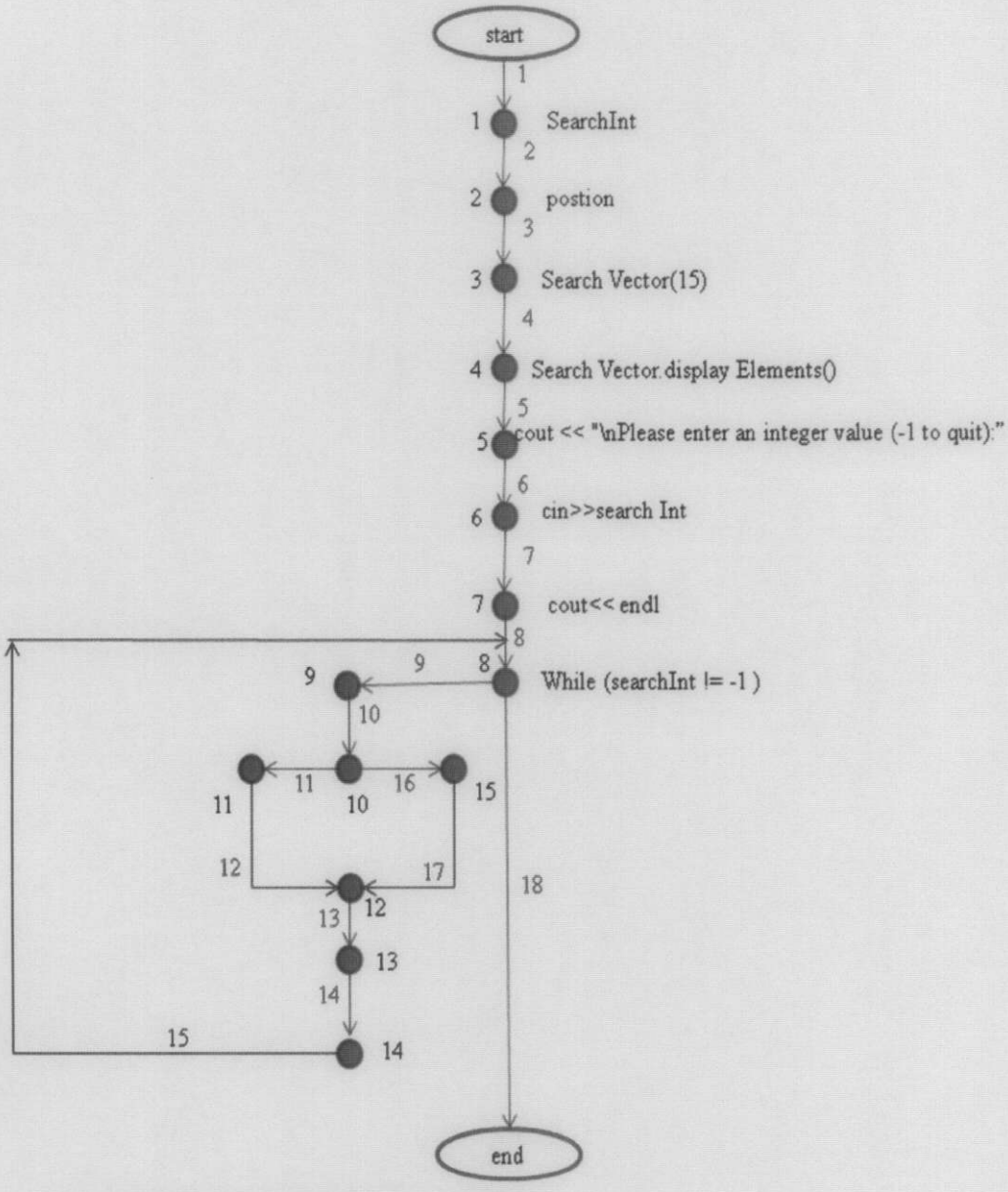


Figure 3.10: Main program- binary search flow chart in C++ language




```
9 position = search Vector.binarySearch( searchInt );
10 If ( position == -1 )
11 cout<< "was found in position"
12 cout << "\n\nPlease enter an integer value (-1 to quit):";
13 cin >> searchInt;
14 cout << endl;
15 cout<< "The integer" "was not found"
```

Figure 3.11: Main program- binary search flow graph written in C++ languages

Table 3.6 shows the complexity of the main program binary search written in C++ language. The table shows that the length in lines of the program is 44, LOC without comments as 25, and LOC + comments as 7. The McCabe complexity is measured as 7 while the program difficulty by Halstead method is 5.1. While the file size of this program is 1393 bytes.

Table 3.6: The complexity of the main program binary search written in C++

Complexity Type	Value
Length (in lines)	44
LOC without comments	25
LOC + comments	7
McCabe complexity	7
The Program difficulty D (As per Halstead method)	5.1
Size Memory	1,393 bytes

The list of the main program (test program) of binary search written in Visual Basic is shown below. The method BinarySearch is not used in the comparison for simplicity. Figure 3.12 shows the flow chart of binary search algorithm written in Visual Basic language from Deitel book "Visual Basic 2005 for programmers". Figure 3.13 shows the flow graph of the algorithm.

```
1     ' Visual Basic
2     ' Binary search of an array using Array.BinarySearch.
3     Imports System
4
5     Public Class FrmBinarySearchTest
6       Dim array1 As Integer() = New Integer(19) {}
7
8       ' create random data
9       Private Sub btnCreate_Click(ByVal sender As System.Object, _
10        ByVal e As System.EventArgs) Handles btnCreate.Click
11
12        Dim randomNumber As Random = New Random()
13        Dim output As String = ("Index" & vbTab & "Value" & vbCrLf)
14
15        ' create random array elements
16        For i As Integer = 0 To array1.GetUpperBound(0)
17          array1(i) = randomNumber.Next(1000)
```

```
18          Next
19
20          Array.Sort(array1) ' sort array to enable binary searching
21
22          ' display sorted array elements
23          For i As Integer = 0 To array1.GetUpperBound(0)
24            output &= (i & vbTab & array1(i) & vbCrLf)
25          Next
27          txtData.Text = output ' displays numbers
28          txtInput.Text = "" ' clear search key text box
29          btnSearch.Enabled = True ' enable search button
30        End Sub ' btnCreate_Click
31
32        ' search array for search key
33        Private Sub btnSearch_Click(ByVal sender As System.Object, _
34          ByVal e As System.EventArgs) Handles btnSearch.Click
35
36          ' if search key text box is empty, display
37          ' message and exit method
38          If txtInput.Text = "" Then
39            MessageBox.Show("You must enter a search key.", "Error", _
40              MessageBoxButtons.OK, MessageBoxIcon.Error)
41            Exit Sub
42          End If
43
44          Dim searchKey As Integer = Convert.ToInt32(txtInput.Text)
45          Dim element As Integer = Array.BinarySearch(array1, searchKey)
46
47          If element >= 0 Then
48            lblResult.Text = "Found Value in index " & element
49          Else
50            lblResult.Text = "Value Not Found"
51          End If
52        End Sub ' btnSearch_Click
53      End Class ' FrmBinarySearchTest
```

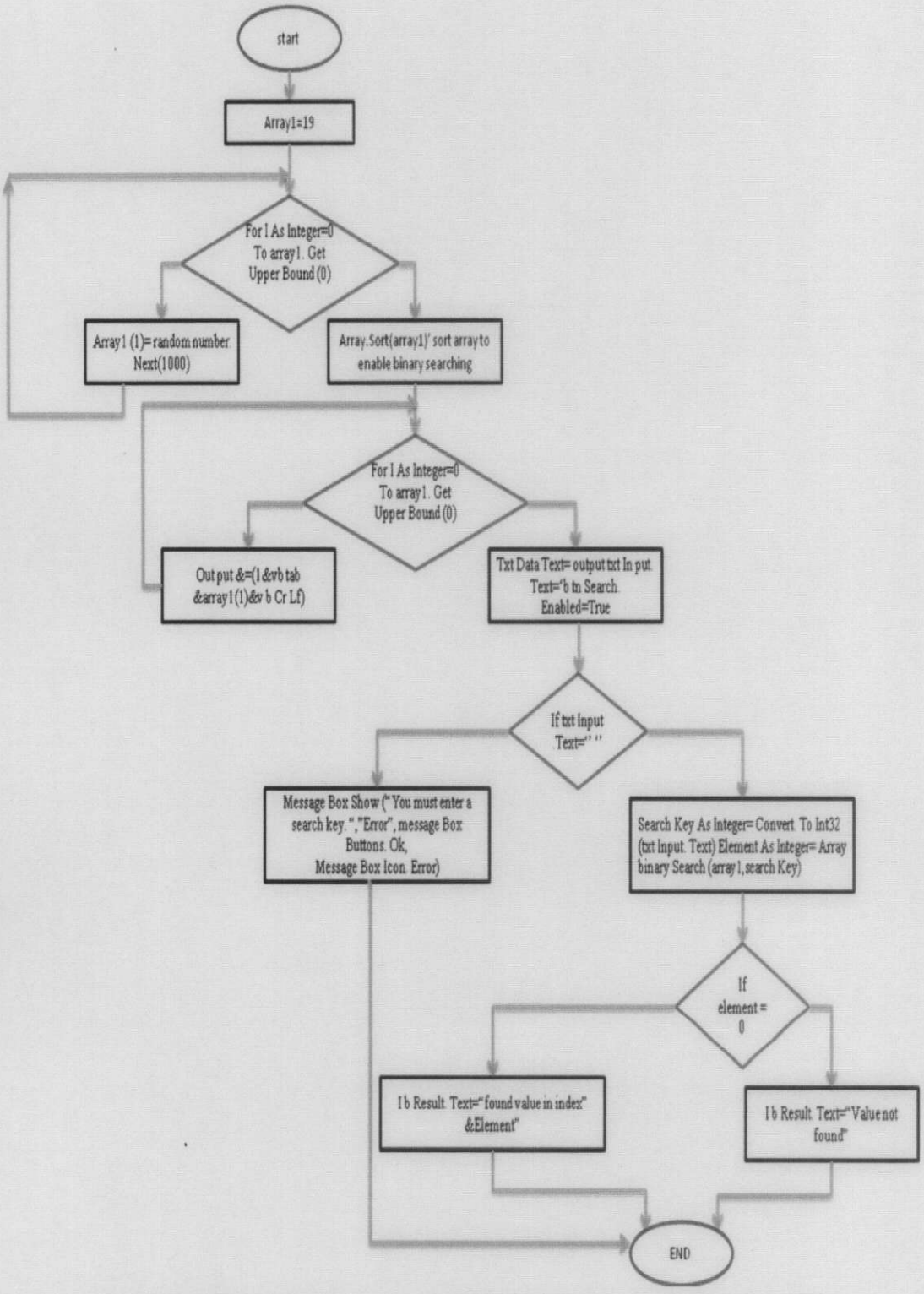
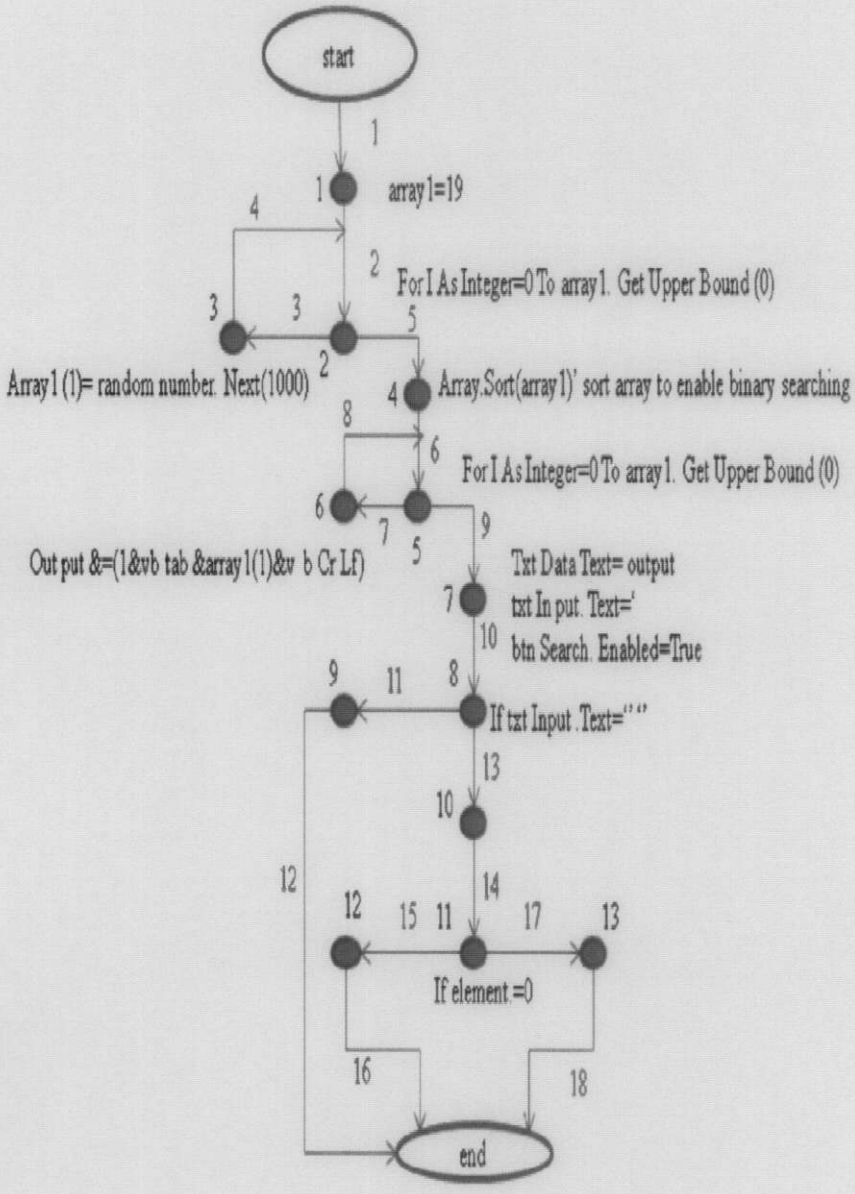


Figure 3.12: Flow chart for binary search code written by Visual Basic language



9 Message Box Show ("You must enter a search key. ","Error", message Box Buttons. Ok, Message Box Icon. Error)
10 Search Key As Integer= Convert. To Int32 (txt Input. Text) Element As Integer= Array binary Search (array1,search Key)
12 I b Result. Text="found value in index" &Element"
13 I b Result. Text="Value not found"

Figure 3.13: Main program- binary search flow graph written in Visual Basic languages

Table 3.7 shows the complexity of the main program of the binary search written in Visual Basic language. The table shows that the length in lines of the program with comments as 53, LOC without comments as 31, and LOC+ comments as 5. The McCabe complexity is measured as 9 while the program difficulty as per Halstead method is 10.1. The file size of this program is 2026 bytes.

Table 3.7: The complexity of the main program binary search written in Visual Basic

Complexity Type	Value
Length (in lines)	53
LOC without comments	31
LOC + comments	5
McCabe complexity	9
The Program difficulty D (As per Halstead method)	10.1
Size Memory	2,026 bytes

The list of the main program (test program) of binary search written in Java is shown below. The method BinarySearch is not used in the comparison for simplicity. Figure 3.14 shows the flow chart of binary search algorithm written in Java language from Deitel book "How to program Java". Figure 3.15 shows the flow graph of the algorithm.

```
1  // Java
2  // Sequentially Binary search an array for an item.
3  import java.util.Scanner;
4
5  public class BinarySearchTest
6  {
7  public static void main( String args[] )
8  {
9     // create Scanner object to input data
10    Scanner input = new Scanner( System.in );
11
12     int searchInt; // search
13     int position; // location of search key in array
14
15     // create array and output it
16     BinaryArray searchArray = new BinaryArray( 16 );
17     System.out.println( searchArray );
18
19     // get input from user
20     System.out.print(
```

```
21       "Please enter an integer value (-1 to quit): " );
22     searchInt = input.nextInt(); // read an int from user
23
24     // repeatedly input an integer; -1 will quit the program
25     while ( searchInt != -1 )
26     {
27       // use binary search to try to find integer
28       position = searchArray.binarySearch( searchInt );
29
30       // return value of -1 indicates integer was not found
31       if ( position == -1 )
32         System.out.println( "The integer " + searchInt +
33           " was not found.\n" );
34       else
35         System.out.println( "The integer " + searchInt +
36           " was found in position " + position + ".\n" );
37
38       // get input from user
39       System.out.print(
40         "Please enter an integer value (-1 to quit): " );
41       searchInt = input.nextInt();
42     } // end while
43   } // end main
44   } // end class BinarySearchTest
```

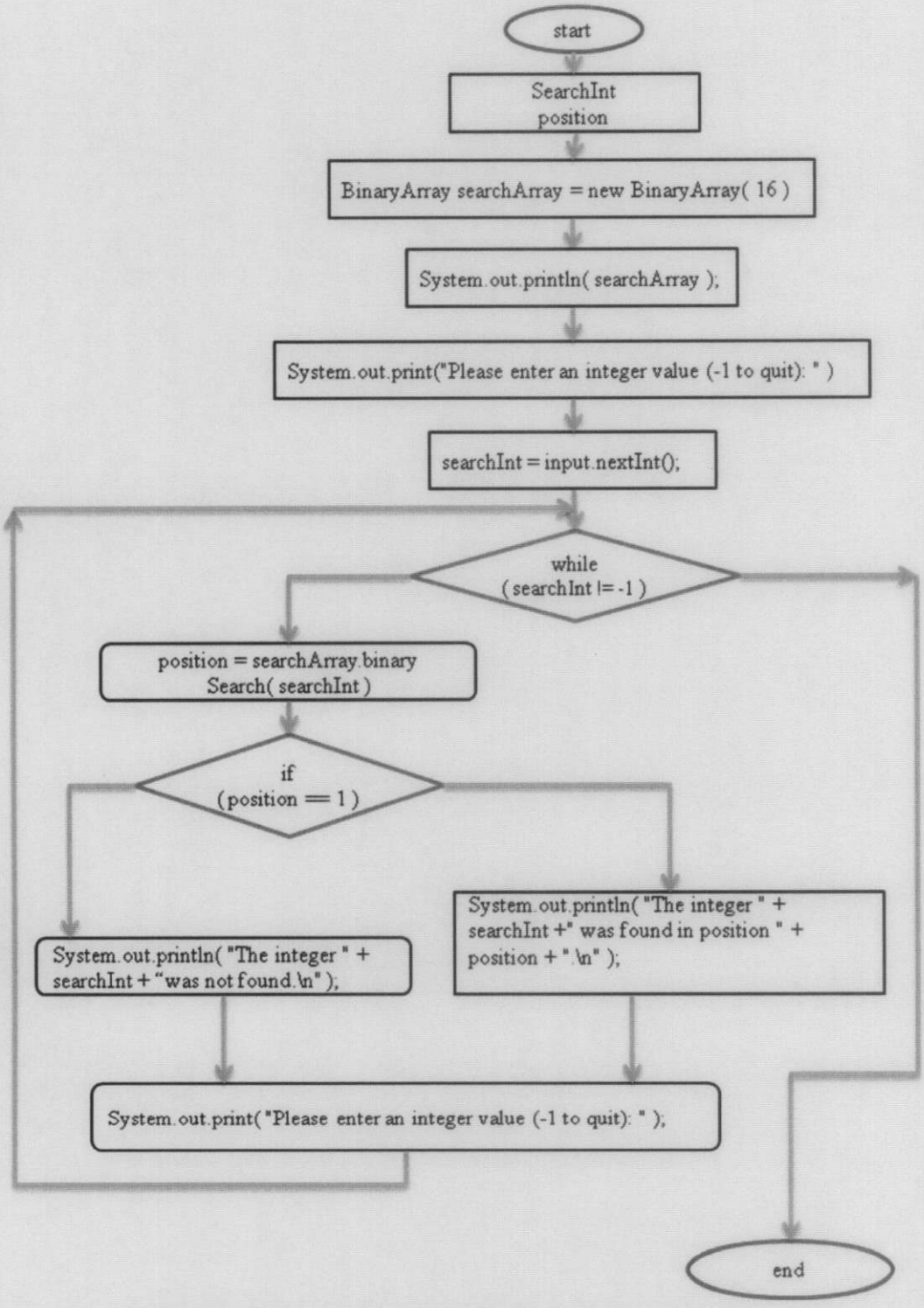


Figure 3.14: Main program- binary search flow chart written by Java language

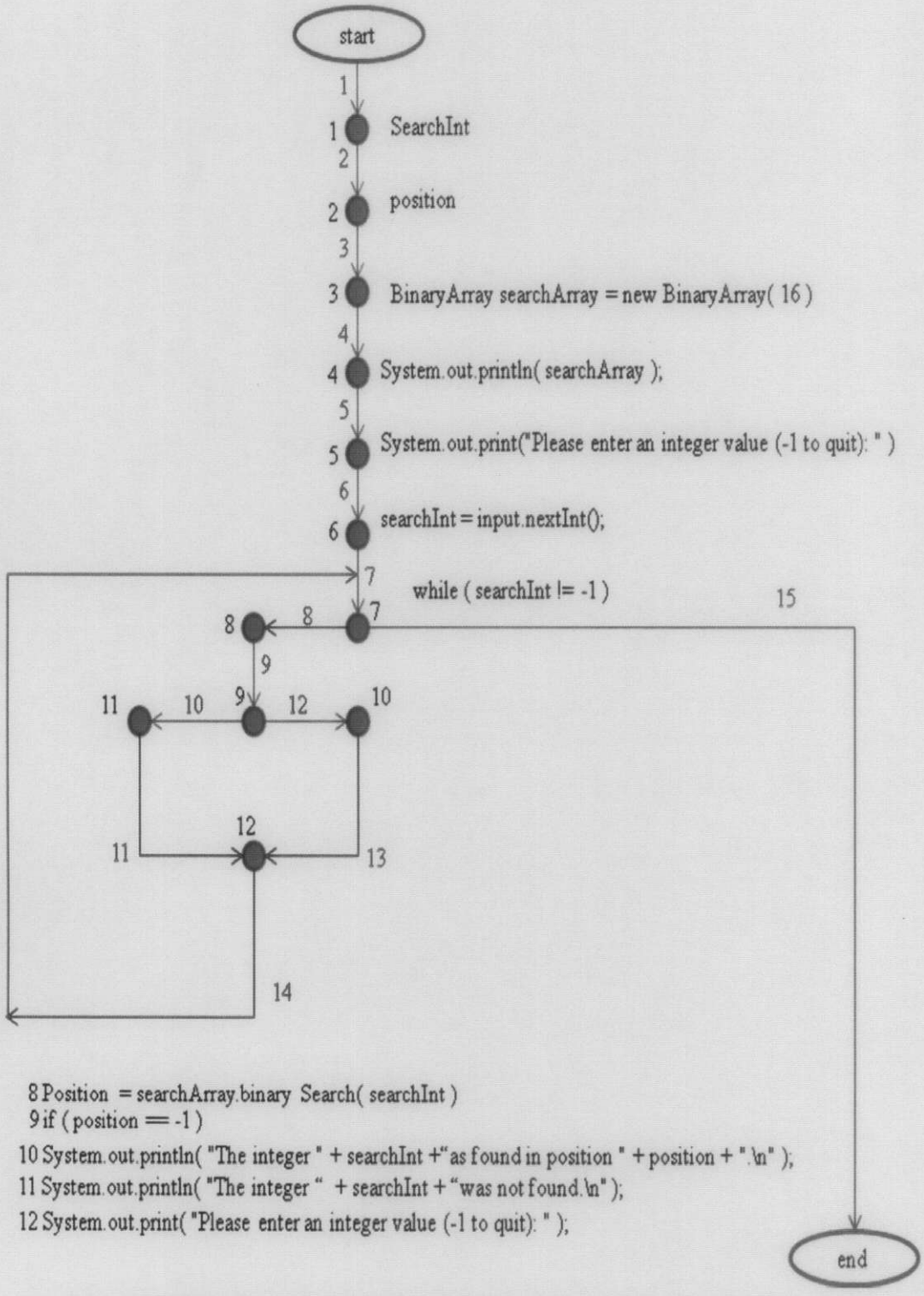


Figure 3.15: Main program- binary search flow graph written in Java languages

Table 3.8 shows the complexity of the main program of the linear search written in Java language. The table shows that the length in lines of the program as 44, LOC without comments as 24, and LOC + comments as 6. The McCabe complexity is measured as 7 while the program difficulty as per Halstead method is 6.4. The token measured for the same example and found 133 whereas the size memory of this program is 1572 bytes.

Table 3.8: The complexity of the main program binary search written by Java

Complexity Type	Value
Length (in lines)	44
LOC without comments	24
LOC + comments	6
McCabe complexity	7
The Program difficulty D (As per Halstead method)	6.4
Size Memory	1,572 bytes

Figure 3.16 shows the comparison between the object oriented language C++, Visual Basic, and Java, by using the main program of the binary search algorithm as a case study for comparison.

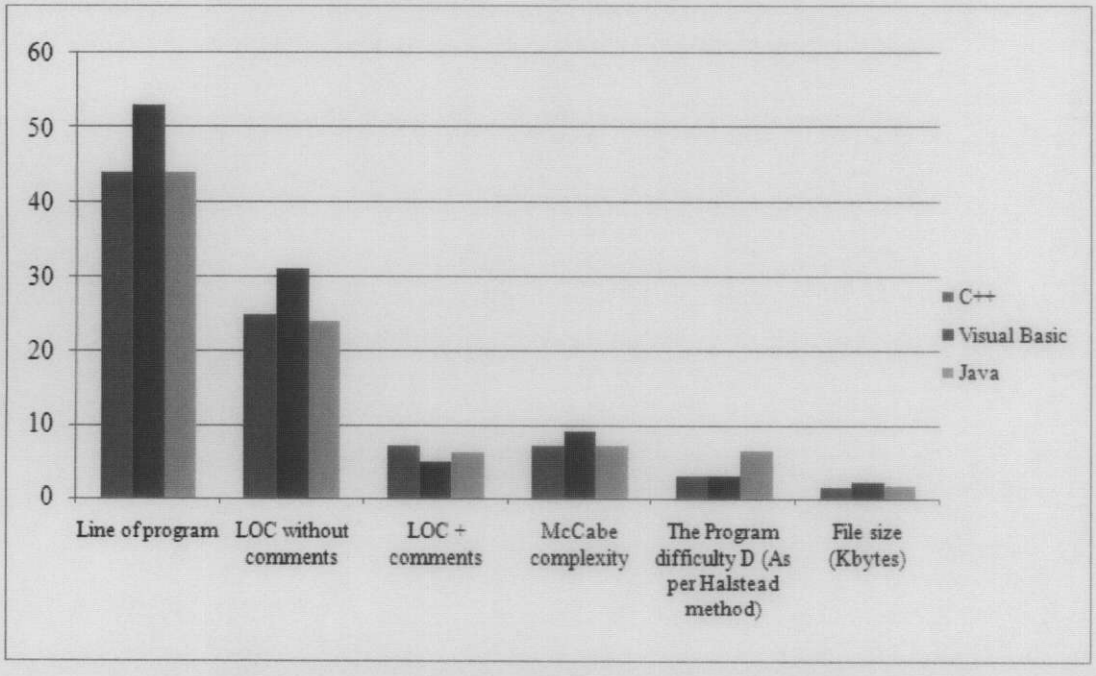


Figure 3.16: Comparison the complexity between the object oriented language C++, Visual Basic, and Java for binary search algorithm

In Table 3.9 shows the comparison between McCabe and Halstead methods for Linear Search of an array taken in consideration three type of languages C++, Visual Basic and Java.

Table 3.9: Comparison of McCabe vs. Halstead methods for linear search of an array

	C++	Visual Basic	Java
McCabe Method	**4**	**6**	**7**
Halstead's Method	**9.3**	**11.6**	**5.8**

In Table 3.10 there are comparison between McCabe and Halstead methods for Binary Search of an Array taken in consideration three type of languages C++, Visual Basic and Java.

Table 3.10: Comparison of McCabe vs. Halstead methods for Binary search of an array

	C++	Visual Basic	Java
McCabe Method	**7**	**9**	**7**
Halstead's Method	**5.1**	**10.1**	**6.4**

The below Figures 3.17 and 3.18 are showing the differences between McCabe and Halstead measurement for Linear and Binary Search of an array for the three languages.

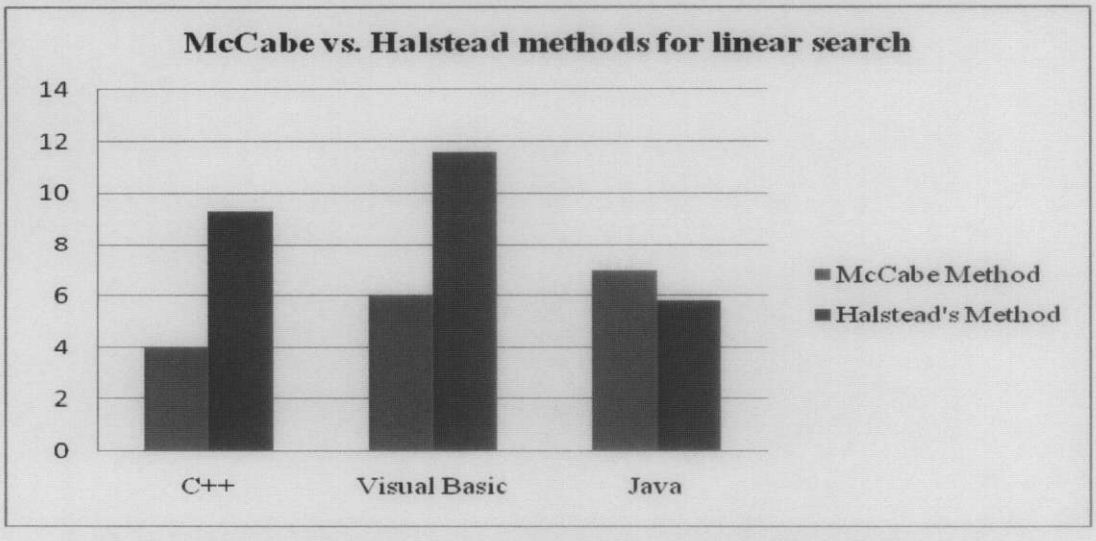


Figure 3.17: Comparison of McCabe vs. Halstead for linear search of an array

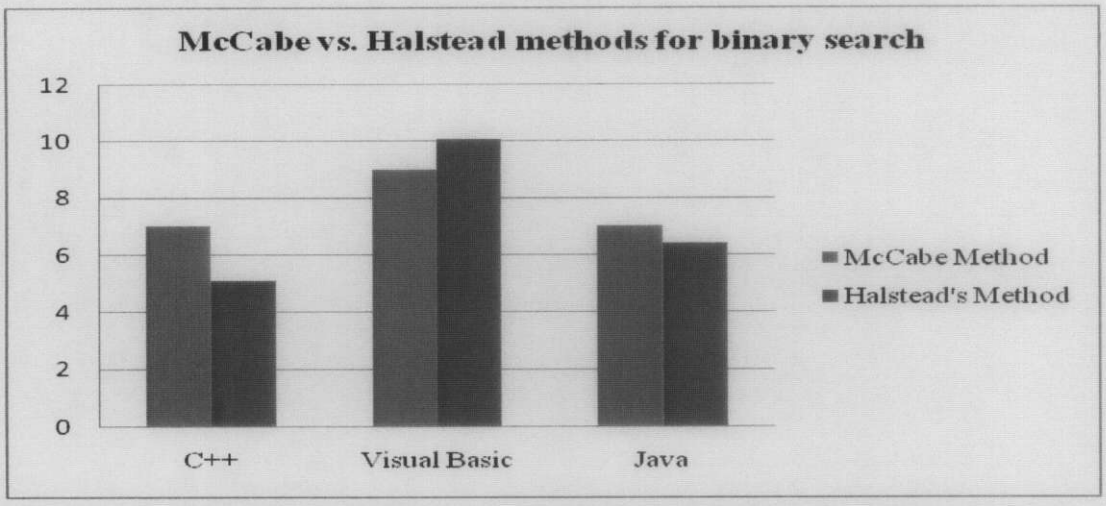


Figure 3.18: Comparison of McCabe vs. Halstead methods for binary search of an array

3.4 Conclusion

In this chapter, a brief description about object oriented programming and the development history of C++, Visual Basic, and Java languages is given. Two case studies to measure the complexity of linear search algorithm and binary search algorithm are discussed. These algorithm written by three most popular object oriented languages C++, Visual Basic, and Java. The complexity to each of these examples is measured based on length in lines of program, line of code (LOC) without comments, LOC + comments (LOCC), McCabe method, the program difficulty using Halstead method and size memory. It is found that McCabe method has various measuring value of complexity for C++, Visual Basic, and Java languages for linear search and same measuring value for C++ and Java languages for binary search as it measured and found the measured value for C++ and Java languages as 7. Though the measured complexity with McCabe method is higher for Visual Basic languages with value equal to 9 in binary search. Using Halstead method implies various measuring values of complexity regarding using different programming languages either in linear or binary search cases. Other words, the measured values of complexity are different for either linear or binary search from one language to another language means that if one program is written in C++ languages then its measured complexity will be different with other type of languages such as Visual Basic and Java.

العنوان:	Measuring software complexity for software engineering quality assurance
المؤلف الرئيسي:	Wohaishi, Ahmad Muhammad Ali
مؤلفين آخرين:	Fahmy, Maged A.، Al Sultanny, Yas A.(Co-Advisor، Advisor)
التاريخ الميلادي:	2009
موقع:	المنامة
الصفحات:	1 - 155
رقم MD:	736039
نوع المحتوى:	رسائل جامعية
اللغة:	English
الدرجة العلمية:	رسالة ماجستير
الجامعة:	جامعة الخليج العربي
الكلية:	كلية الدراسات العليا
الدولة:	البحرين
قواعد المعلومات:	Dissertations
مواضيع:	برامج الحاسوب، هندسة البرمجيات، تكنولوجيا المعلومات
رابط:	https://search.mandumah.com/Record/736039

Chapter 4
Modification of Halstead Mathematical Equation Complexity

Software complexity measurement by using Halstead method will be discussed in this chapter. Halstead method relies on counting the number of operator through a program. A brief discussion for add, multiplication, and division logic design is given to recognize the function of each operator in terms of levels of abstraction, architectural, logical, and geometrical. A modification to Halstead method is suggested based on the fact that mathematical operator have different complexities. It is noted that when using Halstead method for measuring software complexity that it only counts the number of operators within LOC of a program regardless of the type of these operators. This means that the cost of applying multiplication operator for example is the same of that of applying addition multiplication. A big question is raised at that point. Multiplication operator is really achieved using as repetitive addition in computers. Also, subtraction is achieved using twos complement addition. Division is also achieved using sequential subtraction. Operators should be given different weights regarding their types when measuring complexity. This means that if a weight is given to addition operator then multiplication operator weight should be given multiples of that of addition weight. The same is valid regarding division operator; it should be given a weight which is a multiple of that of subtraction operator weight. To clarify this, a discussion of combinatorial logic design for performing mathematical operators inside computers is discussed through the following Sections. Hardware circuits for performing these different mathematical operators are discussed. One can notice that their hardware complexities are different from one operator to other. This reflects the idea and strengthens the idea or the criticism raised by the researcher in this thesis that every mathematical operator should be given a unique complexity weight rather than only counting number of operators as that of Halstead. Also, processor type has a great impact on calculating program complexity. The reason is that execution time for each mathematical operator differs from a processor type to another as it is discussed and

clarified in the following Sections. The effect of execution time on adds, multiplication, and division processes using three different generations of personal computer processors namely: 80286, 80486 and Pentium are considered and discussed and used in modifying Halstead method.

4.1 Combinational Logic Design

Circuit models can be classified in terms of levels of abstraction, architectural, logical, and geometrical. Logic level model deals with all facets of combinational and sequential circuits. Logic synthesis can be defined as the manipulation of functional specifications to a model as an interconnection of primitives components. In other words, logic synthesis determines the gate level structure of circuits. The classical logic synthesis algorithms include the optimization of two quality measures, namely: area and performance (Sarif B. 2003).

4.1.1 Binary Adders

One of the most important tasks performed by a digital computer is the operation of adding two binary numbers. A useful measure of performance is speed. Of course, speed can be improved by using gate designs that favor speed at the expense of other measures, such as power consumption. But for the *logic* designer, the important question is how to design an adder to increase the speed, regardless of the type of gate used. It may be that increasing the speed can be achieved at the expense of increasing circuit complexity. That is, there might be several designs, each characterized by a certain speed and a certain circuit complexity (Sarif B. 2003).

A judgment must be made as to the acceptable trade-offs between them. A symbolic diagram representing a binary adder is shown in Figure 4.1. Each open arrowhead represents multiple variables; in this case the inputs are two binary numbers. If each number has n digits, then each line shown really represents n lines. The sum of two n-bit numbers is an $(n + 1)$-bit number. Thus, S (sum) represents $(n + 1)$ output lines. If this circuit were designed by the methods, we would require a circuit with $(n + 1)$ output functions, each one dependent on $2n$ variables.

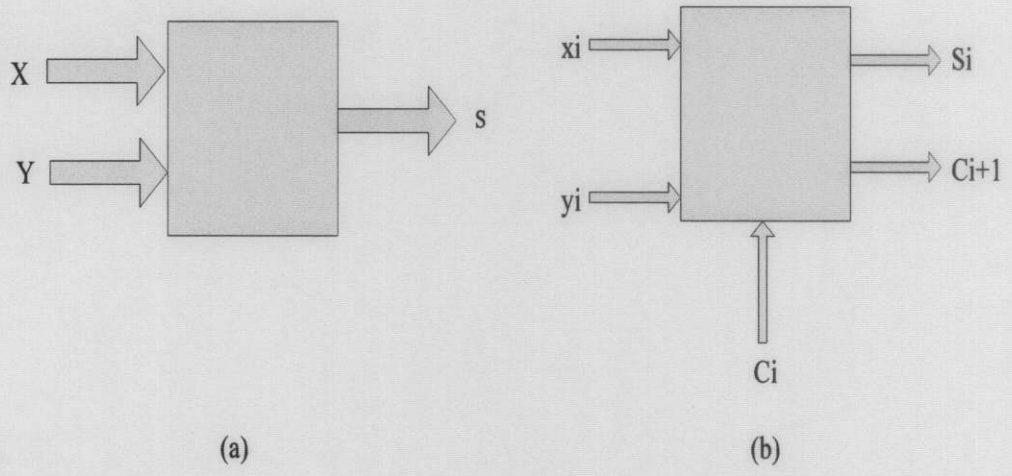


Figure 4.1: Binary addition. (*a*) General adder. (*b*) Full adder of two 1-bit words.

4.1.2 Full Adder

An alternative approach for the addition of two *n*-bit numbers is to use a separate circuit for each corresponding pair of bits. Such a circuit would accept the 2 bits to be added, together with the carry resulting from adding the less significant bits. It would yield as outputs the 1-bit sum and the 1-bit carry out to the more significant bit. Such a circuit is called a *full adder*. A schematic diagram is shown in Figure 4.1. The 2 bits to be added are x_i and y_i, and the *carry in* is C_i. The outputs are the *sum* S_i and the *carry out* C_{i+1}. The truth table for the full adder and the logic maps for the two outputs are shown in Figure 4.2. The minimal sum-of-products expressions for the two outputs obtained from the maps are (Sarif B. 2003);

$$S_i = x_i y_i C_{i'} + x_i y_{i'} C_{i'} + x_{i'} y_{i'} C_i + x_i y_i C_i \quad \ldots(4.1)$$

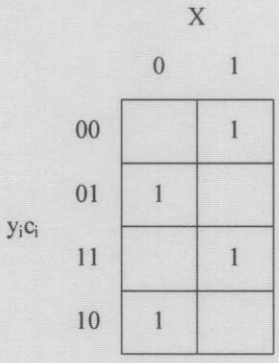
$$C_{i+1} = x_i y_i + x_i C_i + y_i C_i = x_i y_i + C_i(x_i + y_i) \quad \ldots(4.2)$$

Each minterm in the map of S_i constitutes a prime implicant. Hence, a sum-of-products expression will require four 3-input AND gates and a 4-input OR gate. The carry out will

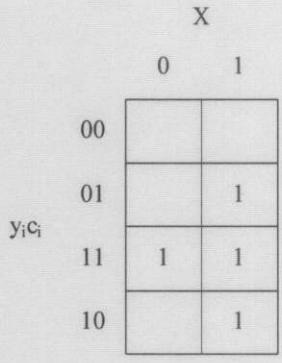
require three AND gates and an OR gate. Assume that each gate has the same propagation delay t_p, then a two-level implementation will have a propagation delay of $2tp$.

C_i	X_i	Y_i	S_i	C_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(a)



(b)



(c)

Figure 4.2: Truth table and logical maps of the full adder. (*a*) Truth table.

(*b*) S_i map. (*c*) C_{i+1} map.

In the map of the carry out, minterm m_7 is covered by each of the three prime implicants. This is overkill; since m_7 is covered by prime implicant x_iy_i, there is no need to cover it again by using it to form prime implicants with m_5 and m_6. If there is some benefit to it, we might use the latter two minterms as implicants without forming prime implicants with m_7. The resulting expression for C_{i+1} becomes

$$C_{i+1} = x_iy_i + C_i(x_iy_i + x_iy_{i'}) = x_iy_i + C_i(x_i + y_i) \qquad \ldots(4.3)$$

We already have an expression for S_i in 4.1, but it is in canonic sum-of-products form. It would be useful to seek an alternative form for a more useful implementation.

With the use of switching algebra, the expression for the sum can be converted to

$$S_i = x_i + y_i + C_i \qquad \ldots(4.4)$$

Using the expressions for S_i and C_{i+1} containing XORs, one can obtain the implementation of the full adder shown in Figure 4.3. Notice that the circuit consists of two identical XOR-AND combinations and an additional OR gate. The circuit inside each dashed box is shown in Figure 4.3; it is named a *half adder*. Its only inputs are the 2 bits to be added, without a carry in. The two outputs are (1) the sum of the 2 bits and (2) the carry out. Assuming that an XOR gate (implemented in a two-level circuit) has a propagation delay of $2t_p$, the full adder in Figure 4.3 has a propagation delay of $4t_p$, both for the sum and for the carry. Hence, reducing the delay experienced by the carry of a full adder is a significant improvement. This is an incentive in seeking other implementations of the full adder.

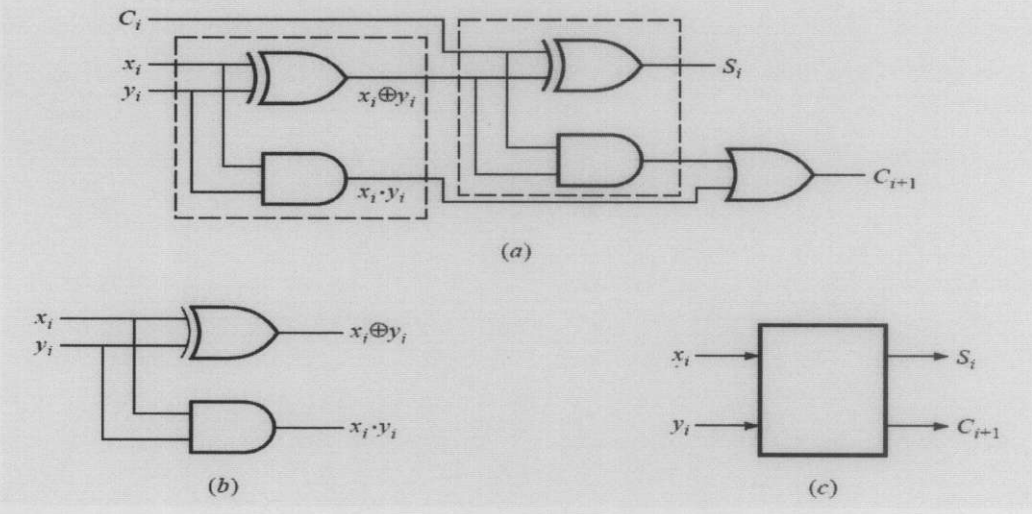


Figure 4.3: Full adder implemented with half adders. (*a*) Full adder. (*b*) Half adder. (*c*) Half adder schematic diagram.

4.2 Effect of Microprocessors Type on Halstead Method

The Halstead method was previously introduced in Chapter 2. This method is studied and analyzed throughout different generation of microprocessors. Halstead complexity metrics were developed by the late Maurice Halstead (1976) as a means of determining a quantitative measure of complexity directly from the operators and operands in the module to measure a program module's complexity directly from source code(Magnus A. et. al. 2004).

There is evidence that Halstead measures are also useful during development, to assess code quality in computationally dense applications. Because maintainability should be a concern during development, the Halstead measures should be considered for use during code development to follow complexity trends. Halstead's software science attempted to capture attributes of a program that paralleled physical and psychological measurements in other disciplines. Tokens of the following categories are all counted as operands by CMT++ (CMT++ is a code metric tool for C and C++) as shows in Table 4.1:

Table 4.1: Operands by CMT++

Operands	Specification
IDENTIFIER TYPENAME	all identifiers that are not reserved words
TYPESPEC (type specifiers)	Reserved words that specify type: *bool, char, double, float, int, long, short, signed, unsigned, void.* This class also includes some compiler specific nonstandard keywords.
CONSTANT	Character, numeric or string constants.

Tokens of the following categories are all counted as operators by CMT++ preprocessor directives as shows in Table 4.2:

Table 4.2: Operators by CMT++

Operators	Specification
SCSPEC (storage class specifiers)	Reserved words that specify storage class: *auto, extern, inlin, register, static, typedef, virtual, mtuable.*
TYPE_QUAL (type qualifiers)	Reserved words that qualify type: *const, friend, volatile.*
RESERVED	Other reserved words of C++: *asm, break, case, class, continue, default, delete, do, else, enum, for, goto, if, new, operator, private, protected, public, return, sizeof, struct, switch, this, union, while, namespace, using, try, catch, throw, const_cast, static_cast, dynamic_cast, reinterpret_cast, typeid, template, explicit, true, false, typename.* This class also includes some compiler specific nonstandard keywords.
OPERATOR	One of the following: ! != % %= & && || &= ( ) * *= + ++ += , - -- -= -> . ... / /= : :: < << <<= <= = == > >= >> >>= ? [ ] ^ ^= { } | |= ~

The following control structures *case ...: for (...), if (...), switch (...), while for (...),* and *catch (...)* are treated in a special way.

The *colon* and the *parentheses* are considered to be a part of the constructs. The *case* and the *colon* or the *for (...), if (...), switch (...), while* for (...), and *catch (...),* and the *parentheses* are counted together as one operator.

The identification of operators and operands depends on the programming language used. The measuring "Size of Vocabulary" and "Program Length" is rather self-explanatory. The volume of a program is akin to the number of mental comparisons needed to write a program of length N. It is supposed to correspond to the amount of computer storage necessary for a uniform binary encoding. Thus, Halstead has proposed reasonable measures of three internal program attributes that reflect different views of size. Table 4.3 shows proposed examples with assumed values of operators and operands to measuring the difficulty and effort by Halstead (equations 2.7 and 2.8) to show the effect of operators and operands on the program complexity. The basic metrics for these tokens where:

- μ_1 = number of unique operators
- μ_2 = number of unique operands
- N_1 = total occurrences of operators
- N_2 = total occurrences of operands

Figure 4.4 is showing the comparison between the total number of operators and effort taken from Table 4.3, according to Halstead method. It is clearly indicate that as long as numbers of operators or operands are increased then the difficulty and effort of the program increased.

Table 4.3: Examples of programs complexity by Halstead method.

Example	μ1	μ2	N1	N2	μ (Vocabulary)	N (Length)	Log_2 μtotal	V (Volume)	(μ1)/2	(N2)/ (μ2)	D (Difficulty)	E (Effort)
Ex01	2	3	2	4	5	6	2.3	13.93	1	1.3	1.3	18.6
Ex02	2	3	4	8	5	12	2.3	27.86	1	2.7	2.7	74.3
Ex03	2	3	6	12	5	18	2.3	41.79	1	4.0	4.0	167.2
Ex04	2	3	8	16	5	24	2.3	55.73	1	5.3	5.3	297.2
Ex05	2	3	10	20	5	30	2.3	69.66	1	6.7	6.7	464.4
Ex06	2	3	15	30	5	45	2.3	104.49	1	10.0	10.0	1044.9
Ex07	2	3	20	40	5	60	2.3	139.32	1	13.3	13.3	1857.5
Ex08	2	3	25	50	5	75	2.3	174.14	1	16.7	16.7	2902.4
Ex09	2	3	30	60	5	90	2.3	208.97	1	20.0	20.0	4179.5
Ex10	2	3	35	70	5	105	2.3	243.80	1	23.3	23.3	5688.7
Ex11	2	3	40	80	5	120	2.3	278.63	1	26.7	26.7	7430.2
Ex12	2	3	45	90	5	135	2.3	313.46	1	30.0	30.0	9403.8
Ex13	2	3	50	100	5	150	2.3	348.29	1	33.3	33.3	11609.6
Ex14	2	3	60	120	5	180	2.3	417.95	1	40.0	40.0	16717.9
Ex15	2	3	70	140	5	210	2.3	487.60	1	46.7	46.7	22754.9
Ex16	2	3	80	160	5	240	2.3	557.26	1	53.3	53.3	29720.7
Ex17	2	3	90	180	5	270	2.3	626.92	1	60.0	60.0	37615.2
Ex18	2	3	100	200	5	300	2.3	696.58	1	66.7	66.7	46438.6
Ex19	2	3	110	220	5	330	2.3	766.24	1	73.3	73.3	56190.7
Ex20	2	3	120	240	5	360	2.3	835.89	1	80.0	80.0	66871.5

Nevertheless, Halstead method is not taken in consideration the type of operators which have different execution time. For example, the execute time for multiplication function is more than execute time for add function as shown in Table 4.4. The Table shows the measurement of execution times for various computers. The 80286, 80486 and Pentium are three generations of personal computers. The Halstead method modify by taken in consideration the execution time for three types of microprocessor as shown in Table 4.4. The execution time to the remaining microprocessor such as Pentium 4 could not get due to security and know-how concept from the technology provider.

Table 4.4: Execution times for several generations of computers (Smith S. 1999).

	80286 (12 MHz)	80486 (33 MHz)	Pentium (100 MHz)
A = B + C	1.6	0.12	0.04
A = B - C	1.6	0.12	0.04
A = B * C	2.7	0.59	0.13
A = B / C	64	9.2	1.5

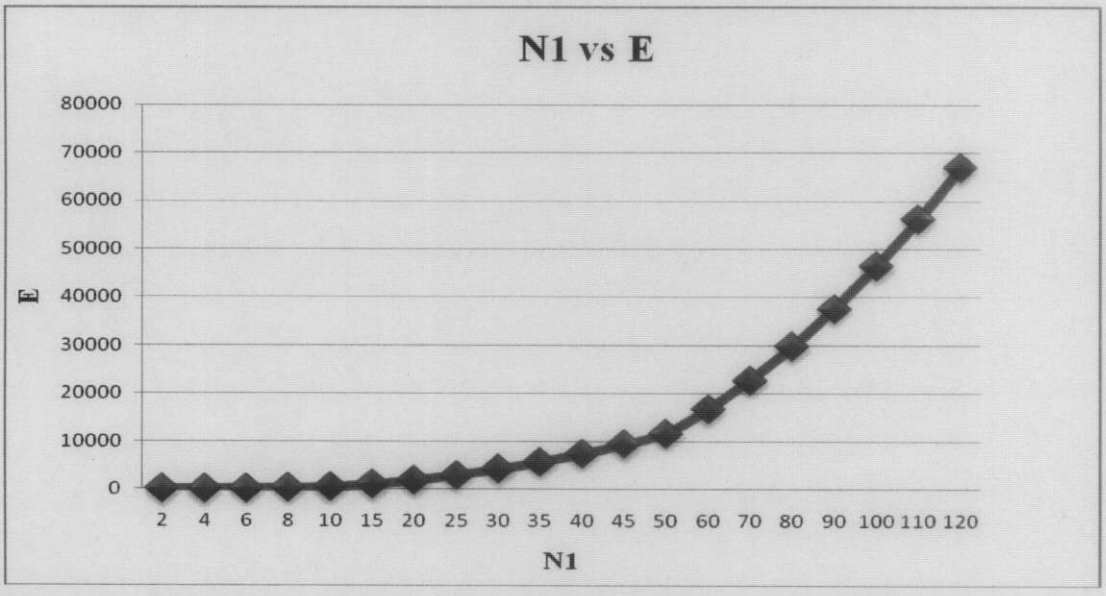


Figure 4.4: Compare between the total number of operators and effort according to Halstead method.

Table 4.5, Table 4.6 and Table 4.7 show examples according to modified method taken in consideration the execution time for add, multiplication and division functions respectively to 80286 microprocessor:

Table 4.5: Examples according to modified method taken in consideration the execution time for add function to 80286 microprocessor.

Example	μ1	μ2	N1	N2	μ (Vocabulary)	N^+ (Length)	Log_2 μ	V (Volume)	(μ1)/2	(N2)/ (μ2)	D (Difficulty)	E^+ (Effort)
Ex01	2	3	2	4	5	7.2	2.3	16.7	1.0	1.3	1.3	22
Ex02	2	3	4	8	5	14.4	2.3	33.4	1.0	2.7	2.7	89
Ex03	2	3	6	12	5	21.6	2.3	50.2	1.0	4.0	4.0	201
Ex04	2	3	8	16	5	28.8	2.3	66.9	1.0	5.3	5.3	357
Ex05	2	3	10	20	5	36.0	2.3	83.6	1.0	6.7	6.7	557
Ex06	2	3	15	30	5	54.0	2.3	125.4	1.0	10.0	10.0	1254
Ex07	2	3	20	40	5	72.0	2.3	167.2	1.0	13.3	13.3	2229
Ex08	2	3	25	50	5	90.0	2.3	209.0	1.0	16.7	16.7	3483
Ex09	2	3	30	60	5	108.0	2.3	250.8	1.0	20.0	20.0	5015
Ex10	2	3	35	70	5	126.0	2.3	292.6	1.0	23.3	23.3	6826
Ex11	2	3	40	80	5	144.0	2.3	334.4	1.0	26.7	26.7	8916
Ex12	2	3	45	90	5	162.0	2.3	376.2	1.0	30.0	30.0	11285
Ex13	2	3	50	100	5	180.0	2.3	417.9	1.0	33.3	33.3	13932
Ex14	2	3	60	120	5	216.0	2.3	501.5	1.0	40.0	40.0	20061
Ex15	2	3	70	140	5	252.0	2.3	585.1	1.0	46.7	46.7	27306
Ex16	2	3	80	160	5	288.0	2.3	668.7	1.0	53.3	53.3	35665
Ex17	2	3	90	180	5	324.0	2.3	752.3	1.0	60.0	60.0	45138
Ex18	2	3	100	200	5	360.0	2.3	835.9	1.0	66.7	66.7	55726
Ex19	2	3	110	220	5	396.0	2.3	919.5	1.0	73.3	73.3	67429
Ex20	2	3	120	240	5	432.0	2.3	1003.1	1.0	80.0	80.0	80246

From Table 4.5, the Example Ex01 shows the number of unique operators is 2 and number of unique operands is 3 as assumed in previous example on Table 4.3. Nevertheless, the effect of execution time will be studied taken in consideration the execution time for add function to 80286 microprocessor. Hence, the calculation based on the following formula:

$$N^+ = N_1^+ + N_2 \qquad \ldots(4.5)$$

Where N^+ is the length add of program and N_1^+ is the total occurrence of operators for add. From Table 4.4 the execution time for add function to 80286 microprocessor is 1.6 therefore, the total occurrence of operators for add on example Ex01 is:

$N_1^+ = 2 * 1.6 = 3.2$ Hence, $N^+ = N_1^+ + N_2 = 3.2 + 4 = 7.2$

Table 4.6: Examples according to modified method taken in consideration the execution time for multiplication function to 80286 microprocessor.

Example	μ1	μ2	N1	N2	μ (Vocabulary)	N* (Length)	Log_2 μ	V (Volume)	(μ1)/2	(N2)/ (μ2)	D (Difficulty)	E* (Effort)
Ex01	2	3	2	4	5	9.4	2.3	21.8	1.0	1.3	1.3	29
Ex02	2	3	4	8	5	18.8	2.3	43.7	1.0	2.7	2.7	116
Ex03	2	3	6	12	5	28.2	2.3	65.5	1.0	4.0	4.0	262
Ex04	2	3	8	16	5	37.6	2.3	87.3	1.0	5.3	5.3	466
Ex05	2	3	10	20	5	47	2.3	109.1	1.0	6.7	6.7	728
Ex06	2	3	15	30	5	70.5	2.3	163.7	1.0	10.0	10.0	1637
Ex07	2	3	20	40	5	94	2.3	218.3	1.0	13.3	13.3	2910
Ex08	2	3	25	50	5	117.5	2.3	272.8	1.0	16.7	16.7	4547
Ex09	2	3	30	60	5	141	2.3	327.4	1.0	20.0	20.0	6548
Ex10	2	3	35	70	5	164.5	2.3	382.0	1.0	23.3	23.3	8912
Ex11	2	3	40	80	5	188	2.3	436.5	1.0	26.7	26.7	11641
Ex12	2	3	45	90	5	211.5	2.3	491.1	1.0	30.0	30.0	14733
Ex13	2	3	50	100	5	235	2.3	545.7	1.0	33.3	33.3	18188
Ex14	2	3	60	120	5	282	2.3	654.8	1.0	40.0	40.0	26191
Ex15	2	3	70	140	5	329	2.3	763.9	1.0	46.7	46.7	35649
Ex16	2	3	80	160	5	376	2.3	873.0	1.0	53.3	53.3	46562
Ex17	2	3	90	180	5	423	2.3	982.2	1.0	60.0	60.0	58931
Ex18	2	3	100	200	5	470	2.3	1091.3	1.0	66.7	66.7	72754
Ex19	2	3	110	220	5	517	2.3	1200.4	1.0	73.3	73.3	88032
Ex20	2	3	120	240	5	564	2.3	1309.6	1.0	80.0	80.0	104765

From Table 4.6, the Example Ex01 shows the number of unique operators is 2 and number of unique operands is 3 as assumed in previous example on Table 4.3. Nevertheless, the effect of execution time will be studied taken in consideration the execution time for multiplication function to 80286 microprocessor. Hence, the calculation based on the following formula:

$$N^* = N_1^* + N_2 \qquad \ldots(4.6)$$

Where N^* is the length multiplication of program and N_1^* is the total occurrence for multiplication. From Table 4.4 the execution time for multiplication function to 80286

microprocessor is 2.7 therefore, the total occurrence operators for multiplication on example Ex01 is:

$N_1^* = 2 * 2.7 = 5.4$ Hence, $N^* = N_1^* + N_2 = 5.4 + 4 = 9.4$

Table 4.7: Examples according to modified method taken in consideration the execution time for division function to 80286 microprocessor.

Example	μ1	μ2	N1	N2	μ (Vocabulary)	N′ (Length)	Log_2 μ	V (Volume)	(μ1)/2	(N2)/ (μ2)	D (Difficulty)	E′ (Effort)
Ex01	2	3	2	4	5	132	2.3	306.5	1.0	1.3	1.3	409
Ex02	2	3	4	8	5	264	2.3	613.0	1.0	2.7	2.7	1635
Ex03	2	3	6	12	5	396	2.3	919.5	1.0	4.0	4.0	3678
Ex04	2	3	8	16	5	528	2.3	1226.0	1.0	5.3	5.3	6539
Ex05	2	3	10	20	5	660	2.3	1532.5	1.0	6.7	6.7	10216
Ex06	2	3	15	30	5	990	2.3	2298.7	1.0	10.0	10.0	22987
Ex07	2	3	20	40	5	1320	2.3	3064.9	1.0	13.3	13.3	40866
Ex08	2	3	25	50	5	1650	2.3	3831.2	1.0	16.7	16.7	63853
Ex09	2	3	30	60	5	1980	2.3	4597.4	1.0	20.0	20.0	91948
Ex10	2	3	35	70	5	2310	2.3	5363.7	1.0	23.3	23.3	125152
Ex11	2	3	40	80	5	2640	2.3	6129.9	1.0	26.7	26.7	163464
Ex12	2	3	45	90	5	2970	2.3	6896.1	1.0	30.0	30.0	206884
Ex13	2	3	50	100	5	3300	2.3	7662.4	1.0	33.3	33.3	255412
Ex14	2	3	60	120	5	3960	2.3	9194.8	1.0	40.0	40.0	367793
Ex15	2	3	70	140	5	4620	2.3	10727.3	1.0	46.7	46.7	500608
Ex16	2	3	80	160	5	5280	2.3	12259.8	1.0	53.3	53.3	653855
Ex17	2	3	90	180	5	5940	2.3	13792.3	1.0	60.0	60.0	827535
Ex18	2	3	100	200	5	6600	2.3	15324.7	1.0	66.7	66.7	1021648
Ex19	2	3	110	220	5	7260	2.3	16857.2	1.0	73.3	73.3	1236195
Ex20	2	3	120	240	5	7920	2.3	18389.7	1.0	80.0	80.0	1471174

From Table 4.7, the Example Ex01 shows the number of unique operators is 2 and number of unique operands is 3 as assumed in previous example on Table 4.3. Nevertheless, the effect of execution time will be studied taken in consideration the execution time for division function to 80286 microprocessor. Hence, the calculation based on the following formula:

$N' = N_1' + N_2$ ...(4.7)

Where N^* is the length division of program and N_1^* is the total occurrence of operators for division. From Table 4.4 the execution time for division function to 80286 microprocessor is 64 therefore, the total occurrence of operators for division on example Ex01 is:

$N_1' = 2 * 64 = 128$ Hence, $N' = N_1' + N_2 = 128 + 4 = 132$

Figure 4.9 compare between the total number of operators and effort according to modified method and taken in consideration the execution time for add, multiplication and division function to 80286 microprocessor and found that there are diverse due to different execution time of each function. Also, the Figure 4.5 shows that as number of operators (N1) is increased, the effort (E) value will increased accordingly.

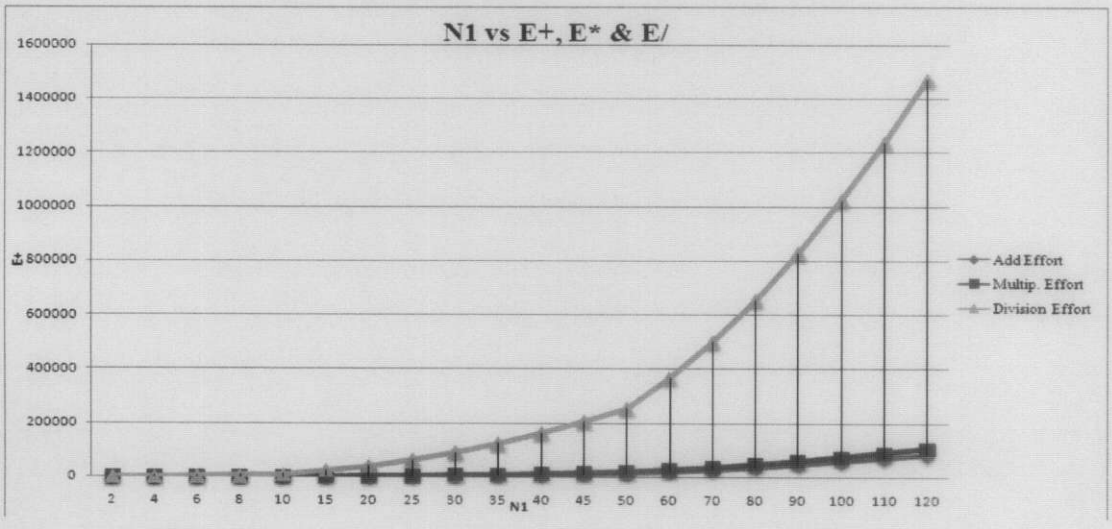


Figure 4.5: Compare between the total number of operators and efforts according to modified method to 80286 microprocessor.

Table 4.8, Table 4.9 and Table 4.10 show examples according to modified method taken in consideration the execution time for add, multiplication and division functions respectively to 80486 microprocessor:

Table 4.8: Examples according to modified method taken in consideration the execution time for add function to 80486 microprocessor.

Example	μ1	μ2	N1	N2	μ (Vocabulary)	N^+ (Length)	Log_2 μ	V (Volume)	(μ1)/2	(N2)/ (μ2)	D (Difficulty)	E^+ (Effort)
Ex01	2	3	2	4	5	4.24	2.3	9.8	1.0	1.3	1.3	13.1
Ex02	2	3	4	8	5	8.48	2.3	19.7	1.0	2.7	2.7	52.5
Ex03	2	3	6	12	5	12.72	2.3	29.5	1.0	4.0	4.0	118.1
Ex04	2	3	8	16	5	16.96	2.3	39.4	1.0	5.3	5.3	210.0
Ex05	2	3	10	20	5	21.2	2.3	49.2	1.0	6.7	6.7	328.2
Ex06	2	3	15	30	5	31.8	2.3	73.8	1.0	10.0	10.0	738.4
Ex07	2	3	20	40	5	42.4	2.3	98.4	1.0	13.3	13.3	1312.7
Ex08	2	3	25	50	5	53	2.3	123.1	1.0	16.7	16.7	2051.0
Ex09	2	3	30	60	5	63.6	2.3	147.7	1.0	20.0	20.0	2953.5
Ex10	2	3	35	70	5	74.2	2.3	172.3	1.0	23.3	23.3	4020.0
Ex11	2	3	40	80	5	84.8	2.3	196.9	1.0	26.7	26.7	5250.7
Ex12	2	3	45	90	5	95.4	2.3	221.5	1.0	30.0	30.0	6645.4
Ex13	2	3	50	100	5	106	2.3	246.1	1.0	33.3	33.3	8204.1
Ex14	2	3	60	120	5	127.2	2.3	295.3	1.0	40.0	40.0	11814.0
Ex15	2	3	70	140	5	148.4	2.3	344.6	1.0	46.7	46.7	16080.1
Ex16	2	3	80	160	5	169.6	2.3	393.8	1.0	53.3	53.3	21002.6
Ex17	2	3	90	180	5	190.8	2.3	443.0	1.0	60.0	60.0	26581.4
Ex18	2	3	100	200	5	212	2.3	492.2	1.0	66.7	66.7	32816.6
Ex19	2	3	110	220	5	233.2	2.3	541.5	1.0	73.3	73.3	39708.1
Ex20	2	3	120	240	5	254.4	2.3	590.7	1.0	80.0	80.0	47255.9

From Table 4.8, the Example Ex01 shows the number of unique operators is 2 and number of unique operands is 3 as assumed in previous example on Table 4.3. Nevertheless, the effect of execution time will be studied taken in consideration the execution time for add function to 80486 microprocessor.

From Table 4.4 the execution time for add function to 80486 microprocessor is 0.12 therefore, the total occurrence of operators for add on example Ex01 is:

$N_1^+ = 2 * 0.12 = 0.24$ Hence, $N^+ = N_1^+ + N_2 = 0.24 + 4 = 4.24$

Table 4.9: Examples according to modified method taken in consideration the execution time for multiplication function to 80486 microprocessor.

Example	μ1	μ2	N1	N2	μ (Vocabulary)	N* (Length)	Log_2 μ	V (Volume)	(μ1)/2	(N2)/ (μ2)	D (Difficulty)	E* (Effort)
Ex01	2	3	2	4	5	5.18	2.3	12.0	1.0	1.3	1.3	16.0
Ex02	2	3	4	8	5	10.36	2.3	24.1	1.0	2.7	2.7	64.1
Ex03	2	3	6	12	5	15.54	2.3	36.1	1.0	4.0	4.0	144.3
Ex04	2	3	8	16	5	20.72	2.3	48.1	1.0	5.3	5.3	256.6
Ex05	2	3	10	20	5	25.9	2.3	60.1	1.0	6.7	6.7	400.9
Ex06	2	3	15	30	5	38.85	2.3	90.2	1.0	10.0	10.0	902.1
Ex07	2	3	20	40	5	51.8	2.3	120.3	1.0	13.3	13.3	1603.7
Ex08	2	3	25	50	5	64.75	2.3	150.3	1.0	16.7	16.7	2505.7
Ex09	2	3	30	60	5	77.7	2.3	180.4	1.0	20.0	20.0	3608.3
Ex10	2	3	35	70	5	90.65	2.3	210.5	1.0	23.3	23.3	4911.3
Ex11	2	3	40	80	5	103.6	2.3	240.6	1.0	26.7	26.7	6414.7
Ex12	2	3	45	90	5	116.55	2.3	270.6	1.0	30.0	30.0	8118.6
Ex13	2	3	50	100	5	129.5	2.3	300.7	1.0	33.3	33.3	10023.0
Ex14	2	3	60	120	5	155.4	2.3	360.8	1.0	40.0	40.0	14433.1
Ex15	2	3	70	140	5	181.3	2.3	421.0	1.0	46.7	46.7	19645.1
Ex16	2	3	80	160	5	207.2	2.3	481.1	1.0	53.3	53.3	25658.9
Ex17	2	3	90	180	5	233.1	2.3	541.2	1.0	60.0	60.0	32474.5
Ex18	2	3	100	200	5	259	2.3	601.4	1.0	66.7	66.7	40092.0
Ex19	2	3	110	220	5	284.9	2.3	661.5	1.0	73.3	73.3	48511.3
Ex20	2	3	120	240	5	310.8	2.3	721.7	1.0	80.0	80.0	57732.4

From Table 4.9, the Example Ex01 shows the number of unique operators is 2 and number of unique operands is 3 as assumed in previous example on Table 4.3. Nevertheless, the effect of execution time will be studied taken in consideration the execution time for multiplication function to 80486 microprocessor.

From Table 4.4 the execution time for multiplication function to 80486 microprocessor is 0.59 therefore, the total occurrence of operators for multiplication on example Ex01 is:

$N_1^* = 2 * 0.59 = 1.18$ Hence, $N^* = N_1^* + N_2 = 1.18 + 4 = 5.18$

Table 4.10: Examples according to modified method taken in consideration the execution time for division function to 80486 microprocessor.

Example	μ1	μ2	N1	N2	μ (Vocabulary)	N' (Length)	Log_2 μ	V (Volume)	(μ1)/2	(N2)/ (μ2)	D (Difficulty)	E' (Effort)
Ex01	2	3	2	4	5	22.4	2.3	52.0	1.0	1.3	1.3	69.3
Ex02	2	3	4	8	5	44.8	2.3	104.0	1.0	2.7	2.7	277.4
Ex03	2	3	6	12	5	67.2	2.3	156.0	1.0	4.0	4.0	624.1
Ex04	2	3	8	16	5	89.6	2.3	208.0	1.0	5.3	5.3	1109.6
Ex05	2	3	10	20	5	112	2.3	260.1	1.0	6.7	6.7	1733.7
Ex06	2	3	15	30	5	168	2.3	390.1	1.0	10.0	10.0	3900.8
Ex07	2	3	20	40	5	224	2.3	520.1	1.0	13.3	13.3	6934.8
Ex08	2	3	25	50	5	280	2.3	650.1	1.0	16.7	16.7	10835.7
Ex09	2	3	30	60	5	336	2.3	780.2	1.0	20.0	20.0	15603.4
Ex10	2	3	35	70	5	392	2.3	910.2	1.0	23.3	23.3	21237.9
Ex11	2	3	40	80	5	448	2.3	1040.2	1.0	26.7	26.7	27739.3
Ex12	2	3	45	90	5	504	2.3	1170.3	1.0	30.0	30.0	35107.6
Ex13	2	3	50	100	5	560	2.3	1300.3	1.0	33.3	33.3	43342.7
Ex14	2	3	60	120	5	672	2.3	1560.3	1.0	40.0	40.0	62413.4
Ex15	2	3	70	140	5	784	2.3	1820.4	1.0	46.7	46.7	84951.6
Ex16	2	3	80	160	5	896	2.3	2080.4	1.0	53.3	53.3	110957.2
Ex17	2	3	90	180	5	1008	2.3	2340.5	1.0	60.0	60.0	140430.2
Ex18	2	3	100	200	5	1120	2.3	2600.6	1.0	66.7	66.7	173370.6
Ex19	2	3	110	220	5	1232	2.3	2860.6	1.0	73.3	73.3	209778.5
Ex20	2	3	120	240	5	1344	2.3	3120.7	1.0	80.0	80.0	249653.7

From Table 4.10, the Example Ex01 shows the number of unique operators is 2 and number of unique operands is 3 as assumed in previous example on Table 4.3. Nevertheless, the effect of execution time will be studied taken in consideration the execution time for division function to 80486 microprocessor.

From Table 4.4 the execution time for division function to 80486 microprocessor is 9.2 therefore, the total occurrence of operators for division on example Ex01 is:

$N_1' = 2 * 9.2 = 18.4$ Hence, $N' = N_1' + N_2 = 18.4 + 4 = 22.4$

Figure 4.6 compare between the total number of operators and effort according to modified method and taken in consideration the execution time for add, multiplication and division function to 80486 microprocessor as we did previously for 80286 microprocessor and found that there are diverse in this case also due to different execution time of each function as mentioned before. Also, the Figure 4.10 shows that as number of operators (N1) is increased, the effort (E) value will increased accordingly.

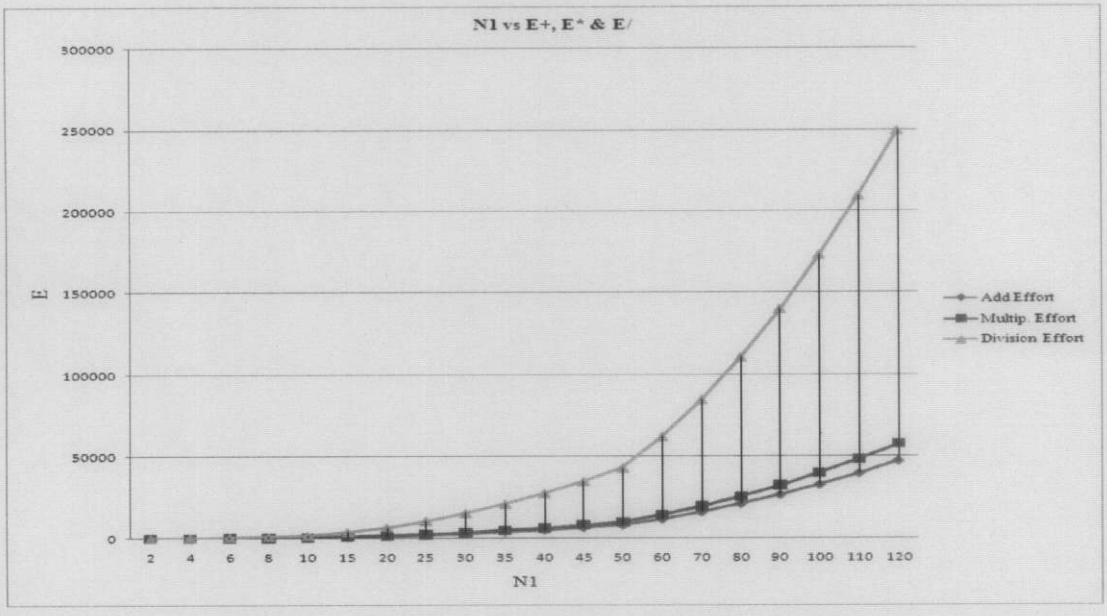


Figure 4.6: Compare between the total number of operators and efforts according to modified method to 80486 microprocessor.

Table 4.11, Table 4.12 and Table 4.13 show examples according to modified method taken in consideration the execution time for add, multiplication and division functions respectively to Pentium microprocessor.

Table 4.11: Examples according to modified method taken in consideration the execution time for add function to Pentium microprocessor.

Example	μ1	μ2	N1	N2	μ (Vocabulary)	N^+ (Length)	Log_2 μ	V (Volume)	(μ1)/2	(N2)/ (μ2)	D (Difficulty)	E^+ (Effort)
Ex01	2	3	2	4	5	4.08	2.3	9.5	1.0	1.3	1.3	12.6
Ex02	2	3	4	8	5	8.16	2.3	18.9	1.0	2.7	2.7	50.5
Ex03	2	3	6	12	5	12.24	2.3	28.4	1.0	4.0	4.0	113.7
Ex04	2	3	8	16	5	16.32	2.3	37.9	1.0	5.3	5.3	202.1
Ex05	2	3	10	20	5	20.4	2.3	47.4	1.0	6.7	6.7	315.8
Ex06	2	3	15	30	5	30.6	2.3	71.1	1.0	10.0	10.0	710.5
Ex07	2	3	20	40	5	40.8	2.3	94.7	1.0	13.3	13.3	1263.1
Ex08	2	3	25	50	5	51	2.3	118.4	1.0	16.7	16.7	1973.6
Ex09	2	3	30	60	5	61.2	2.3	142.1	1.0	20.0	20.0	2842.0
Ex10	2	3	35	70	5	71.4	2.3	165.8	1.0	23.3	23.3	3868.3
Ex11	2	3	40	80	5	81.6	2.3	189.5	1.0	26.7	26.7	5052.5
Ex12	2	3	45	90	5	91.8	2.3	213.2	1.0	30.0	30.0	6394.6
Ex13	2	3	50	100	5	102	2.3	236.8	1.0	33.3	33.3	7894.6
Ex14	2	3	60	120	5	122.4	2.3	284.2	1.0	40.0	40.0	11368.2
Ex15	2	3	70	140	5	142.8	2.3	331.6	1.0	46.7	46.7	15473.3
Ex16	2	3	80	160	5	163.2	2.3	378.9	1.0	53.3	53.3	20210.1
Ex17	2	3	90	180	5	183.6	2.3	426.3	1.0	60.0	60.0	25578.4
Ex18	2	3	100	200	5	204	2.3	473.7	1.0	66.7	66.7	31578.2
Ex19	2	3	110	220	5	224.4	2.3	521.0	1.0	73.3	73.3	38209.6
Ex20	2	3	120	240	5	244.8	2.3	568.4	1.0	80.0	80.0	45472.6

From Table 4.11, the Example Ex01 shows the number of unique operators is 2 and number of unique operands is 3 as assumed in previous example on Table 4.3. Nevertheless, the effect of execution time will be studied taken in consideration the execution time for add function to Pentium microprocessor.

From Table 4.4 the execution time for add function to Pentium microprocessor is 0.04 therefore, the total occurrence of operators for add on example Ex01 is:

$N_1^+ = 2 * 0.04 = 0.08$ Hence, $N^+ = N_1^+ + N_2 = 0.08 + 4 = 4.08$

Table 4.12: Examples according to modified method taken in consideration the execution time for multiplication function to Pentium microprocessor.

Example	μ1	μ2	N1	N2	μ (Vocabulary)	N^* (Length)	Log_2 μ	V (Volume)	(μ1)/2	(N2)/ (μ2)	D (Difficulty)	E^* (Effort)
Ex01	2	3	2	4	5	4.26	2.3	9.9	1.0	1.3	1.3	13.2
Ex02	2	3	4	8	5	8.52	2.3	19.8	1.0	2.7	2.7	52.8
Ex03	2	3	6	12	5	12.78	2.3	29.7	1.0	4.0	4.0	118.7
Ex04	2	3	8	16	5	17.04	2.3	39.6	1.0	5.3	5.3	211.0
Ex05	2	3	10	20	5	21.3	2.3	49.5	1.0	6.7	6.7	329.7
Ex06	2	3	15	30	5	31.95	2.3	74.2	1.0	10.0	10.0	741.9
Ex07	2	3	20	40	5	42.6	2.3	98.9	1.0	13.3	13.3	1318.9
Ex08	2	3	25	50	5	53.25	2.3	123.6	1.0	16.7	16.7	2060.7
Ex09	2	3	30	60	5	63.9	2.3	148.4	1.0	20.0	20.0	2967.4
Ex10	2	3	35	70	5	74.55	2.3	173.1	1.0	23.3	23.3	4039.0
Ex11	2	3	40	80	5	85.2	2.3	197.8	1.0	26.7	26.7	5275.4
Ex12	2	3	45	90	5	95.85	2.3	222.6	1.0	30.0	30.0	6676.7
Ex13	2	3	50	100	5	106.5	2.3	247.3	1.0	33.3	33.3	8242.8
Ex14	2	3	60	120	5	127.8	2.3	296.7	1.0	40.0	40.0	11869.7
Ex15	2	3	70	140	5	149.1	2.3	346.2	1.0	46.7	46.7	16156.0
Ex16	2	3	80	160	5	170.4	2.3	395.7	1.0	53.3	53.3	21101.7
Ex17	2	3	90	180	5	191.7	2.3	445.1	1.0	60.0	60.0	26706.8
Ex18	2	3	100	200	5	213	2.3	494.6	1.0	66.7	66.7	32971.4
Ex19	2	3	110	220	5	234.3	2.3	544.0	1.0	73.3	73.3	39895.4
Ex20	2	3	120	240	5	255.6	2.3	593.5	1.0	80.0	80.0	47478.8

From Table 4.12, the Example Ex01 shows the number of unique operators is 2 and number of unique operands is 3 as assumed in previous example on Table 4.3. Nevertheless, the effect of execution time will be studied taken in consideration the execution time for multiplication function to Pentium microprocessor.

From Table 4.4 the execution time for multiplication function to Pentium microprocessor is 0.13 therefore, the total occurrence of operators for multiplication on example Ex01 is:

$N_1^* = 2 * 0.13 = 0.26$ Hence, $N^* = N_1^* + N_2 = 0.26 + 4 = 4.26$

Table 4.13: Examples according to modified method taken in consideration the execution time for division function to Pentium microprocessor.

Example	μ1	μ2	N1	N2	μ (Vocabulary)	N′ (Length)	Log_2 μ	V (Volume)	(μ1)/2	(N2)/ (μ2)	D (Difficulty)	E′ (Effort)
Ex01	2	3	2	4	5	7	2.3	16.3	1.0	1.3	1.3	21.7
Ex02	2	3	4	8	5	14	2.3	32.5	1.0	2.7	2.7	86.7
Ex03	2	3	6	12	5	21	2.3	48.8	1.0	4.0	4.0	195.0
Ex04	2	3	8	16	5	28	2.3	65.0	1.0	5.3	5.3	346.7
Ex05	2	3	10	20	5	35	2.3	81.3	1.0	6.7	6.7	541.8
Ex06	2	3	15	30	5	52.5	2.3	121.9	1.0	10.0	10.0	1219.0
Ex07	2	3	20	40	5	70	2.3	162.5	1.0	13.3	13.3	2167.1
Ex08	2	3	25	50	5	87.5	2.3	203.2	1.0	16.7	16.7	3386.1
Ex09	2	3	30	60	5	105	2.3	243.8	1.0	20.0	20.0	4876.0
Ex10	2	3	35	70	5	122.5	2.3	284.4	1.0	23.3	23.3	6636.8
Ex11	2	3	40	80	5	140	2.3	325.1	1.0	26.7	26.7	8668.5
Ex12	2	3	45	90	5	157.5	2.3	365.7	1.0	30.0	30.0	10971.1
Ex13	2	3	50	100	5	175	2.3	406.3	1.0	33.3	33.3	13544.6
Ex14	2	3	60	120	5	210	2.3	487.6	1.0	40.0	40.0	19504.2
Ex15	2	3	70	140	5	245	2.3	568.9	1.0	46.7	46.7	26547.4
Ex16	2	3	80	160	5	280	2.3	650.1	1.0	53.3	53.3	34674.1
Ex17	2	3	90	180	5	315	2.3	731.4	1.0	60.0	60.0	43884.4
Ex18	2	3	100	200	5	350	2.3	812.7	1.0	66.7	66.7	54178.3
Ex19	2	3	110	220	5	385	2.3	893.9	1.0	73.3	73.3	65555.8
Ex20	2	3	120	240	5	420	2.3	975.2	1.0	80.0	80.0	78016.8

From Table 4.13, the Example Ex01 shows the number of unique operators is 2 and number of unique operands is 3 as assumed in previous example on Table 4.3. Nevertheless, the effect of execution time will be studied taken in consideration the execution time for division function to Pentium microprocessor.

From Table 4.4 the execution time for division function to Pentium microprocessor is 1.5 therefore, the total occurrence of operators for division on example Ex01 is:

$N_1' = 2 * 1.5 = 3$ Hence, $N' = N_1' + N_2 = 3 + 4 = 7$

Figure 4.7 compare between the total number of operators and effort according to modified method and taken in consideration the execution time for add, multiplication and division function for Pentium microprocessor as did previously for 80286 and 80486 microprocessors and found that there are various in this case also due to different execution time of each function. Also, the Figure 4.11 shows that as number of operators (N1) is increased, the effort (E) value will increased accordingly.

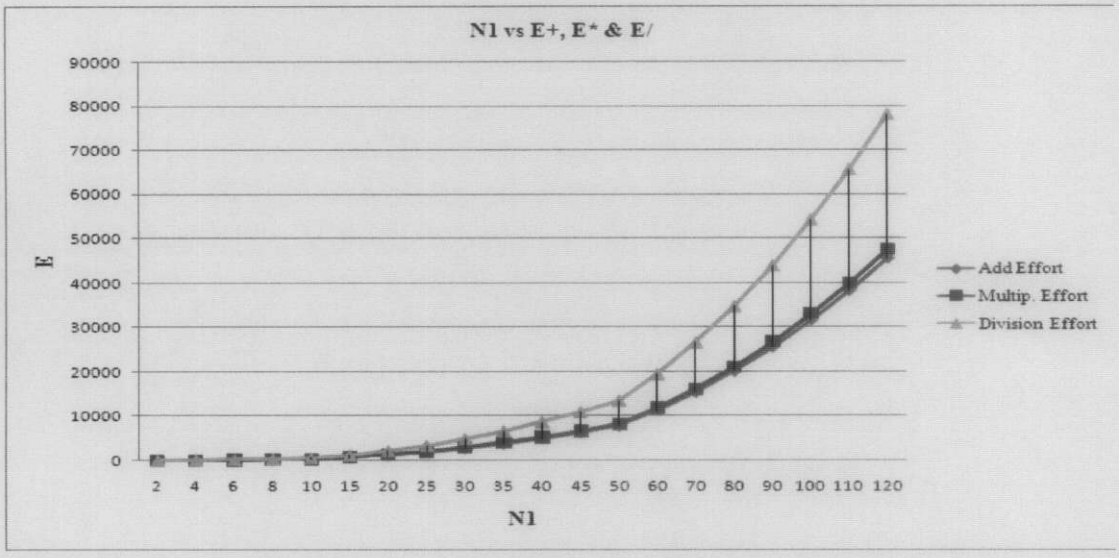


Figure 4.7: Compare between the total number of operators and efforts according to modified to Pentium microprocessor.

4.3 Conclusion

Software complexity measure using Halstead method is exercised in this chapter. It is found that as long as number of operators or operands get increased the difficulty and effort of the program is increased as declare by Halstead method. Moreover, Halstead method does not take into consideration the type of operators which have different execution time. Investigation for the execution time for addition, multiplication, and division functions shows that they are different for each function. Also the factors like, vocabulary, volume, difficulty, and effort are different for one function to another. The experiment is done for three different generations of personal computers namely: 80286, 80486, and Pentium.

العنوان:	Measuring software complexity for software engineering quality assurance
المؤلف الرئيسي:	Wohaishi, Ahmad Muhammad Ali
مؤلفين آخرين:	Fahmy, Maged A.، Al Sultanny, Yas A.(Co-Advisor، Advisor)
التاريخ الميلادي:	2009
موقع:	المنامة
الصفحات:	1 - 155
رقم MD:	736039
نوع المحتوى:	رسائل جامعية
اللغة:	English
الدرجة العلمية:	رسالة ماجستير
الجامعة:	جامعة الخليج العربي
الكلية:	كلية الدراسات العليا
الدولة:	البحرين
قواعد المعلومات:	Dissertations
مواضيع:	برامج الحاسوب، هندسة البرمجيات، تكنولوجيا المعلومات
رابط:	https://search.mandumah.com/Record/736039

Chapter 5

Software Complexity Based on Cognitive Weights

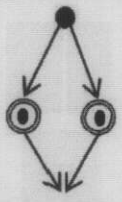
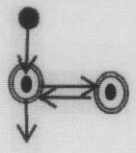
5.1 Introduction

Cognitive weight is a metric to measure the effort required for comprehending a piece of software. Cognitive weights are the basic control structures (BCS). Table 5.1 shows the basic control structures and its cognitive weights which are a measure for the difficulty to understand a control structure (Shao J. et. al 2003).

Table 5.1: Definition of BCSs and their equivalent cognitive weights (Wž)

Category	BCS	structure	*Wž*
Sequence	Sequence (SEQ)		1
Branch	If-then-[else] (ITE)		2
	Case (CASE)		3
Iteration	For-do (Ri)		3
	Repeat-until (R1)		3
	while-do (R0)		3

Continue Table 5.1: Definition of BCSs and their equivalent cognitive weights (Wž)

Category	BCS	structure	*Wž*
Embedded component	Function call(FC)		2
	Recursion (REC)		3
Concurrency	Parallel (PAR)		4
	Interrupt (INT)		4

The cognitive weight of a software component without nested control structures is defined as the sum of the cognitive weights of its control structures according to Table 5.1. The idea behind cognitive weights is to regard basic control structures as patterns that can be understood by the reader as a whole. This approach is based on automatically finding well known architectural patterns in a UML model. The idea behind this approach is that well documented patterns have been found highly mature and using them helps to improve code quality, understandability and maintainability. Obviously, this assertion should be regarded with care: Architectural patterns are only useful if they are used by experienced programmers in the right way, and an extensive use of patterns does not have to mean anything for the quality of the code. So, if the approach is used, a deep knowledge about the patterns and their correct usage is necessary. Cognitive Informatics is a new research area that combines concepts from cognitive sciences and informatics (Cardoso 2006).

Cognitive Weights of a Software Basic control structures (BCS) such as sequence, branch and iteration are the basic logic building blocks of any software and the cognitive weights (W_C) of a software is the extent of difficulty or relative time and effort for comprehending a given software modeled by a number of BCS's. These cognitive weights for BCS's measure the complexity of logical structures of the software. Either all the BCS's are in a linear layout or some BCS's are embedded in others. For the former case, the weights of all the BCS's are summed and for the latter, cognitive weights of inner BCS's are multiplied with the weight of external BCS's (Klemola T. 2000).

An important issue encountered in software complexity analysis is the consideration of software as a human creative artifact and the development of a suitable measure that recognizes this fundamental characteristic. The existing measures for software complexity can be classified into two categories: the macro and the micro measures of software complexity. Major macro complexity measures of software have been proposed by Basili and Kearney in, 1980. They viewed the complexity in terms of the degree of difficulty in programming. The micro measures are based on program code, disregarding comments and stylistic attributes. This type of measure typically depends on program size, program flow graphs, or module interfaces such as Halstead's software science metrics and the most widely known cyclomatic complexity measure developed by McCabe. However, Halstead's software metrics merely calculate the number of operators and operands; they do not consider the internal structures of software components; while McCabe's cyclomatic measure does not consider I/Os of software systems. In cognitive informatics, it is found that the functional complexity of software in design and comprehension is dependent on three fundamental factors: internal processing, input and output. Cognitive complexity is a measure of the cognitive and psychological complexity of software as a human intelligence artifact. Cognitive complexity takes into account both internal structures of software and the I/Os it processes (Boy G. 2005).

5.2 The Cognitive Weight of Software (Cognitive Functional Size)

Cognitive complexity is related to cognitive psychology that aims at studying, among other things, thinking, reasoning, and decision making. The understanding of cognitive complexity is to divide the memory into long term and short term memory. The short term memory limit the duration of storage to less than about 30 seconds whereas the long term memory can last as little as 30 seconds or as long as decades. The chunk of processes that can be captured and stored by a short term memory would be determined as meaningful (Abdul Ghani A. et. al. 2008).

To comprehend a given program, the focus should be on the architecture and basic control structures (BCSs) of the software. BCSs are a set of essential flow control mechanisms that are used for building logical software architectures. Three BCSs are commonly identified: the sequential, branch, and iteration structures. Although it can be proven that an iteration may be represented by the combination of sequential and branch structures, it is convenient to keep iteration as an independent BCS. In addition, two advanced BCSs in system modeling, known as recursion and parallel, have been described by Hoare, 1987. Wang in 2003 extended the set of BCSs to cover function call and interrupt. The cognitive weight of software is the degree of difficulty or relative time and effort required for comprehending a given piece of software modeled by a number of BCSs. The five categories of BCSs described above profound architectural attributes of software systems, where the equivalent cognitive weights (W_i) of each BCS for determining a component's functionality and complexity are defined based on empirical studies in cognitive informatics (Wang Y. 2003).

There are two different architectures for calculating Weights of a Software Basic control structures (Wbcs): either all the BCSs are in a linear layout or some BCSs are embedded in others. For the former case, we may sum the weights of all n BCSs; for the latter, we can multiply the cognitive weights of inner BCSs with the weights of external BCSs. In a generic case, the two types of architectures are combined in various ways. Therefore a general method can be defined as follows, the total cognitive weight of a software

component, Wc, is defined as the sum of the cognitive weights of its q linear blocks composed of individual BCSs. A component's cognitive functional size is found to be proportional to the total weighted cognitive complexity of all internal BCSs and the number of inputs (N_i) and outputs (N_o). In other words, CFS is a function of the three fundamental factors: W_c, N_i, and N_o. Thus, an equivalent cognitive unit of software (CWU) can be defined as the cognitive weight of the simplest software component with only a single I/O and a linear structured BCS. The cognitive functional size of a basic software component that only consists of one method, S_f, is defined as a product of sum of inputs and outputs ($N_{i/o}$) and the total cognitive weight, i.e.,

$$S_f = (N_i + N_o) * W_c \qquad \ldots(5.1)$$

5.3 Formal Description of Software Cognitive Complexity

Software quality, from a cognitive informatics point of view, is defined as the completeness, correctness, consistency, feasibility, and verifiability of the software in both specification and implementation, with no misinterpretation and no ambiguity. Therefore, quality software relies on the explicit description of three dimensional behaviors known as the architecture, static behaviors, and dynamic behaviors. Program comprehension is then a cognitive process to understand a given software system in terms of these three dimensions and their relationships.

Formal methods provide a rigorous way to describe software systems in order to ensure a higher quality of design and implementation. It is found that the complexity of software can be analyzed on the basis of formal specifications early in the development process before code is available. Real-time process algebra is used to demonstrate how the new measurement of software complexity can be analyzed based on formal specifications. Conventional formal methods are based on logic and set theories that lack the capability to describe software architectures and dynamic behaviors. Real Time Process Algebra (RTPA) is designed to describe and specify architectures and behaviors of software systems formally and precisely. RTPA models 16 meta processes and 16 process relations, as partially shown in Table 5.1. A meta process is an elementary and primary process that

serves as a common and basic building block for a software system. Complex processes can be derived from meta processes by a set of process relations that serves as the process combinatory rules. RTPA has been developed as an expressive, easy-to-comprehend, and language-independent notation system, and a specification and refinement method for software description and specification. Based on the above analysis, the cognitive functional size is feasibly obtained to measure software complexity. Due to the precision and rigorousness of RTPA, the measurement of the complexity of software systems can be obtained in an early phase of system development.

5.4 Robustness of the Cognitive Complexity Measure

To analyze the robustness of the new measure of cognitive functional size (S_f) and its relationship with the physical size of software (S_p), a large set of examples and experimental have carried out and analysis. In this subsection, 30 programs are selected from Deitel and Deitel book "C++ How to Program" edition 2007. Each program is analyzed in terms of the physical size (with the unit known as lines of code or LOC) and cognitive functional size as shown in Table 5.2.

Figure 5.1 shows S_f and Sp to the (30) programs. The physical size (S_p) of software, or program length, can be used as a predictor of program characteristics such as effort and difficulty of maintenance. However, it characterizes only one specific aspect of size, namely the static length, because it takes no account of functionality. The cognitive functional size (S_f) is concerned with functional and cognitive complexity. An interesting relationship between S_f and S_p, known as coding efficiency (E_c), can be derived as shown below:

$$E_c = S_f / S_p [\text{CWU/LOC}] \qquad \ldots(5.2)$$

Example P1 shows that physical size (S_p) equal to 10 CWU and the function size (S_f) is equal to 6 LOC. Therefore, from equation 5.1 we can calculate the coding efficiency which will be 0.6 CWU/LOC. Figure 5.1 demonstrates that the trends of both physical size and

cognitive functional size follow basically the same pattern. As the physical size increases, so does the corresponding cognitive functional size. It is noteworthy that there are four points for which the cognitive functional size goes up sharply.

Table 5.2: Analysis of the physical and functional sizes

Example*	LOC with blank	LOC without blank (S_p)	Number of inputs (N_i)	Number of outputs (N_o)	Cognitive weight (W_c)	Cognitive functional size (S_f)
P 1	21	10	4	2	1	6
P 2	20	7	2	2	1	4
P 3	23	8	0	6	1	6
P 4	27	12	2	1	7	21
P 5	37	20	5	2	11	77
P 6	18	6	1	2	1	3
P 7	12	3	0	1	1	1
P 8	58	34	7	2	21	189
P 9	22	9	1	1	6	12
P 10	25	10	2	1	6	18
P 11	26	12	0	10	1	10
P 12	19	6	0	2	1	2
P 13	27	14	1	6	1	7
P 14	51	26	1	12	1	13
P 15	28	7	0	2	1	2
P 16	40	19	1	10	1	11
P 17	38	18	0	11	1	11
P 18	40	16	2	6	1	8
P 19	29	13	0	5	1	5
P 20	23	11	2	1	6	18
P 21	24	12	3	4	1	7
P 22	33	17	4	3	6	42
P 23	23	12	3	2	6	30
P 24	22	8	1	3	6	24
P 25	19	7	1	1	6	12
P 26	27	13	2	2	11	44
P 27	67	41	2	16	29	522
P 28	70	41	2	16	29	522
P 29	28	13	2	3	14	70
P 30	18	7	1	1	6	12

* The source of the programs in the Appendix A

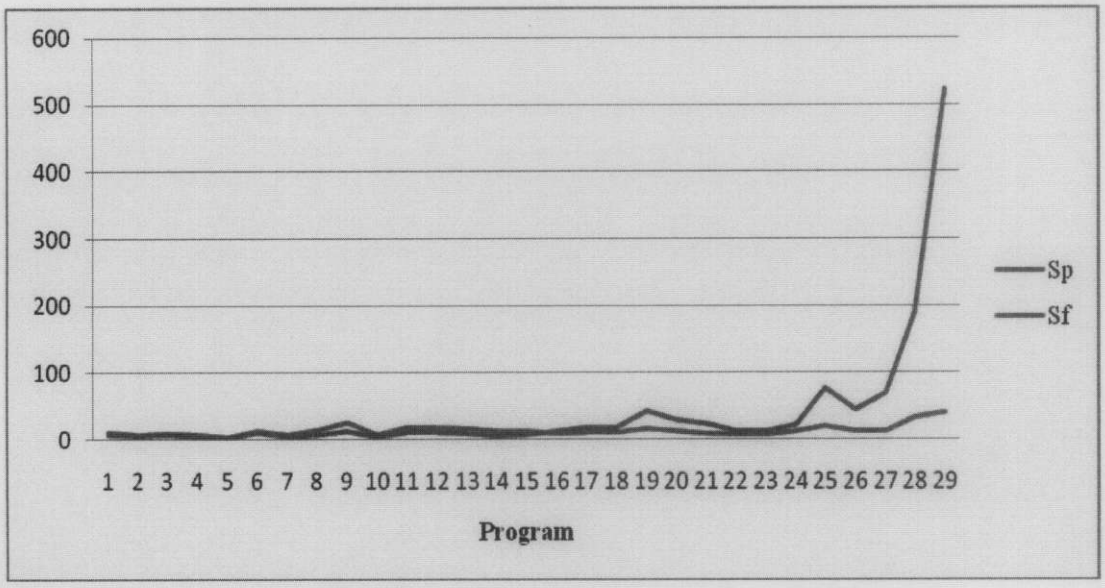


Figure 5.1: Plot of S_f and S_p of 30 sample programs.

As indicated in Figure 5.2, the physical size (S_f) cannot measure or predict the software complexity. The cognitive complexity of software is not proportional to its physical size. For example, there are three cases with similar physical size around 13 LOC, but one has cognitive functional size 5 CWU, the second one has cognitive functional size 44 CWU while that of the third is 70 CWU.

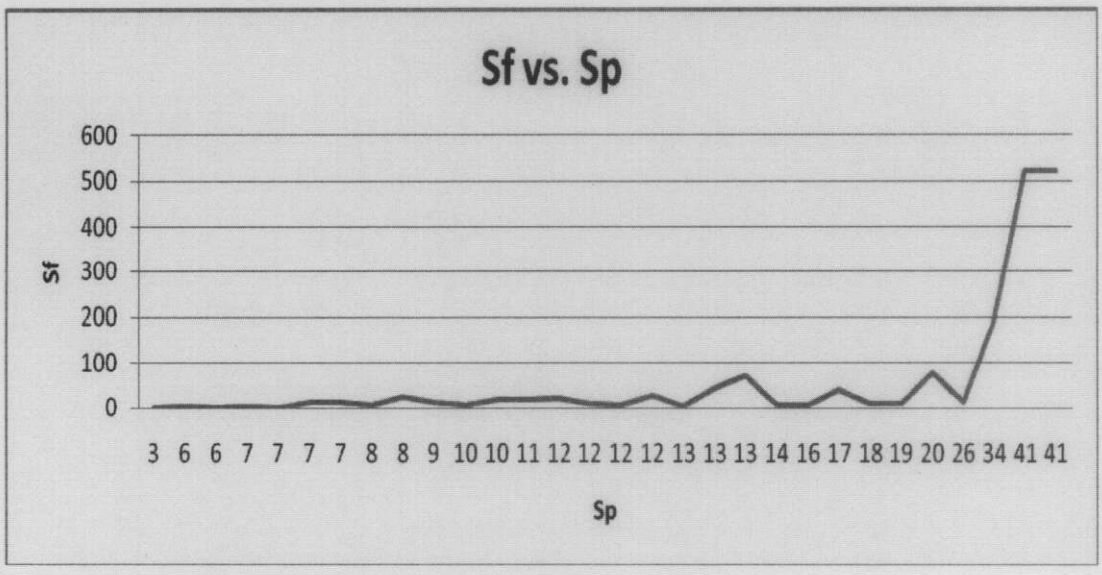


Figure 5.2: Plot physical size against functional size.

5.5 Conclusion

Software complexity measures serve both as an analyzer and a predictor in quantitative software engineering. The cognitive functional size (CFS) is used on the basis of cognitive weights, permitting determination of software complexity from both architectural and cognitive aspects. Cognitive weights for basic control structures (BCSs) have been introduced to measure the complexity of logical structures of software. A set of examples has been carried out to analyze the relationship between physical size and cognitive functional size of software. The cognitive functional size has been shown to be a fundamental measure of software artifacts based on the cognitive weight. This work has produced three substantial findings:

a) The CFS of software in design and comprehension is dependent on three factors: internal processing structures, as well as the number of inputs and outputs.
b) The cognitive complexity measure (CFS) is more robust than the physical-size measure and independent of language/implementation.
c) CFS provides a foundation for cross-platform analysis of complexity and size of both software specifications and implementations for either design or comprehension purposes in software engineering.

العنوان:	Measuring software complexity for software engineering quality assurance
المؤلف الرئيسي:	Wohaishi, Ahmad Muhammad Ali
مؤلفين آخرين:	Fahmy, Maged A.، Al Sultanny, Yas A.(Co-Advisor، Advisor)
التاريخ الميلادي:	2009
موقع:	المنامة
الصفحات:	1 - 155
رقم MD:	736039
نوع المحتوى:	رسائل جامعية
اللغة:	English
الدرجة العلمية:	رسالة ماجستير
الجامعة:	جامعة الخليج العربي
الكلية:	كلية الدراسات العليا
الدولة:	البحرين
قواعد المعلومات:	Dissertations
مواضيع:	برامج الحاسوب، هندسة البرمجيات، تكنولوجيا المعلومات
رابط:	https://search.mandumah.com/Record/736039

Chapter 6

Software Production Complexity Model

6.1 Introduction

Each software system and each software project is unique. Modeling software quality or productivity therefore has to be product or project specific. The goal of this Chapter is to propose a model of software production complexity. Within the discussion of the sources, the nature and the effects of complexity assist to laid down the theoretical framework for a graphical disclosure of the understanding of the software complexity concept as shown in Figure 6.1. The relationships between sources and activities have already been explained in this chapter. The sources of complexity have been discussed in the beginning of this chapter. The problem gives the project its inherent complexity, and further complexity is added during the design and code phase of the software development process. This model is not claiming and encompassing all thinkable factors, but examples of components that are settling the occurrence of errors in the software and the size of the project are given. The management is interesting because a project can suffer as much from a defective leadership as it can benefit from a good one. Our belief is that a manager who has the ability to direct a project also creates better prerequisites for a low level of error density.

The environment in itself, e.g. the working place, the colleagues, the workload etc., is also a determinant factor for the error proneness of the system. A "negative" environment usually means small possibilities of creating high-quality software. The use of good and relevant working methods is also important for the ability to reduce the number of errors in a program, as well as the competence, i.e. the experience and knowledge, of the people working with the project. The proposed general model of software production complexity is as shown in Figure 6.1 and will be discussed in the next sections.

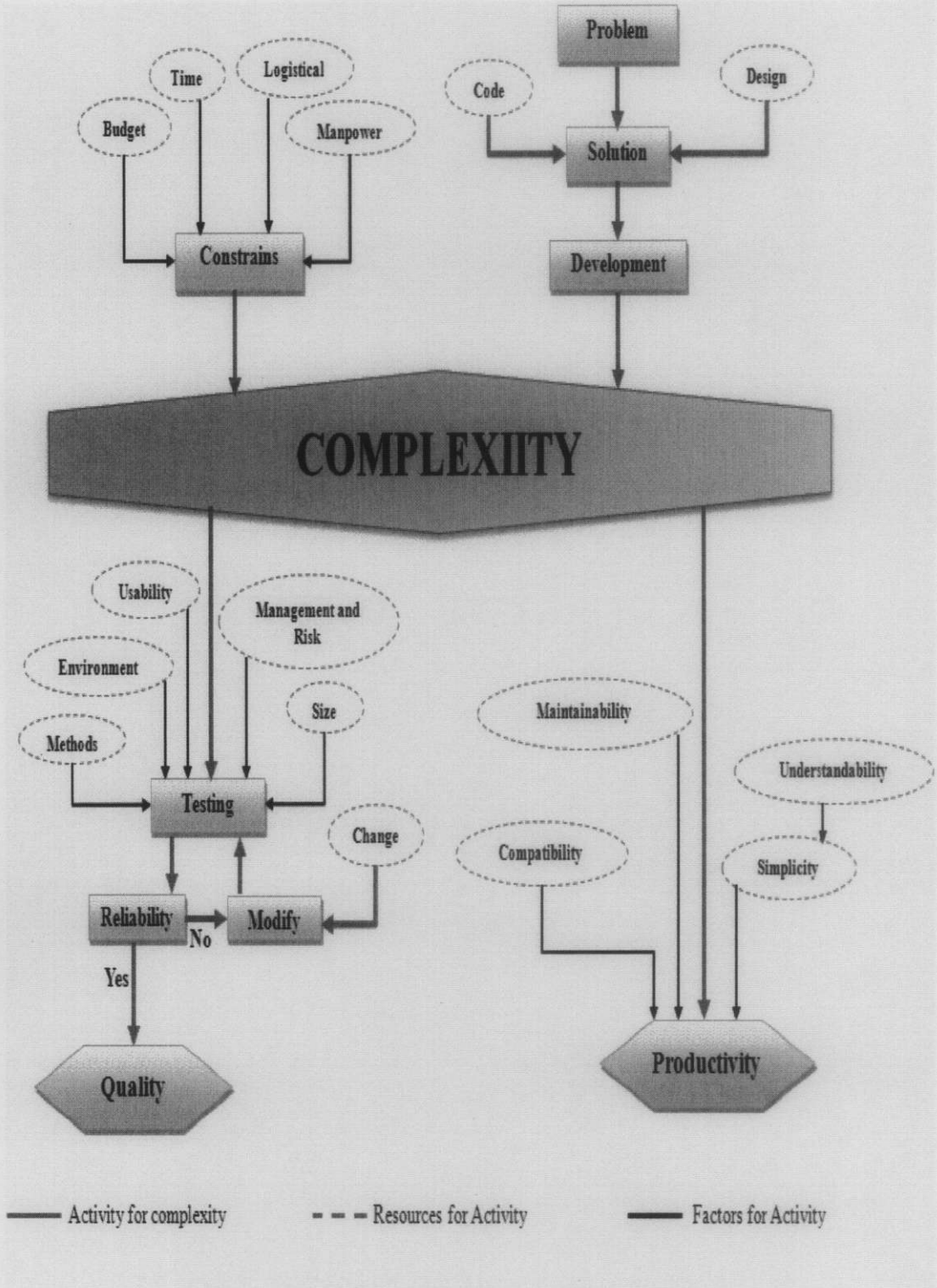


Figure 6.1: A model of software production complexity

6.2 Problem with Software Complexity Model

Complexity of the problem which is the inherent complexity, created during the requirements phase. Complexity of the solution (also referred to as added complexity), is the complexity being attached to the complexity of the problem. This type of complexity is added during the development stages following the requirements phase, primarily during the designing and coding phases. Another way to look at software complexity is to see it as the resources needed for the project, or the efficiency of the project. With this approach, the complexity of a problem can be defined as the amount of resources required for an optimal solution of the problem.

6.2.1 Solution

Then complexity of a solution can be regarded in terms of the resources needed to implement a particular solution.

i. Design:

While requirements are meant to specify what a program should do, design as shown in Figure 6.1 is meant, at least at a high level, to specify how the program should do it. The usefulness of design is also questioned by some, but those who look to formalize the process of ensuring reliability often offer good software design processes as the most significant means to accomplish it. Software design usually involves the use of more abstract and general means of specifying the parts of the software and what they do. As such, it can be seen as a way to break a large program down into many smaller programs, such that those smaller pieces together do the work of the whole program. The purposes of high-level design are as follows. It separates what are considered to be problems of architecture, or overall program concept and structure, from problems of actual coding, which solve problems of actual data processing.

ii. Code

To a computer, there is no real concept of "well-written" source code. However, to a human, the way a program is written can have some important consequences for the human maintainers. Many source code programming style guides, which stress readability and some language-specific conventions are aimed at the maintenance of the software source code, which involves debugging and updating. Other issues also come into considering whether code is well written, such as the logical structuring of the code into more manageable sections.

6.2.2 Development

Software development is the translation of a user need or marketing goal into a software product. Software development is sometimes understood to encompass the processes of software engineering combined with the research and goals of software marketing to develop computer software products. This is in contrast to marketing software, which may or may not involve new product development. It is often difficult to isolate whether engineering or marketing is more responsible for the success or failure of a software product to satisfy customer expectations. This is why it is important to understand both processes and facilitate collaboration between both engineering and marketing in the total software development process.

6.3 Constrains on Software Complexity Model

These resources have at least two aspects: time, i.e. computer time and man-hours, and space, i.e. computer memory. On other hand, this will help us to classified which constrains either budget, time, logistical or manpower need to be analyze to find which aspects will have an impact to the program complexity. The advantage of this approach is also that by concentrating on how to measure complexity at different phases of the development, we can create models for predicting how much complexity will be added to the project later in the process.

i. Budget:

Every project has a certain budget allocated to it. The money allocated for each project covers the entire scope of work (complexity) which includes design, development until final completion of the project. The budget must also take into account any unexpected contingencies. An efficient project is one which will work within the budget allocated. Any cost-overrun will make the project unprofitable and unviable.

ii. Time:

Any project has to be completed within a given time. Only then the project becomes viable and hence profitable. If there is delay in completing the project due to unforeseen circumstances, it leads to various other complications and sometimes even liquidated damages from the client. Hence, within the given time, the project has to be completed.

iii. Logistic:

For carrying out a project, proper logistics have to be planned well in advance. For example, this could be constraints in transport, infrastructure, UPS power when power failure occurs, and so many such issues external to, but which aid in making the program a success.

iv. Manpower:

If adequately trained manpower are not available or leave half-way in a project, it could create a big problem. Hence having enough backup of trained manpower is absolutely essential for the success of any project.

6.4 Software Quality

In the case of most software-based system, the achievable software quality goals largely depend on the availability of resources for performing software quality improvement activities. As a general rule, the more resources the project can devote towards testing and software quality assurance, the better the software quality of the end product. Due to the commonly occurring scenario of limited availability of project resources, a software

development team is often faced with task of providing the best software quality within the means of the given resources.

The transcendental view, that sees quality as something that can be recognized, but not defined; the user view, which sees quality as fitness for the user's purpose; the manufacturers view, which sees quality as conformance to specification; the product view, which sees quality as tied to inherent characteristics of the product; and the value-based view, which sees quality as dependent on what a customer is willing to pay for it. Thus the view of software quality is pragmatic and relatively simple - high quality software is software that "works well enough" to serve its intended function and is software that is "available when needed" to perform that function.

6.4.1 Testing

Software testing is the process of checking software, to verify that it satisfies its requirements and to detect errors. Software testing is an empirical investigation conducted to provide stakeholders with information about the quality of the product or service under test, with respect to the context in which it is intended to operate. This includes, but is not limited to, the process of executing a program or application with the intent of finding software bugs. Testing can never completely establish the correctness of computer software. Instead, it furnishes a criticism or comparison that compares the state and behavior of the product against a specification. Over its existence, computer software has continued to grow in complexity and size. Every software product has a target audience. Therefore, when an organization develops or otherwise invests in a software product, it presumably must assess whether the software product will be acceptable to its end users, its target audience, its purchasers, and other stakeholders. The following issues are the main factors to be taken in consideration during the software testing:

i. Size:

One of the oldest and most common forms of software measurement is size. There are many possibilities for representing size. Types of line counts include lines of code, non-comment non-blank source lines of code, and executable statements. Line of Code (LOC) -

A line of code is any line of program test, regardless of the purpose or number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements. All other forms of line counts are a subset of this count. Non-comment Non-blank (Source lines of code - SLOC) is line of code that is not a comment line or a blank line. General industry standards suggest that 50 to 100 lines is the maximum size that any module should attain; larger modules tend to be difficult to understand and thus lower in maintainability and reusability.

ii. Methods

Software testing methods are traditionally divided into black box testing and white box testing. These two approaches are used to describe the point of view that a test engineer takes when designing test cases. Black box testing treats the software as a black-box without any understanding of internal behavior. It aims to test the functionality according to the requirements. Thus, the tester inputs data and only sees the output from the test object. This level of testing usually requires thorough test cases to be provided to the tester who then can simply verify that for a given input, the output value (or behavior), is the same as the expected value specified in the test case. Black box testing methods include: equivalence partitioning, boundary value analysis, all-pairs testing, fuzz testing, model-based testing, traceability matrix etc. White box testing, however, is when the tester has access to the internal data structures and algorithms and the code that implement these.

iii. Usability

Usability testing is a technique used to evaluate a product by testing it on users. This can be seen as an irreplaceable usability practice, since it gives direct input on how real users use the system. This is in contrast with usability inspection methods where experts use different methods to evaluate a user interface without involving users. Usability testing focuses on measuring a human-made product's capacity to meet its intended purpose. Examples of products that commonly benefit from usability testing are web sites or web applications, computer interfaces, documents, or devices. Usability testing measures the usability, or

ease of use, of a specific object or set of objects, whereas general human-computer interaction studies attempt to formulate universal principles.

iv. Environment

The ultimate situation would be a collection of test environments that mimic exactly all possible hardware, software, network, data, and usage characteristics of the expected live environments in which the software will be used. For many software applications, this would involve a nearly infinite number of variations, and would clearly be impossible. For new software applications, it may also be impossible to predict all the variations in environments in which the application will run. For very large, complex systems, duplication of a 'live' type of environment may be prohibitively expensive. In reality judgments must be made as to which characteristics of a software application environment are important, and test environments can be selected on that basis after taking into account time, budget, and logistical constraints. Such judgments are preferably made by those who have the most appropriate technical knowledge and experience, along with an understanding of risks and constraints. For smaller or low risk projects, an informal approach is common, but for larger or higher risk projects (in terms of money, property, or lives) a more formalized process involving multiple personnel and significant effort and expense may be appropriate.

v. Management and Risk:

Risk is defined as the possibility of loss, and the degree of probability of such a loss. Barry Boehm 1989, in his book "Software Risk Management", defines risk management as a combination of risk assessment and risk control. Risk assessment includes identification, analysis, and prioritization; risk control includes management, planning, resolution and monitoring. The project risk management process may be summarized as identification of risks, ranking and prioritization of risks, and monitoring risks throughout their applicable life. In order to develop a software quality model that is useful in the risk management process, it is necessary to identify a set of risks that is common to most projects and based on the project manager's idea of software quality. Given that definition, the risk areas are:

- Correctness
- Reliability
- Maintainability
- Reusability
- Schedule

The project manager might have a different prioritization of the listed risks for different segments of software that are being developed by the project.

6.4.2 Reliability

Reliability is the capability of the software product to maintain a specified level of performance when used under specified conditions. Software Reliability is the probability of failure-free software operation for a specified period of time in a specified environment. Software Reliability is also an important factor affecting system reliability. It differs from hardware reliability in that it reflects the design perfection, rather than manufacturing perfection. The high complexity of software is the major contributing factor of Software Reliability problems. Software Reliability is not a function of time - although researchers have come up with models relating the two. The modelling technique for Software Reliability is reaching its prosperity, but before using the technique, we must carefully select the appropriate model that can best suit our case. Hardware reliability is often defined in terms of the Mean-Time-To-Failure, or MTTF, of a given set of equipment.

6.4.3 Modify or Change

Software maintainability is defined as the ease of finding and correcting errors in the software. It is analogous to the hardware quality of Mean-Time-To-Repair, or MTTR. While there is as yet no way to directly measure or predict software maintainability, there is a significant body of knowledge about software attributes that make software easier to maintain. These include modularity, self (internal) documentation, code readability, and structured coding techniques. These same attributes also improve sustainability, the ability to make improvements to the software. Maintainability is the capability of the software product to be modified. Modifications may include corrections, improvements, or

adaptation of the software to changes in environment, and in requirements and functional specifications.

6.5 Software Productivity

Software productivity is the capability of the software product to enable users to expend appropriate amounts of resources in relation to the effectiveness achieved in a specified context of use. The interest of a productivity measure is to appreciate the efficiency of a production process and to determine how to improve it. The issue is to find when the minimal cost is reached to realize the optimum profit. But the cost function is project-specific and depends on the kind of produced assets. For instance, the initial cost in building a model refinement framework is obviously different from an Human-Machine Interaction (HMI) framework; a learning team is less productive than an experimented one.

6.5.1 Maintainability

Software complexity may have a direct impact upon maintenance costs as well as the costs incurred through the presences of software errors. Software requirements must be derived from system requirements and be maintainable to the system requirements to assure that the software developed is going to work properly in the system setting. The software requirements must also be traceable to the implementing design and code, to ensure that they are in fact part of the software. They must be easy to maintain to tests to ensure that they have been validated and verified. There are two metrics for maintainability, the number of requirements not maintain to higher level requirements and the number not maintain to code and tests. To compute these metrics requires a trace matrix which is to contain the maintenance information. A tool to help with upward maintenance is not yet available, so only downward maintenance is now done.

6.5.2 Simplicity (Understandability)

The attribute of simplicity relates to the ability of the developers to understand clearly what is met by the requirements document. The programmer is currently looking for good ways to automatically evaluate understandability. At this time we are looking at two metrics, one

based on numbering structure and a second based on readability evaluations. The width and breadth of the document's numbering structure provides an indication of extent that the requirements have been organized and the amount of detail provided.

6.5.3 Compatibility

Compatibility of a software is defined as minimum hardware resources required to perform its functions. There are many other software qualities. Some of them will not be important to a specific software system, thus no activities will be performed to assess or improve them. Maximizing some qualities may cause others to be decreased. For example, increasing the compatibility of a piece of software may require writing parts of it in assembly language. This will decrease the transportability and maintainability of the software.

6.6 Conclusion

The concept of complexity is very hard to define, and it is even more so when it comes to software complexity. We proposed the concept of software complexity which is the degree of difficulty in analyzing, maintaining, testing, designing and modifying software, i.e. software complexity is a subject during the entire software development process. When we established this definition, we turned to the sources of complexity. They were divided in two main classes: complexity of the problem, and the constrains that might affect complexity. The last-mentioned was also divided into complexity generated during the design, and code phases of the solution development.

We are introduced the proposed software complexity model which shows the relation between the problem and constrains that might be affect the quality and productivity of the software. We study the issues and resources that effect the testing such as usability, reliability, management and risk which will help us to improve the quality to develop the software complexity. Also, we study the compatibility, maintainability and understandability that effecting our proposed model which will influence the software productivity. To explore the nature of complexity, we can classify the concept, in order to look at it from different viewpoints. The effects of complexity were divided in two main

parts: error-proneness and size. A program that is more complex than another is also more likely to contain more errors and generate a larger system. The error-proneness, in its turn, influences such attributes of the system as the usability, reliability and need of change. Size, on the other hand, influences the maintainability, understandability and computational power needed for implementing the system.

العنوان:	Measuring software complexity for software engineering quality assurance
المؤلف الرئيسي:	Wohaishi, Ahmad Muhammad Ali
مؤلفين آخرين:	Fahmy, Maged A.، Al Sultanny, Yas A.(Co-Advisor، Advisor)
التاريخ الميلادي:	2009
موقع:	المنامة
الصفحات:	1 - 155
رقم MD:	736039
نوع المحتوى:	رسائل جامعية
اللغة:	English
الدرجة العلمية:	رسالة ماجستير
الجامعة:	جامعة الخليج العربي
الكلية:	كلية الدراسات العليا
الدولة:	البحرين
قواعد المعلومات:	Dissertations
مواضيع:	برامج الحاسوب، هندسة البرمجيات، تكنولوجيا المعلومات
رابط:	https://search.mandumah.com/Record/736039

الخلاصة

المقاييس هي التي نستخدمها من خلال الصناعة البرمجية لتحديد التطور والتشغيل والصيانة في البرمجيات. إن ممارسة تطبيق مقاييس البرامج في عمليات ومنتجات البرمجيات هي عملية معقدة و تتطلب الدراسة والانضباط وهي تساعدنا في تقديم المعرفة اللازمة عن وضع العمليات أو المنتج فيما يخص البرمجيات والأهداف التي نريد الوصول إليها.

في هذه الرسالة تطرقنا إلى طرق حساب التعقيد للبرامج مثل طريقة مكاب وطريقة هالستيد ومن خلال هذه الدراسة قمنا بتطبيق هذه الطرق القياسية على عينة من الأمثلة وقمنا بتقييم نتائجها. وقمنا بتسليط الضوء على بعض المفاهيم الخاطئة والكامنة وراء هذا النهج والقياس واشرنا كذلك إلى ضرورة فهم التعاريف والنماذج لطرق القياس وممارسة تطبيقها في الجودة النوعية للبرامج التي سوف تزداد بشكل كبير.

إن التعقيد في البرامج هو مفهوم شامل يشير إلى العوامل التي تقرر مستوى الصعوبة في عملية المشاريع البرمجية. ضمان الجودة من خلال القياسات ونماذج الجودة النوعية في النماذج المفهومة والتصاميم للبرامج والبيانات أو قواعد المعرفة دائما ما تكون مصدر قلق كبير للبرامج والأنظمة ومطوري قواعد البيانات.

في رسالة الماجستير هذه قمنا بدراسة المفاهيم في تعقيد البرامج التي تؤثر على البرامج وقمنا بدراسة ومقارنة حساب التعقيد في مثال البحث الخطي والثنائي وذلك بما يتعلق من ناحية الإنتاجية والجودة في عدة لغات مثل C++ والجافا والفيجوال بيسك. دراستنا قادتنا إلى تقديم نموذج خاص بنا فيما يتعلق بالتعقيد. ووفقا لنموذجنا في التعقيد فان القياسات التي قدمناها ركزت على الحسابات الهندسية للتعقيد لتطوير الجودة النوعية والإنتاجية للبرامج.

بالإضافة إلى ذلك طرحنا طريقتنا لحساب التعقيد مع الأخذ بالاعتبار الوقت اللازم لتنفيذ العمليات من خلال عدة أنواع من المعالجات مثل البنتيوم. ولقد قدمنا من خلال هذه الأطروحة النموذج الخاص بنا الذي يبين العلاقة بين المشاكل والعوائق التي من الممكن أن يواجهها البرنامج لكي نستطيع تحليلها ومراعاتها من اجل أن يكون البرنامج ذو جودة وإنتاجية عالية.

العنوان:	Measuring software complexity for software engineering quality assurance
المؤلف الرئيسي:	Wohaishi, Ahmad Muhammad Ali
مؤلفين آخرين:	Fahmy, Maged A.، Al Sultanny, Yas A.(Co-Advisor، Advisor)
التاريخ الميلادي:	2009
موقع:	المنامة
الصفحات:	1 - 155
رقم MD:	736039
نوع المحتوى:	رسائل جامعية
اللغة:	English
الدرجة العلمية:	رسالة ماجستير
الجامعة:	جامعة الخليج العربي
الكلية:	كلية الدراسات العليا
الدولة:	البحرين
قواعد المعلومات:	Dissertations
مواضيع:	برامج الحاسوب، هندسة البرمجيات، تكنولوجيا المعلومات
رابط:	https://search.mandumah.com/Record/736039

Index

Abstract i
Acknowledgements ii
Dedication iii
Index iv
List of Symbols and abbreviations vii
List of Figures ix
List of Tables xi
Chapter 1 Introduction and Overview 1
1.1 Software Engineering 1
1.2 Software Quality Assurance 5
1.3 Software Quality Control 7
1.4 Fundamental of Software Testing 12
1.5 Standards 14
1.6 Problem Statement 16
1.7 Research Significance 16
1.8 Research Objective 17
1.9 Research Methodology 17
1.10 Research Variables 18
1.11 Thesis outline 18
Chapter 2 Literature Survey on Software Complexity 20
2.1 Structure measures of software complexity 20
2.2 Line of Code Method 23
2.3 Information Flow 25
2.4 McCabe Method 28
2.5 Halstead's Method 30
2.6 Cognitive Weights Method 32
Chapter 3 Implementation of Measuring Object-Oriented Programming Complexity 35
3.1 Introduction 35
3.2 Most Common Used Programming Languages and History 36

3.2.1 C and C# Language 37
3.2.2 Visual Basic, Early Versions 37
3.2.3 HTML Language 38
3.2.4 JavaScript 38
3.2.5 VBScript 38
3.2.6 Object-Oriented Programming 39
3.2.7 Comparison Between C# and Java Keywords 43
3.3 Examples of Measuring Complexity for Software Program 44
3.3.1 Case Study I (Linear Search Complexity) 44
3.3.2 Case Study II (Binary Search Complexity) 57
3.4 Conclusion 71
Chapter 4 Modification of Halstead Mathematical Equation Complexity 72
4.1 Combinational Logic Design 73
4.1.1 Binary Adders 73
4.1.2 Full Adder 74
4.2 Effect of Microprocessors on Halstead method 76
4.3 Conclusion 92
Chapter 5 Software Complexity Based on Cognitive Weights 93
5.1 Introduction 93
5.2 The Cognitive Weight of Software 96
5.3 Formal Description of Software Cognitive Complexity 97
5.4 Robustness of the Cognitive Complexity Measure 98
5.5 Conclusion 101
Chapter 6 Software Production Complexity Model 102
6.1 Introduction 102
6.2 Problem with Software Complexity Model 104
6.2.1 Solution 104
6.2.2 Development 105
6.3 Constrains on Software Complexity Model 105
6.4 Software Quality 106

6.4.1 Testing 107
6.4.2 Reliability 110
6.4.3 Modify or change 110
6.5 Software Productivity 111
6.5.1 Maintainability 111
6.5.2 Simplicity 111
6.5.3 Compatibility 112
6.6 Conclusion 112
Chapter 7 Conclusion and Future Works 114
7.1 Conclusion 114
7.2 Future Works 115
References 116
Appendix 121

العنوان:	Measuring software complexity for software engineering quality assurance
المؤلف الرئيسي:	Wohaishi, Ahmad Muhammad Ali
مؤلفين آخرين:	Fahmy, Maged A.، Al Sultanny, Yas A.(Co-Advisor، Advisor)
التاريخ الميلادي:	2009
موقع:	المنامة
الصفحات:	1 - 155
رقم MD:	736039
نوع المحتوى:	رسائل جامعية
اللغة:	English
الدرجة العلمية:	رسالة ماجستير
الجامعة:	جامعة الخليج العربي
الكلية:	كلية الدراسات العليا
الدولة:	البحرين
قواعد المعلومات:	Dissertations
مواضيع:	برامج الحاسوب، هندسة البرمجيات، تكنولوجيا المعلومات
رابط:	https://search.mandumah.com/Record/736039

ARABIAN GULF UNIVERSITY

College of Graduate Studies

Technology Management Programme

Measuring Software Complexity for Software Engineering Quality Assurance

A Thesis Submitted in Partial Fulfillment of the Requirements for the Master's Degree in Technology Management
(Specializing in Engineering)

Submitted by

Ahmed Mohammed Ali Wohaishi

Bachelor of Electrical Engineering, King Fahad University of Petroleum and Minerals, Kingdom of Saudi Arabia, 1999

Supervised by

Dr. Yas A. Al-Sultanny
Associate Professor of Technology Management
Arabian Gulf University

Dr. Maged M. Fahmy
Assistant Professor of Computers
King Faisal University
Kingdom of Saudi Arabia

KINGDOM OF BAHRAIN

January 2009 (A.D.)

Muharram 1430 (A.H.)

أعضاء لجنة المناقشة والحكم

الاســـــــــــــــــم	التوقيـــع
ممتحن خارجي **د. علي أظهر خان** أستاذ مشارك قسم علم الحاسوب – جامعة البحرين	
ممتحن داخلي **د. حافظ إبراهيم المطوع** أستاذ مساعد برنامج إدارة التقنية – جامعة الخليج العربي	
مشرف رئيس **د. ياس السلطاني** استاذ مشارك –برنامج إدارة التقنية جامعة الخليج العربي	
مشرف مشارك **د. ماجد محمد محمود فهمي** أستاذ مساعد قسم علم الحاسوب جامعة الملك فيصل المملكة العربية السعودية	

BOX: 26671, KINGDOM OF BAHRAIN, TEL.: 17239999, FAX: 17272815

Acknowledgements

It is a great honor for me to have studied at the College of Postgraduate Studies at the Arabian Gulf University, where innovation and constant improvement are the order of the day. There are a number of people without whom this thesis might not have been written, and to whom I am greatly indebted.

I am deeply indebted to Dr. Yas Al-Sultanny and Dr. Maged Fahme for having been so supportive and for having given me all their help in the making of this thesis. I also express my gratitude for all my friends without whom this thesis might not have been successfully completed.

Dedication

This thesis is dedicated to my wonderful parents, my mother Alya'a and my father Mohammed, who have raised me to be the person I am today. They have been with me every step of the way, through good times and bad. Thank you for all the unconditional love, guidance, and support that you have always given me, helping me to succeed and instilling in me the confidence that I am capable of doing anything I put my mind to.

Thank you for everything. I love you!

Index

Abstract i
Acknowledgements ii
Dedication iii
Index iv
List of Symbols and abbreviations vii
List of Figures ix
List of Tables xi
Chapter 1 Introduction and Overview 1
1.1 Software Engineering 1
1.2 Software Quality Assurance 5
1.3 Software Quality Control 7
1.4 Fundamental of Software Testing 12
1.5 Standards 14
1.6 Problem Statement 16
1.7 Research Significance 16
1.8 Research Objective 17
1.9 Research Methodology 17
1.10 Research Variables 18
1.11 Thesis outline 18
Chapter 2 Literature Survey on Software Complexity 20
2.1 Structure measures of software complexity 20
2.2 Line of Code Method 23
2.3 Information Flow 25
2.4 McCabe Method 28
2.5 Halstead's Method 30
2.6 Cognitive Weights Method 32
Chapter 3 Implementation of Measuring Object-Oriented Programming Complexity 35
3.1 Introduction 35
3.2 Most Common Used Programming Languages and History 36

3.2.1 C and C# Language 37
3.2.2 Visual Basic, Early Versions 37
3.2.3 HTML Language 38
3.2.4 JavaScript 38
3.2.5 VBScript 38
3.2.6 Object-Oriented Programming 39
3.2.7 Comparison Between C# and Java Keywords 43
3.3 Examples of Measuring Complexity for Software Program 44
3.3.1 Case Study I (Linear Search Complexity) 44
3.3.2 Case Study II (Binary Search Complexity) 57
3.4 Conclusion 71
Chapter 4 Modification of Halstead Mathematical Equation Complexity 72
4.1 Combinational Logic Design 73
4.1.1 Binary Adders 73
4.1.2 Full Adder 74
4.2 Effect of Microprocessors on Halstead method 76
4.3 Conclusion 92
Chapter 5 Software Complexity Based on Cognitive Weights 93
5.1 Introduction 93
5.2 The Cognitive Weight of Software 96
5.3 Formal Description of Software Cognitive Complexity 97
5.4 Robustness of the Cognitive Complexity Measure 98
5.5 Conclusion 101
Chapter 6 Software Production Complexity Model 102
6.1 Introduction 102
6.2 Problem with Software Complexity Model 104
6.2.1 Solution 104
6.2.2 Development 105
6.3 Constrains on Software Complexity Model 105
6.4 Software Quality 106

6.4.1 Testing 107
6.4.2 Reliability 110
6.4.3 Modify or change 110
6.5 Software Productivity 111
6.5.1 Maintainability 111
6.5.2 Simplicity 111
6.5.3 Compatibility 112
6.6 Conclusion 112
Chapter 7 Conclusion and Future Works 114
7.1 Conclusion 114
7.2 Future Works 115
References 116
Appendix 121

List of Symbols and Abbreviations

μ_1	Number of unique operators
μ_2	Number of unique operands
μ	Vocabulary of program
BCPL	Basic combined programming language
BCS	Basic control structures
BPEL	Business process execution language
BPMs	Business process models
C	Structure complexity
CASE	Computer-Aided Software Engineering
CFS	Cognitive functional size
CLOC	Commented source line of code
CMM	Capability maturity model
CMT++	Code metric tool for C and C++
CWU	Cognitive weight unit of software
D	Difficulty of program
e	Number of edges
E	Effort of program
E_c	Coding efficiency
EXEC	Executable statements
FTP	File Transfer Protocol
GUIs	Graphical user interfaces
HTML	Hyper Text Markup Language
HTTP	Hypertext transfer protocol
IEEE	Institute of Electrical &Electronics Engineers
I/Os	Inputs / outputs
ISO	International Standard Organization
ISP	Internet service provider
LOC	Lines of code
LOCC	Line of code with comments

n	Number of nodes
N	Length of program
N_1	Total occurrences of operators
N_2	Total occurrences of operands
N_1^+	Total occurrences of operators for add
N_1^*	Total occurrences of operators for multiplication
$N_1^/$	Total occurrences of operators for division
N_+	Length add of program
N_*	Length multiplication of program
$N_/$	Length division of program
NCLOC	Noncommented source line of code
NTDS	Navel tactical data system
N/A	Not applicable
MC	McCabe cyclomatic complexity
MHz	Mega Hertz
OOP	Object-oriented programming
p	Number of connected components
QA	Quality assurance
RTPA	Real time process algebra
S_f	Functional size
S_p	Physical size
SPI	Software process improvement
SQA	Software quality assurance
SQE	Software quality engineering
SVV	Software verification and validation
V	The volume of program
UML	Unified modeling language
URL	Uniform resource locator
W_i & W_c	Cognitive weights

List of Figures

Figure 1.1 Contemporary View 2

Figure 1.2 A short history of software 3

Figure 1.3 The phases of the software life cycle/software process 4

Figure 1.4 Process based quality 7

Figure 1.5 Quality control process 8

Figure 1.6 ISO 9000 and quality management 15

Figure 2.1 A program and its associated flow graph 27

Figure 2.2 Example for McCabe's cyclomatic number 28

Figure 3.1 The relationship between Class, Superclass and Subclass 41

Figure 3.2 A pseudo code definition of a class 42

Figure 3.3 Main program- linear search flow chart written in C++ languages 46

Figure 3.4 Main program- linear search flow graph written in C++ languages 47

Figure 3.5 Main program- linear search flow chart written in VB languages 50

Figure 3.6 Main program- linear search flow graph written in VB languages 51

Figure 3.7 Main program- linear search flow chart written in Java languages 54

Figure 3.8 Main program- linear search flow graph written in Java languages 55

Figure 3.9 Comparison the complexity between the object oriented languages C++, Visual Basic, and Java for linear search algorithm 56

Figure 3.10 Main program- binary search flow chart written in C++ languages 59

Figure 3.11 Main program- binary search flow graph written in C++ languages 60

Figure 3.12 Main program- binary search flow chart written in VB languages 63

Figure 3.13 Main program- binary search flow graph written in VB languages 64

Figure 3.14 Main program- binary search flow chart written in Java languages 67

Figure 3.15 Main program- binary search flow graph written in Java languages 68

Figure 3.16 Comparison the complexity between the object oriented languages C++, Visual Basic, and Java for binary search algorithm 69

Figure 3.17 Comparison of McCabe vs. Halstead for linear search of an array 70

Figure 3.18 Comparison of McCabe vs. Halstead for binary search of an array 71

Figure 4.1 Binary addition 74
Figure 4.2 Truth table and logical maps of the full adder 75
Figure 4.3 Full adder implemented with half adders 76
Figure 4.4 Compare between the total number of operators and effort according to Halstead method 80
Figure 4.5 Compare between the total number of operators and efforts according to modified method to 80286 microprocessor..... 84
Figure 4.6 Compare between the total number of operators and efforts according to modified method to 80486 microprocessor..... 88
Figure 4.7 Compare between the total number of operators and efforts according to modified to Pentium microprocessor..... 92
Figure 5.1 Plot of S_f and S_p of 20 sample programs 100
Figure 5.2 Plot physical size against functional size 100
Figure 6.1 A model of software complexity 103

List of Tables

Table 2.1 Basic control structures and its cognitive weights 33

Table 3.1 Java and C# keywords 43

Table 3.2 Brief comparison between C# and Java languages keywords.....44

Table 3.3 The complexity of the main program linear search written by C++ 48

Table 3.4 The complexity of the main program linear search written by VB..... 52

Table 3.5 The complexity of the main program linear search written by Java 56

Table 3.6 The complexity of the main program binary search written by C++ 61

Table 3.7 The complexity of the main program binary search written by VB..... 65

Table 3.8 The complexity of the main program binary search written by Java 69

Table 3.9 Comparison of McCabe vs. Halstead methods for linear search of an array..... 70

Table 3.10 Comparison of McCabe vs. Halstead methods for Binary search of an array 70

Table 4.1 Operands by CMT++ 77

Table 4.2 Operators by CMT++ 77

Table 4.3 Examples of programs complexity by Halstead method 79

Table 4.4 Execution times for several generations of computers 80

Table 4.5 Examples according to modified method taken in consideration the execution time for add function to 80286 microprocessor 81

Table 4.6 Examples according to modified method taken in consideration the execution time for multiplication function to 80286 microprocessor 82

Table 4.7 Examples according to modified method taken in consideration the execution time for division function to 80286 microprocessor 83

Table 4.8 Examples according to modified method taken in consideration the execution time for add function to 80486 microprocessor 85

Table 4.9 Examples according to modified method taken in consideration the execution time for multiplication function to 80486 microprocessor 86

Table 4.10 Examples according to modified method taken in consideration the execution time for division function to 80486 microprocessor 87

Table 4.11 Examples according to modified method taken in consideration the execution time for add function to Pentium microprocessor 89

Table 4.12 Examples according to modified method taken in consideration the execution time for multiplication function to Pentium microprocessor 90

Table 4.13 Examples according to modified method taken in consideration the execution time for division function to Pentium microprocessor 91

Table 5.1 Definition of BCSs and their equivalent cognitive weights 93

Table 5.2 Analysis of the physical and functional sizes 99

Chapter 1

Introduction and Overview

As software systems importance has grown, the software community has continually attempted to develop technologies that will make it easier, and less expensive to build and maintain high-quality computer programs. Some of these technologies are targeted at a specific application domain (e.g., web-site design and implementation), other focus on a technology domain e.g. objected-oriented systems or aspect-oriented programming (Deitel H. et. al. 2009). This chapter presents an overview of the thesis, it describes the problem statement and continues with the thesis contribution. Then, it presents thesis outline.

1.1 Software Engineering

Several software systems have been developed over the past few years. However, the absence of a standard regulatory mechanism in terms of quality control/quality assurance with respect to implementation and managing projects, particularly in the industrial sector has lead to an inconsistency among the various software systems. The complexity of each project and uniqueness make the task even more difficult. With an eye on new methodologies and tools relevant to the entire life cycle, from conceptualization to implementation, the quality assurance of software has to be visualized. Development of software for managing projects is an extremely complex affair. Usually, the evolution of the software is the result of team work or rather several groups of specialists, who individually are experts in their respective disciplines, but probably may now have enough expertise in other disciplines. The work is quite tasking and time consuming. Whatever be the technique for final testing of the software, however organized be the methodology, however systematic be the documentation involved as well as control of the final configuration, it would be a fruitless exercise if proper management of quality assurance is not in place (Pressman R. 2005).

Software can also be seen as an interface between the problem domain and the computer as shown in Figure 1.1. Software Engineering, as defined in IEEE Standard 610.12, is: "The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software".

Virtually all countries now depend on complex computer-based systems (Somerville I. 2006). More and more products incorporate computers and controlling software in some form. The software in these systems represents a large and increasing proportion of the total system costs. Therefore, producing software in a cost-effective way is essential for the functioning of national and international economies software engineering is an engineering discipline whose goal is the cost effective development of software systems. In some ways, this simplifies software engineering as there are no physical limitations on the potential of software. In other ways, however, this lack of natural constraints means that software can easily become extremely complex and hence very difficult to understand (Nasib S. 2005).

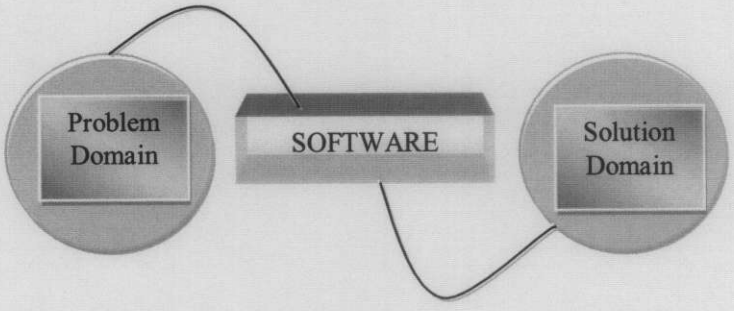


Figure 1.1: Contemporary View

Figure 1.2 shows short history of software. During the early years of the computer era, general-purpose hardware became commonplace. Product software (i.e., programs developed to be sold to one or more customers) was in its infancy. The second era of computer system spanned the decade from the mid-1960s to the late 1970s. Multiprogramming and multi-user systems introduced new concepts of human-machine interaction. Real-time systems could collect, analyze, and transform data from multiple sources. The second era was also characterized by the use of product software and the advent of "software houses." The third era of computer system evolution began in mid-1970s and spanned more than a full decade. The distributed system greatly increased the complexity of computer-based systems. The conclusion of third era was characterized by

the advent and widespread use of microprocessors. In less than a decade, computers became readily accessible to the public at large. The fourth era of computer system evolution moves us away from individual computers and computer programs and toward the collective impact of computers and software. Powerful desk-top machines controlled by sophisticated operating systems, networked locally and globally, and coupled with advanced software applications have become the norm. The Internet exploded and changed the way of life and business (Pressman R. 2005).

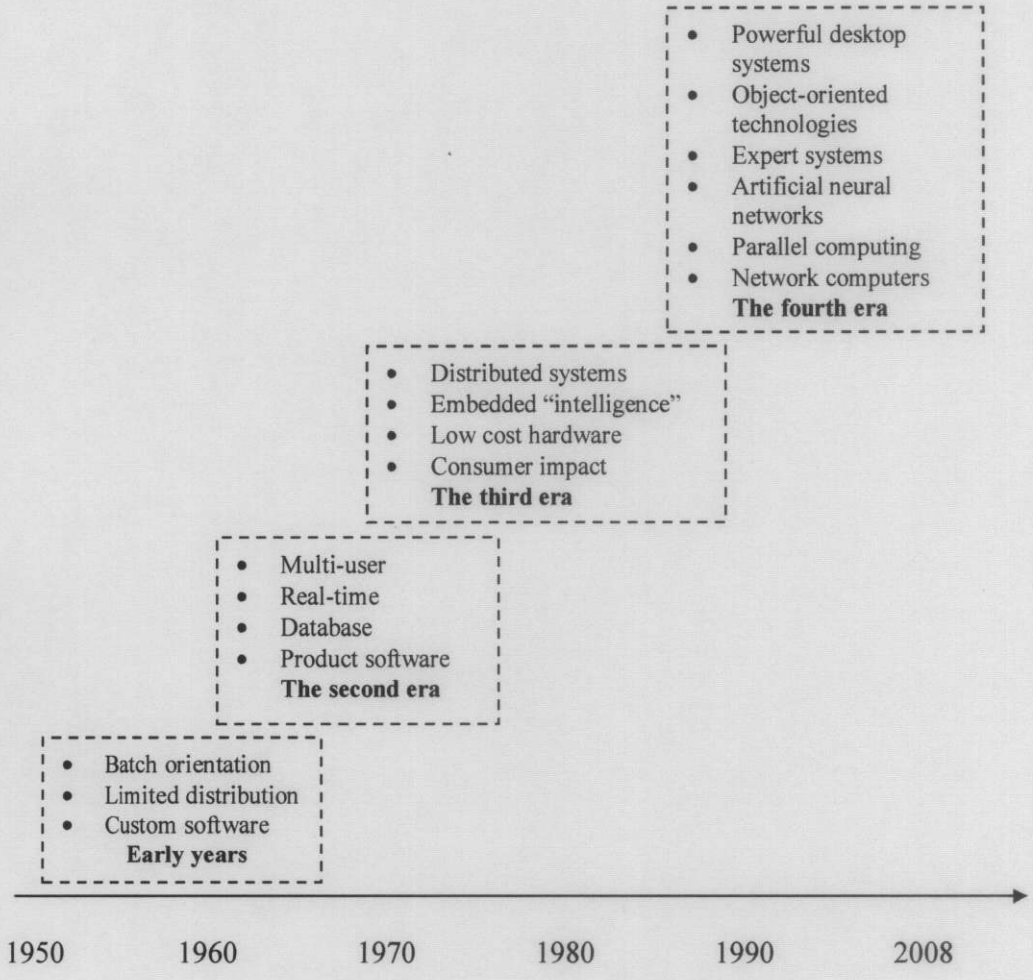


Figure 1.2: A short history of software

The series of steps that software undergoes, from concept exploration through final retirement, is termed its life cycle as shown in Figure 1.3. During this time, the product

goes through a series of phases: requirements, specification (analysis), planning, design, implementation, integration, maintenance (which is the highest cost among all these phases), and retirement (Schach S. 1997).

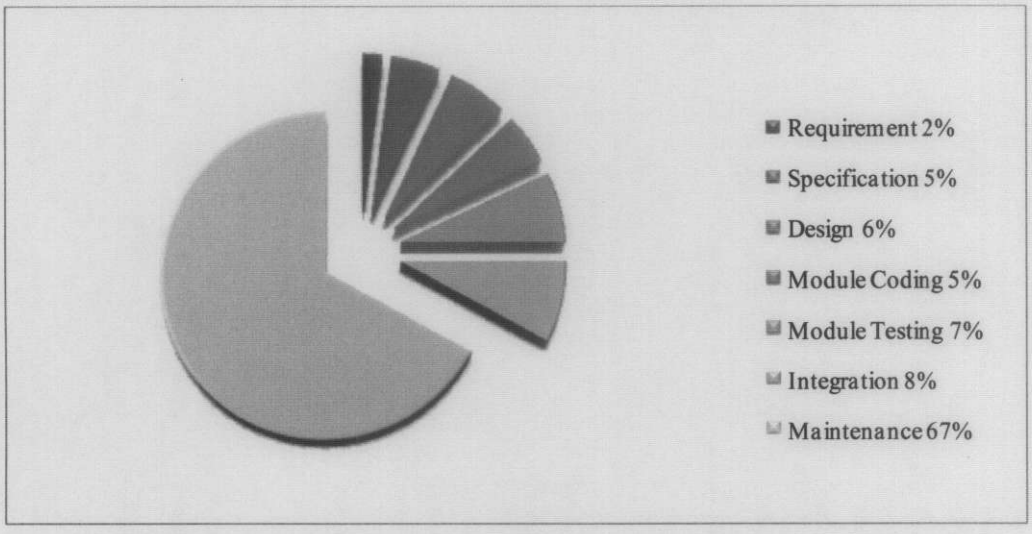


Figure 1.3: The phases of the software life cycle/software process

1. *Requirements phase*: The concept is explored and refined, and the client's requirements are elicited.
2. *Specifications (analysis) phase*: The client's requirements are analyzed and presented in the form of the specification document ("what the product is supposed to do"). Sometimes it is called the specification phase since a plan is drawn up for the software project management and the proposed development is described in detail.
3. *Design phase*: The specifications undergo two consecutive processes of architectural design (the product as a whole is broken down into modules) and detailed design (each module is designed).
4. *Implementation phase*: The various components undergo coding and testing.
5. *Integration phase*: The components are combined and tested as a whole (integration) when the developers are satisfied that the product functions correctly,

it is tested by the client (acceptance testing) implementation phase ends when the product is accepted by the client and installed on the client's computer.

6. *Maintenance phase*: All changes to the product once the product has been delivered and installed on the client's computer. It includes corrective maintenance (software repair) which consists of the removal of residual faults while leaving the specifications unchanged and enhancements (software updates) which consists of changes to the specifications and the implementation of those changes. The two types of enhancements are perfective (changes the client thinks will improve the effectiveness of the product, such as additional functionality or decreased response time) and adaptive (changes made in response to changes in the environment, such as new hardware/operating system or new government regulations).
7. *Retirement phase*: The product is removed from service. Provided functionality is no longer of use to the client.

1.2 Software Quality Assurance

The quality of software has improved significantly over the last few years and the reason for this is that companies have new techniques and technology such as the use of object-oriented development and associated Computer-Aided Software Engineering (CASE) support. In addition, however, there has a greater awareness of the importance of software quality management and the adoption of quality management techniques from manufacturing in the software industry. However, software quality is a complex concept that is not directly comparable with quality in manufacturing. In manufacturing, the notion of quality means that the developed product should meet its specification. In an ideal world this definition should be applied to all products but for software system the problems with this is as the following (Moores B 1994):

- A specification should be oriented toward the characteristics of the product that the customer wants. However, the development organization may also have equipments (such as maintainability requirements) that are not included in the specification.
- Unknown how to specify certain quality characteristics (e.g., maintain ability) in an unambiguous way.

Quality assurance (QA) is the process to define how the software quality can be achieved and how the development organization knows that the software has the required level of quality. The QA process is primarily concerned with defining or selecting standards that should be applied to the software development process or software product. As the part of QA process tools and methods to support these standards are selected and procured. The two types of standards that may be established as part of the quality assurance process are (Sommerville I. 2006):

1. Product standards: these standards are applied to the software product being developed. These include document standards, such as the structure of requirements documents; documentation standards.
2 Process standards: these standards define the process that should be followed during software development. It include definitions of specification design and validation process and a description of the documents that should be written in the path of these processes.

SQA must plan what checks to do early in the project. The most important selection criterion for software quality assurance planning is risk. Common risk areas in software development are novelty, complexity, staff capability, staff experience, manual procedures and organizational maturity. SQA staff should concentrate on those items that have a strong influence on product quality. They should check as early as possible the following (Jones M. et. al. 1997):

- Project is properly organized, with an appropriate life cycle;
- development team members have defined tasks and responsibilities;
- documentation plans are implemented;
- documentation contains what it should contain;
- documentation and coding standards are followed;
- standards, practices and conventions are adhered to;
- metric data is collected and used to improve products and processes;
- reviews and audits take place and are properly conducted;
- tests are specified and rigorously carried out;

- problems are recorded and tracked;
- projects use appropriate tools, techniques and methods;
- software is stored in controlled libraries;
- software is stored safely and securely;
- software from external suppliers meets applicable standards;
- proper records are kept of all activities;
- staff are properly trained;
- risks to the project are minimized.

Project management is responsible for the organization of SQA activities, the definition of SQA roles and the allocation of staff to those roles (Jones M. et. al. 1997).

1.3 Software Quality Control

Good quality managers aim to develop a 'quality culture' where everyone responsible for product developments is committed to achieving a high level of product quality as shown in Figure 1.4. They encourage teams to take responsibility for the quality for their work and to develop new approaches to quality improvement, while standards and procedures are the basis of quality management, experienced quality managers recognize that there are intangible aspects to software quality (elegance, readability, etc.) that cannot be embodied in standards. They support people who are interested in these intangible aspects of quality and encourage professional behavior in all team members (Daniel G. 2003).

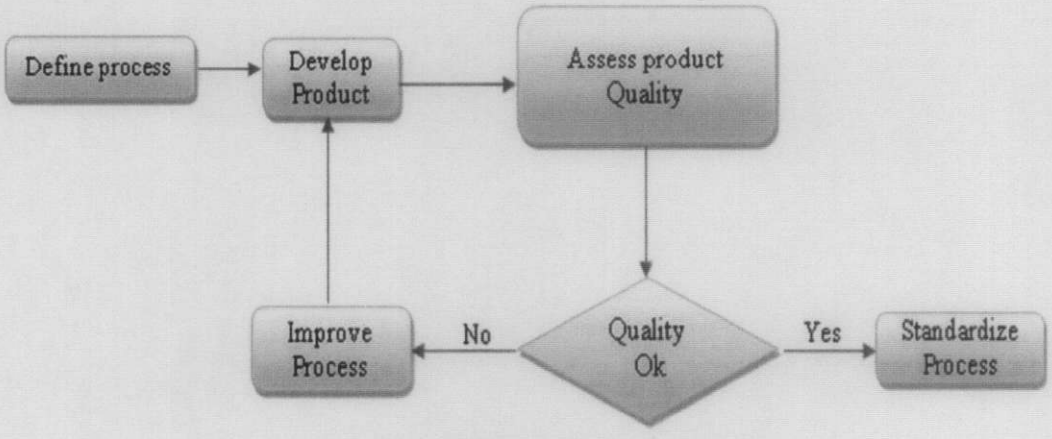


Figure 1.4: Process based quality

Variation control may be equated to quality control as shown in Figure 1.5. But how do we achieve quality control?. Quality control involves the series of inspection, reviews, and tests used throughout the software process to ensure each work product meets the requirements placed upon it. Quality control includes a feedback loop to the process that created the work product. The combination of measurement and feedback allows us to tune the process when the work products created fail to meet their specifications. A key concept of quality control is that all work products have defined, measurable specifications to which we may compare the output of each process. The feedback loop is essential to minimize the defects product (Vigder M. et. al. 1994).

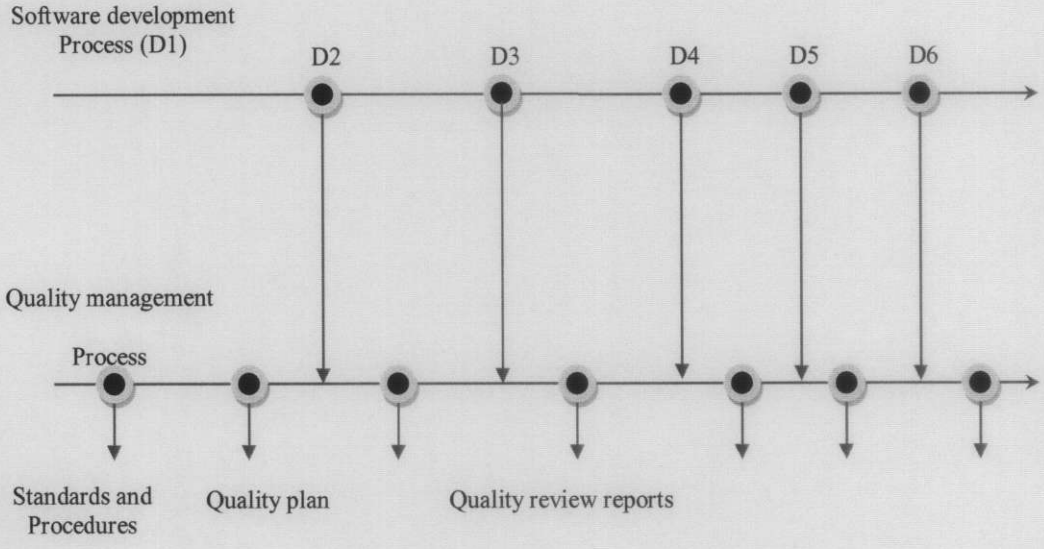


Figure 1.5: Quality control process

Testing presents an interesting anomaly for the software engineers, who by their nature are constructive people. Testing requires that the developer discard preconceived notions of the "correctness" of software just developed and then work hard to design test cases to "break" the software. Software testability is simply how easily can be tested. The following characteristics lead to testable software (Gray M. 1999):

Operability: "The better it works, the more efficiently it can be tested." If a system is designed and implemented with quality in mind, relatively few bugs will block the execution of tests, allowing testing to progress without fits and starts.

Observability: "What you see is what you test." Inputs provided as part of testing produce distinct outputs. System states and variable are visible during execution. Incorrect output is easily identified. Internal errors are automatically detected and reported. Source code is accessible.

Controllability: "The better we can control the software. The more the testing can be automated and optimized." software and hardware states and variables can be controlled directly by the test engineer. Tests can be conveniently specified automated, and reproduced.

Decomposability: "By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting." the software is built from independent modules that can be tested independently.

Simplicity: "The less there is to test, the more quickly we can test it." The program should exhibit functional simplicity (e.g., the feature set is the minimum necessary to meet requirements), structural simplicity (e.g., architecture is modularized to limit the propagation of faults), and code simplicity (e.g., a coding standard is adopted for ease of inspection and maintenance).

Stability: "The fewer the changes, the fewer the disruption to testing." changes to the software are infrequent, controlled when they do occur, and do not invalidate existing tests. The software recovers well from failure.

Understandability: "The more information we have, the smarter we will test." The architectural design and the dependencies between internal, external and shared components are well understood. Technical documentation is instantly accessible, well organized specific and detailed, and accurate. Changes to the design are communicated to testers.

Most systems eventually reach a point when questions arise about their maintainability and supportability. Some systems are supportable for years, while others have supportability problems from initial deployment. Many of these problems are indicative of insufficient resources being applied to system support. The key to having cost-effective systems is to have applied the correct quality controls during initial development and implemented good recovery strategies to existing systems. Quality controls used during maintenance may need to be different than those used when the software was created. There are a number of important issues when considering improving the quality of existing software systems. Some are (Brenda C. et. al. 2002):

- Most likely the system is being managed in a different environment than it was developed.
- Customers or users should be involved and their expectations need to be carefully considered, particularly in terms of failures or errors and availability.
- The people involved in support of the system may not be the same ones that developed it.
- How the system was developed or constructed may not necessarily be obvious to current maintainers.
- Documentation and a change management process may not be adequate.
- Planning for adequate resources and identification of their sources needs to be done.
- Integration into information architectures or modernization plans needs to be considered.

There are good reasons to consider improving the quality of existing systems. If a system is becoming difficult to support, it may well hinder an organization's ability to achieve business success. Support costs can be up to 80% of the system's overall life cycle cost; quality improvements can provide a clear return on investment (Chatzigeorgiou A. et. al. 2003).

To provide a methodology for designing, applying, and validating software quality guidelines, we recommend and briefly summarize IEEE standard 1061 (IEEE Std 1061 1998). This standard gives a process for constructing and implementing a software quality metrics framework that can be tailor-made to meet quality requirements for a particular

project and/or organization. Since the introduction of source code metrics into the discipline of software engineering, controversy has surrounded attempts to validate their usefulness as indicators and discriminators of quality. Traditional metrics such as McCabe's cyclomatic complexity, Halstead's software science metrics, and source lines-of-code, while persisting as commonly used metrics for indicating quality, unfortunately still lack conclusive evidence to support this practice. The reasons why traditional code metrics have been inadequate as measures of quality and complexity include the following(Crutchfield et. al. 1994):

- They are narrow in the scope of software characteristics they measure, and they are defined to be language independent.
- Their conception was strongly influenced by the relative simplicity of programming languages.
- They are based primarily on lexical and syntactic features of code, rather than on the semantic and structural relationships that exist among program units and smaller elements within units.
- They focus on executable statements and generally neglect the influence on complexity of data definitions and the interaction between data and computation flow.
- They have often been designed without regard to the programmers task, problem domain, or environment and ignore the stylistic characteristics of a program.

All testing activities should include processes to uncover defects during the complete life cycle of the software. The focus on full life cycle testing (as opposed to system integration testing only) is important because the cost of defects rises exponentially the later the phase of the cycle. The testing activities include, but are not limited to (Bussieck M. et. al. 2004):

Unit testing: Testing of the individual component using both black-box (input-output only) and white-box (known internal code structure) type tests.

Regression testing: Testing to determine if changes to the software or fixing a defect cause any problems to other components in the system.

العنوان:	Measuring software complexity for software engineering quality assurance
المؤلف الرئيسي:	Wohaishi, Ahmad Muhammad Ali
مؤلفين آخرين:	Fahmy, Maged A.، Al Sultanny, Yas A.(Co-Advisor، Advisor)
التاريخ الميلادي:	2009
موقع:	المنامة
الصفحات:	1 - 155
رقم MD:	736039
نوع المحتوى:	رسائل جامعية
اللغة:	English
الدرجة العلمية:	رسالة ماجستير
الجامعة:	جامعة الخليج العربي
الكلية:	كلية الدراسات العليا
الدولة:	البحرين
قواعد المعلومات:	Dissertations
مواضيع:	برامج الحاسوب، هندسة البرمجيات، تكنولوجيا المعلومات
رابط:	https://search.mandumah.com/Record/736039

Chapter 7

Conclusion and Future Works

7.1 Conclusion

Software complexity measurement methods such as McCabe, Halstead, and cognitive methods are discussed in this thesis. Linear and binary search algorithms are used as a case studies to test the complexity of different object oriented language such as C++, Visual Basic, and Java. Linear and binary search program complexities are measured using methods: LOC without comments, LOC + comments, McCabe complexity, the program difficulty as per Halstead complexity method, and file size are studied and compared regarding three mentioned programming languages. Moreover, modification to Halstead complexity measure is suggested based on the fact that operator execution time for addition, multiplication and division are different. Also, to measure the program complexity, the execution time for microprocessor should be taken into consideration. This execution time for different processor types are compared for 80286, 80486, and Pentium processors. It is found that the operators time for addition, multiplication and division are different for each definitely vocabulary, volume, difficulty and effort for each operator. Then, the cognitive functional size (CFS) used on the basis of cognitive weights, permitting determination of software complexity from both architectural and cognitive aspects. Real-time process algebra (RTPA) has been adopted to describe and measure software complexity. A set of examples had been carried out to analyze the relationship between physical size and cognitive functional size of software. Moreover, software complexity model showed the relation between the problem and constrains that might be affect the quality and productivity of software is introduced. The issues and resources that effect the testing such as usability, reliability, management and risk which will help us to improve the quality to develop the software complexity are studied. Also, the compatibility, maintainability and understandability that effecting the proposed model which will influence the software productivity are studied.

Structural and algorithmic complexity are two aspects of software that had been measured and evaluated. The way of measuring productivity and quality are discussed and found that

weighing these dimensions together in a measure of software performance. McCabe's and Halstead methods were chosen since it is widely used and available among software developments automatic counting tools. It can be implemented early in the software development process. The measure of expected quality can then be seen as an inverse measure of defect density, and McCabe's method is therefore used as an estimation of the number of errors in the code. When these measures of productivity and quality are calculated, an overall picture of expected and real performance were got by drawing the graph description. The measurement of expected performance can be used to evaluate, if more resources are needed for testing the system, and estimate how long time the integration and verification sessions will be taken. If the reliability (quality) of the system is the principal goal, the project should end up in the right part of the graph. On the other hand, if timeliness of the product is most important, and the aiming at placing the project as high up as possible relatively to other projects.

7.2 Future Works

Subjects which need further research and investigation are;

- Build system which will be able to measure automatically the complexity of a program.
- Get the execution time for the other microprocessors and perform the comparison with other type of microprocessors.
- Software provider need to give in table format, the execution time and access time for keywords in all languages such as *if*, *for* and *while* for all type of microprocessors which will help to measure the complexity of software more accurately.
- Implement a new software that can provide measurement of software productivity and quality as suggested in the proposed model.

العنوان:	Measuring software complexity for software engineering quality assurance
المؤلف الرئيسي:	Wohaishi, Ahmad Muhammad Ali
مؤلفين آخرين:	Fahmy, Maged A.، Al Sultanny, Yas A.(Co-Advisor، Advisor)
التاريخ الميلادي:	2009
موقع:	المنامة
الصفحات:	1 - 155
رقم MD:	736039
نوع المحتوى:	رسائل جامعية
اللغة:	English
الدرجة العلمية:	رسالة ماجستير
الجامعة:	جامعة الخليج العربي
الكلية:	كلية الدراسات العليا
الدولة:	البحرين
قواعد المعلومات:	Dissertations
مواضيع:	برامج الحاسوب، هندسة البرمجيات، تكنولوجيا المعلومات
رابط:	https://search.mandumah.com/Record/736039

References

Abdul Ghani A. et. al., 2008, "Complexity Metrics for Measuring the Understandability and Maintainability of Business Process Models using Goal-Question-Metric", International Journal of Computer Science and Network Security, Volume 8 No. 5, May 2008, pp. 219-225.

Abran A. et. al.,2004, "An Analysis of the McCabe Cyclomatic Complexity Number", 12th International Workshop on Software Measurement IWSM2004, Königs Wusterhausen, Germany.

Ali K. et. al., 2004, "A Software Architecture Approach to Remote Vehicle Diagnostics", Msc thesis in Informatics, It University of Göteborg, Sweden, pp. 37-41.

Ali M., 2006, " Metrics for Requirements Engineering", Msc thesis, Department of Computer Science, UMEA University, Sweden, June 2006.

Areejit P. et. al., 2005, "Complexity metrics for manufacturing control architectures based on software and information flow", Computers & Industrial Engineering, Volume 49, pp. 1-20.

Ashrafi 2003, "The impact of software process improvement on quality: in theory and practice", Information & Management, Issue 40, pp. 677-690.

Banker R. 1989, "Software Complexity and Maintainability", International Conference on Information Systems, Proceedings of the tenth international conference on Information Systems, pp. 247 - 255 .

Basili V. 1980, "Qualitative Software Complexity Models: A Summary in Tutorial on Models and Methods for Software Management and Engineering", Los Alamitos, Calif.: IEEE Computer Society Press.

Benussi L., 1995, "Analysing the technological history of the Open Source Phenomenon", FLOSS history, working paper, version 3.0, Department of Economics – University of Turin.

Boehm B., 1989, "Software Risk Management", Institute of Electrical & Electronics Engineering (IEEE), August 1989.

Boy G., 2005, "Decision Making: A Cognitive Function Approach", European Institute of Cognitive Sciences and Engineering, Proceedings of the Seventh International NDM Conference, June 2005.

Brenda C. et.al. 2002, Software Quality Assurance Subcommittee, "Software Quality Assurance Control of Existing Systems", SQA Control of Existing Systems.

Bussieck M. et. al. 2004, " Software Quality Assurance for Mathematical Modeling Systems", GAMA Development Corporation.

Campione M. et. al. 2001, "The Java Tutorial, As Short Course on the Basics", Addison Wesley.

Cardoso, 2006, " Approaches to Compute Workflow Complexity", Dagstuhl Seminar, The Role of Business Processes in Service Oriented Architectures, July 2006, Germany.

Cardoso, 2006, "Complexity Analysis of BPEL Web Processes", Improvement and Practice Journal, John Wiley & Sons.

Cardoso et. al. 2006, "A Discourse on Complexity of Process Models", BPM 2006 Workshops, LNCS 4103, pp. 115–126.

Chatzigeorgiou A. et. al., 2003, " Efficient management of inspections in software development project", Information & Management 45, 2003, pp. 671-680.

Chaumun et. al. 2002, "A change impact model for changeability assessment in object-oriented software system", Science of Computer Programming, Issue 45, pp. 155-174.

Chis 2008, "Evolutionary Decision Trees and Software Metrics for Module Defects Identification", World academy of science, engineering and technology Vol. 28, pp. 273-277 , April 2008.

Crutchfield et. al. 1994, "Quality Guidelines = Designer Metrics", ACM SIGSOFT Software Engineering Notes, pp. 29-40, February 1994.

Dale N. et. al., 2000, "Programming and Problem Solving with C++", Jones and Bartlett.

Daniel G, 2003, "Software Quality Assurance: From Theory to Implementation", Addison-Wesley.

Deitel H. et. al., 2004, "C++ How to Program", Pearson Prentice Hall.

Deitel H. et. al., 2005, "Java How to Program", Pearson Prentice Hall.

Deitel H. et. al., 2006, "Visual Basic 2005 for Programmers", Pearson Prentice Hall.

Deitel H. et. al., 2009, "Internet & World Wide Web How to Program", Pearson Prentice Hall.

Fagerholm 2007, "Measuring and tracking quality factors in Free and Open Source Software projects", Msc thesis, University of Helsinki, Faculty of Science, Department of Computer Science.

Fateman R., 2000, "Software Fault Prevention by Language Choice: Why C is Not My Favorite Language", Computer Science Division, University of California, Berkeley.

Forouzan B. et. al., 2001, "A Structured programming Approach Using C", Brooks/Cole.

Garcia 2008, "Software metrics through fault data from empirical evaluation using verification & validation tools", Msc thesis, Texas Tech University.

Gopal N. et. al., 2004, "Distributed Parallel Virtual Machine: An Object-Oriented Approach", Report, Department of Computer Science, S.C.T College of Engineering.

Gray M., 1999, " Applicability of Metrology to Information Technology", Journal of Research of National Institute of Standards and Technology, Volume 104, Number 6, November – December 1999, pp. 567- 578.

Gruhn V. et. al., 2006, "Complexity Metrics for Business Process Models", 9th international conference on business information systems (BIS 2006), Vol. 85 of Lecture Notes in Informatics, pp. 1-12.

Henry et. al., 1981, "Software structure metrics based on information flow", IEEE Transactions on Software Engineering, Volume 7, No. 5, pp. 510-518.

Hoare C., 1987, "Laws of Programming", Comm. ACM, Vol. 30, No. 8, August 1987, pp. 672-686.

IEEE Std 1061, 1998, "IEEE Standard for a Software Quality Metrics Methodology", Software Engineering Standards Committee of the IEEE Computer Society.

Jones M. et al. 1997, "Twenty years of software engineering standardization in ESA", European Space Research and Technology Centre (ESTEC), Noordwijk, The Netherlands.

Jorgensen P., 2002, "Has the Object-Oriented Paradigm Kept Its Promise?", Department of Computer Science and Information Systems, Grand Valley State University.

Kan S. H., 2003, "Metrics and Models in Software Quality Engineering", Pearson Education.

Kaner C., 2004, "The Ongoing Revolution in Software Testing", Software Test & Performance Conference, December 2004.

Kearney J. et. al., 1986, "Software Complexity Measurement", Vol. 28, New York: ACM Press, pp. 1044–1050.

Klasky H., 2003, "A study of Software Metrics", Msc thesis, Rutgers, The State University of New Jersey, May 2003.

Klemola T., 2000, " A Cognitive Model for Complexity Metrics", Center for Object-Oriented Technology Applications and Research, University of Technology.

Kushwaha, D. et. al., 2005, "A Modified Cognitive Information Complexity Measure of Software", Proceeding of the 7th International Conference on Cognitive Systems(ICCS'05).

Kushwaha, D. et. al., 2006, "Robustness Analysis of Cognitive Information Complexity Measure using Weyuker Properties", ACM SIGSOFT Software Engineer Notes, Vol. 31, No. 1.

Lou M., 1997, "Measuring Software Complexity", Enterprise System Journal.

McCabe 1976, "A Complexity Measure", IEEE Transactions on Software Engineering, Vol. 2, No. 4, pp. 308-320, December 1976.

Magnus A. et. al.,2004, "Object-Oriented Design Quality Metrics", Msc Thesis, Information Technology, Computing Science Department.

Markku O., 1999, "International Conference on Product Focused Software Process Improvement", pp. 503-507.

Moores B., 1994, "Concepts of quality and quality management in industry and the service sector", Technische Universität, Berlin, ALLEMAGNE, Vol. 28, No2, pp. 211-218.

Naeem N. et. al., 2006, "Metrics for Measuring the Effectiveness of Decompilers and Obfuscators", McGill University.

Nasib S., 2005, "Factors Affecting Effective Software Quality Management Revisited", ACM SIGSOFT Software Engineering Notes, Vol. 30, No. 2, pp. 1-4.

Obasanjo D., 2007, "A Comparison of Microsoft's C# Programming Language to Sun Microsystems' Java Programming Language" Article.

Pan J., 1999, " Software Reliability", Article, Carnegie Mellon University.

Pressman R., 2005, " Software Engineering: A Practitioner's Approach", Mc Graw Hill.

Sarif B., 2003, "Modified Ant Colony Algorithm for Combinational Logic Circuits Design", Bsc thesis, Computer Engineering, King Fahd University of Petroleum and Minerals, November 2003.

Schach S. R. , 1997, "Software Engineering with JAVA", McGraw-Hill

Schach S. R. , 2002, "Object-Oriented and Classical Software Engineering", McGraw-Hill

Schildt H., 1998, "The sinle easiest Way to Master C++ Programming", Mc Graw Hill.

Schneidewind, 1975, "Analysis of Error Processes in Computer Software".

Shao J. et. al., 2003, "A new measure of software complexity based on cognitive weights", Canada Journal Electrical Computer Engineering, Vol. 28, No. 2, April 2003.

Smith S., 1999, "The Scientist and Engineer's Guide to Digital Signal Processing", California Technical Publishing.

Sommerville I., 2006, "Software Engineering", Pearson Education Ltd.

Sofia N., 1999, "Software Complexity and Project Performance", Department of Informatics School of Economics and Commercial Law at the University of Gothenburg, pp. 59-61.

Thomas J. et. al., 1994, "Software complexity", Article, McCabe and Associates, Inc.

River C., 2006, "Introduction to Programming in Visual Basic", Charles River Media.

Vigder M. R. et. al. , 1994, "Software Cost Estimation and Control", Article, National Research Council of Canada.

Wang Y., 2003, "The Real-Time Process Algebra (RTPA)", Annals of Software Engineering, Vol. 14, October 2002, pp. 235-274.

Watson et. al. 1996, "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric", National Institute of Standards and Technology.

Woodman M. et. al., 1997, "The object shop—using CD-ROM multimedia to introduce object concepts", ACM SIGCSE Bulletin, Volume 29 , Issue 1, March 1997. pp. 345 – 349.

Yanming C. et. al., 2007, "Exploration of Complexity in Software Reliability", Tsinghua Science & Technology, Volume 12, Supplement 1, July 2007, Pages 266-269.

العنوان:	Measuring software complexity for software engineering quality assurance
المؤلف الرئيسي:	Wohaishi, Ahmad Muhammad Ali
مؤلفين آخرين:	Fahmy, Maged A.، Al Sultanny, Yas A.(Co-Advisor، Advisor)
التاريخ الميلادي:	2009
موقع:	المنامة
الصفحات:	1 - 155
رقم MD:	736039
نوع المحتوى:	رسائل جامعية
اللغة:	English
الدرجة العلمية:	رسالة ماجستير
الجامعة:	جامعة الخليج العربي
الكلية:	كلية الدراسات العليا
الدولة:	البحرين
قواعد المعلومات:	Dissertations
مواضيع:	برامج الحاسوب، هندسة البرمجيات، تكنولوجيا المعلومات
رابط:	https://search.mandumah.com/Record/736039

Appendix A

The examples are taken from How to program C++ Book by Deitel & Deitel Pearson Prentice Hall, 2005.

P 1:

```
// Calculate the product of three integers
#include <iostream> // allows program to perform input and output

using std::cout; // program uses cout
using std::cin; // program uses cin
using std::endl; // program uses endl

// function main begins program execution
int main()
{
  int x; // first integer to multiply
  int y; // second integer to multiply
  int z; // third integer to multiply
  int result; // the product of the three integers

  cout << "Enter three integers: "; // prompt user for data
  cin >> x >> y >> z; // read three integers from user
  result = x * y * z; // multiply the three integers; store result
  cout << "The product is " << result << endl; // print result; end line

  return 0; // indicate program executed successfully
} // end function main
```

A.1: program number (P 1)

P 2:

```
// Exercise 2.16 Solution
#include <iostream> // allows program to perform input and output

using std::cout; // program uses cout
using std::endl; // program uses endl
using std::cin; // program uses cin

int main()
{
  int number1; // first integer read from user
  int number2; // second integer read from user

  cout << "Enter two integers: "; // prompt user for data
  cin >> number1 >> number2; // read values from user

  // output the results
  cout << "The sum is " << number1 + number2
    << "\nThe product is " << number1 * number2
    << "\nThe difference is " << number1 - number2
    << "\nThe quotient is " << number1 / number2 << endl;

  return 0; // indicate successful termination

} // end main
```

A.1: program number (P 2)

P 3:

```
// Exercise 2.17 Solution
#include <iostream> // allows program to perform input and output

using std::cout; // program uses cout
using std::endl; // program uses endl

int main()
{
  // Part a
  cout << "1 2 3 4\n";

  // Part b
  cout << "1 " << "2 " << "3 " << "4\n";

  // Part c
  cout << "1 ";
  cout << "2 ";
  cout << "3 ";
  cout << "4" << endl;

  return 0; // indicate successful termination

} // end main
```

A.1: program number (P 3)

P 4:

```
// Exercise 2.18 Solution
#include <iostream> // allows program to perform input and output

using std::cout; // program uses cout
using std::endl; // program uses endl
using std::cin; // program uses cin

int main()
{
  int number1; // first integer read from user
  int number2; // second integer read from user

  cout << "Enter two integers: "; // prompt user for data
  cin >> number1 >> number2; // read two integers from user

  if ( number1 == number2 )
    cout << "These numbers are equal." << endl;

  if ( number1 > number2 )
    cout << number1 << " is larger." << endl;

  if ( number2 > number1 )
    cout << number2 << " is larger." << endl;

  return 0; // indicate successful termination

} // end main
```

A.1: program number (P 4)

P 5:

```
// Exercise 2.19 Solution
#include <iostream> // allows program to perform input and output

using std::cout; // program uses cout
using std::endl; // program uses endl
using std::cin; // program uses cin

int main()
{
   int number1; // first integer read from user
   int number2; // second integer read from user
   int number3; // third integer read from user
   int smallest; // smallest integer read from user
   int largest; // largest integer read from user
   cout << "Input three different integers: "; // prompt
   cin >> number1 >> number2 >> number3; // read three integers

   largest = number1; // assume first integer is largest

   if ( number2 > largest ) // is number2 larger?
      largest = number2; // number2 is now the largest

   if ( number3 > largest ) // is number3 larger?
      largest = number3; // number3 is now the largest

   smallest = number1; // assume first integer is smallest

   if ( number2 < smallest ) // is number2 smaller?
      smallest = number2; // number2 is now the smallest

   if ( number3 < smallest ) // is number3 smaller?
      smallest = number3; // number3 is now the smallest

   cout << "Sum is " << number1 + number2 + number3
      << "\nAverage is " << ( number1 + number2 + number3 ) / 3
      << "\nProduct is " << number1 * number2 * number3
      << "\nSmallest is " << smallest
      << "\nLargest is " << largest << endl;

   return 0; // indicate successful termination

} // end main
```

A.1: program number (P 5)

P 6:

```
// Exercise 2.20 Solution
#include <iostream> // allows program to perform input and output

using std::cout; // program uses cout
using std::endl; // program uses endl
using std::cin; // program uses cin

int main()
{
  int radius; // variable to store a circle's radius

  cout << "Enter the circle radius: "; // prompt user for radius
  cin >> radius; // read radius from user

  cout << "Diameter is " << radius * 2.0
    << "\nCircumference is " << 2 * 3.14159 * radius
    << "\nArea is " << 3.14159 * radius * radius << endl;

  return 0; // indicate successful termination

} // end main
```

A.1: program number (P 6)

P 7:

```
// Exercise 2.21 Solution
#include <iostream> // allows program to perform input and output

using std::cout; // program uses cout
using std::endl; // program uses endl

int main()
{
  cout << "*********    ***      *      *\n"
       << "*     *   *   *    ***     * *\n"
       << "*     *  *     *   *****   *  *\n"
       << "*     *  *     *    *    *   *\n"
       << "*     *  *     *    *   *    *\n"
       << "*     *  *     *    *   *    *\n"
       << "*     *  *     *    *    *   *\n"
       << "*     *   *    *    *     * *\n"
       << "*********    ***      *      *" << endl;

  return 0; // indicate successful termination

} // end main
```

A.1: program number (P 7)

P 8:

```
// Exercise 2.23 Solution
#include <iostream> // allows program to perform input and output

using std::cout; // program uses cout
using std::endl; // program uses endl
using std::cin; // program uses cin

int main()
{
  int number1; // first integer read from user
  int number2; // second integer read from user
  int number3; // third integer read from user
  int number4; // fourth integer read from user
  int number5; // fifth integer read from user
  int smallest; // smallest integer read from user
  int largest; // largest integer read from user

  cout << "Enter five integers: "; // prompt user for data
  cin >> number1 >> number2 >> number3 >> number4 >> number5;

  largest = number1; // assume first integer is largest
  smallest = number1; // assume first integer is smallest

  if ( number1 > largest ) // is number1 larger?
    largest = number1; // number1 is new largest

  if ( number2 > largest ) // is number2 larger?
    largest = number2; // number2 is new largest

  if ( number3 > largest ) // is number3 larger?
    largest = number3; // number3 is new largest

  if ( number4 > largest ) // is number4 larger?
    largest = number4; // number4 is new largest

  if ( number5 > largest ) // is number5 larger?
    largest = number5; // number5 is new largest

  if ( number1 < smallest ) // is number1 smaller?
    smallest = number1; // number1 is new smallest

  if ( number2 < smallest ) // is number2 smaller?
    smallest = number2; // number2 is new smallest
```

```
   if ( number3 < smallest ) // is number3 smaller?
      smallest = number3; // number3 is new smallest

   if ( number4 < smallest ) // is number4 smaller?
      smallest = number4; // number4 is new smallest

   if ( number5 < smallest ) // is number5 smaller?
      smallest = number5; // number5 is new smallest

   cout << "Largest is " << largest
      << "\nSmallest is " << smallest << endl;

   return 0; // indicate successful termination

} // end main
```

A.1: program number (P 8)

P 9:

```
// Exercise 2.24 Solution
#include <iostream> // allows program to perform input and output

using std::cout; // program uses cout
using std::endl; // program uses endl
using std::cin; // program uses cin

int main()
{
  int number; // integer read from user

  cout << "Enter an integer: "; // prompt
  cin >> number; // read integer from user

  if ( number % 2 == 0 )
    cout << "The integer " << number << " is even." << endl;

  if ( number % 2 != 0 )
    cout << "The integer " << number << " is odd." << endl;

  return 0; // indicate successful termination

} // end main
```

A.1: program number (P 9)

P 10:

```
// Exercise 2.25 Solution
#include <iostream> // allows program to perform input and output

using std::cout; // program uses cout
using std::endl; // program uses endl
using std::cin; // program uses cin

int main()
{
  int number1; // first integer read from user
  int number2; // second integer read from user

  cout << "Enter two integers: "; // prompt
  cin >> number1 >> number2; // read two integers from user

  // using modulus operator
  if ( number1 % number2 == 0 )
    cout << number1 << " is a multiple of " << number2 << endl;

  if ( number1 % number2 != 0 )
    cout << number1 << " is not a multiple of " << number2 << endl;

  return 0; // indicate successful termination

} // end main
```

A.1: program number (P 10)

P 11:

```
// Exercise 2.26 Solution
#include <iostream> // allows program to perform input and output

using std::cout; // program uses cout
using std::endl; // program uses endl

int main()
{
  // Eight output statements
  cout << "* * * * * * * *\n";
  cout << " * * * * * * * *\n";
  cout << "* * * * * * * *\n";
  cout << " * * * * * * * *\n";
  cout << "* * * * * * * *\n";
  cout << " * * * * * * * *\n";
  cout << "* * * * * * * *\n";
  cout << " * * * * * * * *\n\n";

  // One output statement; 3 parts
  cout << "* * * * * * * *\n * * * * * * * *\n* * * * * * * *\n"
    << " * * * * * * * *\n* * * * * * * *\n * * * * * * * *\n"
    << "* * * * * * * *\n * * * * * * * *\n";

  cout << endl; // ensure everything is displayed

  return 0; // indicate successful termination

} // end main
```

A.1: program number (P 11)

P 12:

```
// Exercise 2.27 Solution
#include <iostream> // allows program to perform input and output

using std::cout;
using std::endl;
using std::cin;

int main()
{
  char symbol; // char read from user

  cout << "Enter a character: "; // prompt user for data
  cin >> symbol; // read the character from the keyboard

  cout << symbol << "'s integer equivalent is "
    << static_cast< int >( symbol ) << endl;

  return 0; // indicate successful termination

} // end main
```

A.1: program number (P 12)

P 13:

```
// Exercise 2.28 Solution
#include <iostream> // allows program to perform input and output

using std::cout; // program uses cout
using std::endl; // program uses endl
using std::cin; // program uses cin

int main()
{
  int number; // integer read from user

  cout << "Enter a five-digit integer: "; // prompt
  cin >> number; // read integer from user

  cout << number / 10000 << "   ";
  number = number % 10000;
  cout << number / 1000 << "   ";
  number = number % 1000;
  cout << number / 100 << "   ";
  number = number % 100;
  cout << number / 10 << "   ";
  number = number % 10;
  cout << number << endl;

  return 0; // indicate successful termination

} // end main
```

A.1: program number (P 13)

P 14:

```
// Exercise 2.29 Solution
#include <iostream> // allows program to perform input and output

using std::cout; // program uses cout
using std::endl; // program uses endl

int main()
{
  int number; // integer to square and cube

  number = 0; // set number to 0
  cout << "integer\tsquare\tcube\n"; // output column heads

  // output the integer, its square and its cube
  cout << number << '\t' << number * number << '\t'
    << number * number * number << "\n";

  number = 1; // set number to 1
  cout << number << '\t' << number * number << '\t'
    << number * number * number << "\n";

  number = 2; // set number to 2
  cout << number << '\t' << number * number << '\t'
    << number * number * number << "\n";

  number = 3; // set number to 3
  cout << number << '\t' << number * number << '\t'
    << number * number * number << "\n";

  number = 4; // set number to 4
  cout << number << '\t' << number * number << '\t'
    << number * number * number << "\n";

  number = 5; // set number to 5
  cout << number << '\t' << number * number << '\t'
    << number * number * number << "\n";

  number = 6; // set number to 6
  cout << number << '\t' << number * number << '\t'
    << number * number * number << "\n";

  number = 7; // set number to 7
  cout << number << '\t' << number * number << '\t'
```

```
    << number * number * number << "\n";

  number = 8; // set number to 8
  cout << number << '\t' << number * number << '\t'
    << number * number * number << "\n";

  number = 9; // set number to 9
  cout << number << '\t' << number * number << '\t'
    << number * number * number << "\n";

  number = 10; // set number to 10
  cout << number << '\t' << number * number << '\t'
    << number * number * number << endl;

  return 0; // indicate successful termination

} // end main
```

A.1: program number (P 14)

P 15:

```
// Exercise 3.11 Solution: ex03_11.cpp
// Test program for modified GradeBook class.
#include <iostream>
using std::cout;
using std::endl;

// include definition of class GradeBook from GradeBook.h
#include "GradeBook.h"

// function main begins program execution
int main()
{
  // create a GradeBook object; pass a course name and instructor name
  GradeBook gradeBook(
    "CS101 Introduction to C++ Programming", "Professor Smith" );

  // display initial value of instructorName of GradeBook object
  cout << "gradeBook instructor name is: "
    << gradeBook.getInstructorName() << "\n\n";

  // modify the instructorName using set function
  gradeBook.setInstructorName( "Assistant Professor Bates" );

  // display new value of instructorName
  cout << "new gradeBook instructor name is: "
    << gradeBook.getInstructorName() << "\n\n";

  // display welcome message and instructor's name
  gradeBook.displayMessage();
  return 0; // indicate successful termination
} // end main
```

P 16:

```
// Exercise 3.12 Solution: ex03_12.cpp
// Create and manipulate Account objects.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

// include definition of class Account from Account.h
#include "Account.h"

// function main begins program execution
int main()
{
  Account account1( 50 ); // create Account object
  Account account2( 25 ); // create Account object
  // display initial balance of each object
  cout << "account1 balance: $" << account1.getBalance() << endl;
  cout << "account2 balance: $" << account2.getBalance() << endl;

  int withdrawalAmount; // stores withdrawal amount read from user

  cout << "\nEnter withdrawal amount for account1: "; // prompt
  cin >> withdrawalAmount; // obtain user input
  cout << "\nattempting to subtract " << withdrawalAmount
    << " from account1 balance\n\n";
  account1.debit( withdrawalAmount ); // try to subtract from account1
  // display balances
  cout << "account1 balance: $" << account1.getBalance() << endl;
  cout << "account2 balance: $" << account2.getBalance() << endl;

  cout << "\nEnter withdrawal amount for account2: "; // prompt
  cin >> withdrawalAmount; // obtain user input
  cout << "\nattempting to subtract " << withdrawalAmount
    << " from account2 balance\n\n";
  account2.debit( withdrawalAmount ); // try to subtract from account2

  // display balances
  cout << "account1 balance: $" << account1.getBalance() << endl;
  cout << "account2 balance: $" << account2.getBalance() << endl;
  return 0; // indicate successful termination
} // end main
```

A.1: program number (P 16)

P 17:

```
// Exercise 3.13 Solution: ex03_13.cpp
// Create and manipulate an Invoice object.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

// include definition of class Invoice from Invoice.h
#include "Invoice.h"

// function main begins program execution
int main()
{
  // create an Invoice object
  Invoice invoice( "12345", "Hammer", 100, 5 );

  // display the invoice data members and calculate the amount
  cout << "Part number: " << invoice.getPartNumber() << endl;
  cout << "Part description: " << invoice.getPartDescription() << endl;
  cout << "Quantity: " << invoice.getQuantity() << endl;
  cout << "Price per item: $" << invoice.getPricePerItem() << endl;
  cout << "Invoice amount: $" << invoice.getInvoiceAmount() << endl;

  // modify the invoice data members
  invoice.setPartNumber( "123456" );
  invoice.setPartDescription( "Saw" );
  invoice.setQuantity( -5 ); // negative quantity, so quantity set to 0
  invoice.setPricePerItem( 10 );
  cout << "\nInvoice data members modified.\n\n";

  // display the modified invoice data members and calculate new amount
  cout << "Part number: " << invoice.getPartNumber() << endl;
  cout << "Part description: " << invoice.getPartDescription() << endl;
  cout << "Quantity: " << invoice.getQuantity() << endl;
  cout << "Price per item: $" << invoice.getPricePerItem() << endl;
  cout << "Invoice amount: $" << invoice.getInvoiceAmount() << endl;
  return 0; // indicate successful termination
} // end main
```

A.1: program number (P 17)

P 18:

```
// Exercise 3.14 Solution: ex03_14.cpp
// Create and manipulate two Employee objects.
#include <iostream>
using std::cout;
using std::endl;
 #include "Employee.h" // include definition of class Employee
// function main begins program execution
int main()
{
  // create two Employee objects
  Employee employee1( "Lisa", "Roberts", 4500 );
  Employee employee2( "Mark", "Stein", 4000 );

  // display each Employee's yearly salary
  cout << "Employees' yearly salaries: " << endl;

  // retrieve and display employee1's monthly salary multiplied by 12
  int monthlySalary1 = employee1.getMonthlySalary();
  cout << employee1.getFirstName() << " " << employee1.getLastName()
    << ": $" << monthlySalary1 * 12 << endl;

  // retrieve and display employee2's monthly salary multiplied by 12
  int monthlySalary2 = employee2.getMonthlySalary();
  cout << employee2.getFirstName() << " " << employee2.getLastName()
    << ": $" << monthlySalary2 * 12 << endl;
  // give each Employee a 10% raise
  employee1.setMonthlySalary( monthlySalary1 * 1.1 );
  employee2.setMonthlySalary( monthlySalary2 * 1.1 );

  // display each Employee's yearly salary again
  cout << "\nEmployees' yearly salaries after 10% raise: " << endl;

  // retrieve and display employee1's monthly salary multiplied by 12
  monthlySalary1 = employee1.getMonthlySalary();
  cout << employee1.getFirstName() << " " << employee1.getLastName()
    << ": $" << monthlySalary1 * 12 << endl;

  monthlySalary2 = employee2.getMonthlySalary();
  cout << employee2.getFirstName() << " " << employee2.getLastName()
    << ": $" << monthlySalary2 * 12 << endl;
  return 0; // indicate successful termination
} // end main
```

A.1: program number (P 18)

P 19:

```
// Exercise 3.15 Solution: ex03_15.cpp
// Demonstrates class Date's capabilities.
#include <iostream>
using std::cout;
using std::endl;

#include "Date.h" // include definition of class Date from Date.h

// function main begins program execution
int main()
{
  Date date( 5, 6, 1981 ); // create a Date object for May 6, 1981

  // display the values of the three Date data members
  cout << "Month: " << date.getMonth() << endl;
  cout << "Day: " << date.getDay() << endl;
  cout << "Year: " << date.getYear() << endl;

  cout << "\nOriginal date:" << endl;
  date.displayDate(); // output the Date as 5/6/1981

  // modify the Date
  date.setMonth( 13 ); // invalid month
  date.setDay( 1 );
  date.setYear( 2005 );

  cout << "\nNew date:" << endl;
  date.displayDate(); // output the modified date (1/1/2005)
  return 0; // indicate successful termination
} // end main
```

A.1: program number (P 19)

P 20:

```
// Exercise 4.5 Solution: ex04_05.cpp
// Calculate the sum of the integers from 1 to 10.
#include <iostream>
using std::cout;
using std::endl;

int main()
{
  int sum; // stores sum of integers 1 to 10
  int x; // counter

  x = 1; // count from 1
  sum = 0; // initialize sum

  while ( x <= 10 ) // loop 10 times
  {
    sum += x; // add x to sum
    x++; // increment x
  } // end while

  cout << "The sum is: " << sum << endl;
  return 0; // indicate successful termination
} // end main
```

A.1: program number (P 20)

P 21:

```
// Exercise 4.6 Solution: ex04_06.cpp
// Calculate the value of product and quotient.
#include <iostream>
using std::cout;
using std::endl;

int main()
{
  int x = 5;
  int product = 5;
  int quotient = 5;

  // part a
  product *= x++; // part a statement
  cout << "Value of product after calculation: " << product << endl;
  cout << "Value of x after calculation: " << x << endl << endl;

  // part b
  x = 5; // reset value of x
  quotient /= ++x; // part b statement
  cout << "Value of quotient after calculation: " << quotient << endl;
  cout << "Value of x after calculation: " << x << endl << endl;
  return 0; // indicate successful termination
} // end main
```

A.1: program number (P 21)

P 22:

```
// Exercise 4.8 Solution: ex04_08.cpp
// Raise x to the y power.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

int main()
{
  int x; // base
  int y; // exponent
  int i; // counts from 1 to y
  int power; // used to calculate x raised to power y

  i = 1; // initialize i to begin counting from 1
  power = 1; // initialize power

  cout << "Enter base as an integer: ";  // prompt for base
  cin >> x; // input base

  cout << "Enter exponent as an integer: "; // prompt for exponent
  cin >> y; // input exponent

  // count from 1 to y and multiply power by x each time
  while ( i <= y )
  {
    power *= x;
    i++;
  } // end while

  cout << power << endl; // display result
  return 0; // indicate successful termination
} // end main
```

A.1: program number (P 22)

P 23:

```
// Exercise 4.12 Solution: ex04_12.cpp
// What does this program print?
#include <iostream>
using std::cout;
using std::endl;

int main()
{
  int y; // declare y
  int x = 1; // initialize x
  int total = 0; // initialize total

  while ( x <= 10 ) // loop 10 times
  {
    y = x * x; // perform calculation
    cout << y << endl; // output result
    total += y; // add y to total
    x++; // increment counter x
  } // end while

  cout << "Total is " << total << endl; // display result
  return 0; // indicate successful termination
} // end main
```

P 24:

```
// Exercise 4.18 Solution: ex04_18.cpp
// Print table of values with counter-controlled repetition.
#include <iostream>
using std::cout;
using std::endl;

int main()
{
  int n = 0;

  // display table headers with tabbing
  cout << "N\t10*N\t100*N\t1000*N\n\n";

  while ( ++n <= 5 ) // loop 5 times
  {
    // calculate and display table values
    cout << n << '\t' << 10 * n << '\t' << 100 * n
      << '\t' << 1000 * n << '\n';
  } // end while

  cout << endl;
  return 0; // indicate program ended successfully
} // end main
```

P 25:

```
// Exercise 4.21: ex04_21.cpp
// What does this program print?
#include <iostream>
using std::cout;
using std::endl;

int main()
{
  int count = 1; // initialize count

  while ( count <= 10 ) // loop 10 times
  {
    // output line of text
    cout << ( count % 2 ? "****" : "++++++++" ) << endl;
    count++; // increment count
  } // end while

  return 0; // indicate successful termination
} // end main
```

A.1: program number (P 25)

P 26:

```
// Exercise 4.22: ex04_22.cpp
// What does this program print?
#include <iostream>
using std::cout;
using std::endl;

int main()
{
  int row = 10; // initialize row
  int column; // declare column

  while ( row >= 1 ) // loop until row < 1
  {
    column = 1; // set column to 1 as iteration begins

    while ( column <= 10 ) // loop 10 times
    {
      cout << ( row % 2 ? "<" : ">" ); // output
      column++; // increment column
    } // end inner while

    row--; // decrement row
    cout << endl; // begin new output line
  } // end outer while

  return 0; // indicate successful termination
} // end main
```

A.1: program number (P 26)

P 27:

```
// Exercise 4.23 Solution: ex04_23.cpp
// Dangling-else problem.
#include <iostream>
using std::cout;
using std::endl;

int main()
{
  // part A, x=9 and y=11
  int x = 9;
  int y = 11;
  cout << "Output for part A, x=9 and y=11:" << endl;

  if ( x < 10 )
    if ( y > 10 )
      cout << "*****" << endl;
    else
      cout << "#####" << endl;

  cout << "$$$$$" << endl;

  // part A, x=11 and y=9
  x = 11;
  y = 9;
  cout << endl << "Output for part A, x=11 and y=9:" << endl;

  if ( x < 10 )
    if ( y > 10 )
      cout << "*****" << endl;
    else
      cout << "#####" << endl;

  cout << "$$$$$" << endl;

  // part B, x=9 and y=11
  x = 9;
  y = 11;
  cout << endl << "Output for part B, x=9 and y=11:" << endl;

  if ( x < 10 )
  {
    if ( y > 10 )
      cout << "*****" << endl;
```

```
  } // end outer if
  else
  {
    cout << "#####" << endl;
    cout << "$$$$$" << endl;
  } // end else

  // part B, x=11 and y=9
  x = 11;
  y = 9;
  cout << endl << "Output for part B, x=11 and y=9:" << endl;

  if ( x < 10 )
  {
    if ( y > 10 )
      cout << "*****" << endl;
  } // end outer if
  else
  {
    cout << "#####" << endl;
    cout << "$$$$$" << endl;
  } // end else

  return 0; // indicate successful termination
} // end main
```

A.1: program number (P 27)

P 28:

```
// Exercise 4.24 Solution: ex04_24.cpp
// Dangling-else problem.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

int main()
{
  int x = 5; // initialize x to 5
  int y = 8; // initialize y to 8

  // part a
  if ( y == 8 )
  {
    if ( x == 5 )
      cout << "@@@@@" << endl;
    else
      cout << "#####" << endl;
  } // end if

  cout << "$$$$$" << endl;
  cout << "&&&&&" << endl << endl;

  // part b
  if ( y == 8 )
  {
    if ( x == 5 )
      cout << "@@@@@" << endl;
    else
    {
      cout << "#####" << endl;
      cout << "$$$$$" << endl;
      cout << "&&&&&" << endl;
    } // end inner else
  } // end outer if

  cout << endl;

  // part c
  if ( y == 8 )
  {
    if ( x == 5 )
```

```
      cout << "@@@@@" << endl;
    else
    {
      cout << "#####" << endl;
      cout << "$$$$$" << endl;
    } // end inner else
  } // end outer if

  cout << "&&&&&&" << endl << endl;

  // part d
  y = 7;

  if ( y == 8 )
  {
    if ( x == 5 )
      cout << "@@@@@" << endl;
  } // end if
  else
  {
    cout << "#####" << endl;
    cout << "$$$$$" << endl;
    cout << "&&&&&&" << endl;
  } // end else

  return 0; // indicate successful termination
} // end main
```

A.1: program number (P 28)

P 29:

```
// Exercise 4.28 Solution: ex04_28.cpp
// Prints out an 8 x 8 checkerboard pattern.
#include <iostream>
using std::cout;
using std::endl;

int main()
{
  int row = 8; // row counter
  int side; // side counter

  while ( row-- > 0 ) // loop 8 times
  {
    side = 8; // reset side counter

    // if even row, begin with a space
    if ( row % 2 == 0 )
      cout << ' ';

    while ( side-- > 0 ) // loop 8 times
      cout << "* ";

    cout << endl; // go to next line
  } // end while

  cout << endl;
  return 0; // indicate successful termination
} // end main
```

A.1: program number (P 29)

العنوان:	Measuring software complexity for software engineering quality assurance
المؤلف الرئيسي:	Wohaishi, Ahmad Muhammad Ali
مؤلفين آخرين:	Fahmy, Maged A.، Al Sultanny, Yas A.(Co-Advisor، Advisor)
التاريخ الميلادي:	2009
موقع:	المنامة
الصفحات:	1 - 155
رقم MD:	736039
نوع المحتوى:	رسائل جامعية
اللغة:	English
الدرجة العلمية:	رسالة ماجستير
الجامعة:	جامعة الخليج العربي
الكلية:	كلية الدراسات العليا
الدولة:	البحرين
قواعد المعلومات:	Dissertations
مواضيع:	برامج الحاسوب، هندسة البرمجيات، تكنولوجيا المعلومات
رابط:	https://search.mandumah.com/Record/736039

ARABIAN GULF UNIVERSITY

College of Graduate Studies

Technology Management Programme

Measuring Software Complexity for Software Engineering Quality Assurance

A Thesis Submitted in Partial Fulfillment of the Requirements for the Master's Degree in Technology Management
(Specializing in Engineering)

Submitted by

Ahmed Mohammed Ali Wohaishi

Bachelor of Electrical Engineering, King Fahad University of Petroleum and Minerals, Kingdom of Saudi Arabia, 1999

Supervised by

Dr. Yas A. Al-Sultanny
Associate Professor of Technology Management
Arabian Gulf University

Dr. Maged M. Fahmy
Assistant Professor of Computers
King Faisal University
Kingdom of Saudi Arabia

KINGDOM OF BAHRAIN

January 2009 (A.D.)

Muharram 1430 (A.H.)

جامعة الخليج العربي

كليـــة الدراسات العليا

برنـــامج إدارة التقنية

قياس مستوى تعقيد البرمجيات لضمان الجودة في هندسة البرمجيات

رسالة مقدمة كجزء من متطلبات الحصول على درجة الماجستير في
إدارة التقنية
(تخصص الهندسة)

إعـــــــداد

احمد محمد علي وحيشي

بكالوريوس هندسة كهربائية، جامعة الملك فهد للبترول والمعادن، المملكة العربية السعودية، ١٩٩٩ م

إشـــــراف

د. ياس عباس السلطاني
أستاذ إدارة التقنية المشارك
جامعة الخليج العربي

د. ماجد محمد فهمي
أستاذ الحاسب المساعد
جامعة الملك فيصل
المملكة العربية السعودية

مملكـــة البحـــــرين

محرم ١٤٣٠ هـــ

يناير ٢٠٠٩ م