SOFTWARE COMPREHENSION: THEORY AND METRICS

Tuomas Klemola

A thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

December 1998

0-612-39115-9

Abstract

Software Comprehension: Theory and Metrics

Tuomas Klemola

The understandability of a program specification has a direct bearing on several important aspects of software quality. These include reliability, modifiability, reusability, and maintainability to name a few.

The process of comprehension has been studied by psychologists. Their findings have implications for software engineering practises. A survey of pertinent studies in memory usage and comprehension processes reveals motivators for good software engineering practices.

Software metrics are used in software engineering to predict human performance, for instance faults per thousand lines of code. A survey of software metrics which are related to human performance is included. Recently proposed metrics are examined.

Rules for developing software that respect human limitations are derived based on psychological research and software engineering practises.

An empirical study of human performance against a newly proposed metric based on comprehension processes is done using the performance of students on final examinations. The metric, identifier density, is found to predict human error.

Acknowledgments

I would like to acknowledge professors Deborah Boehm-Davis and K. A. Ericsson for providing me with relevant psychological research; Professors Deborah Boehm-Davis, K. A. Ericsson, and Lynn Reder for answering questions on psychological issues; Professor Brian Henderson-Sellers for directing me to relevant metrics research and for writing Object Oriented Metrics: Measures of Complexity which was a catalyst for this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis is the result of searching for an answer to the question: "Is it possible to identify objective criteria for choosing a programming style, a programming language, and a software methodology for large scale software development? If so, what are they?" Opinions are usually formed with experience, and different experiences often lead to different opinions. Some answers can be found by considering the human factors in software development. Deborah Boehm-Davis, a psychologist who has researched software comprehension processes in programmers states:

> The limited capacity of short term memory provides one of our greatest limitations in developing large scale computer systems.[5]

One of the most significant challenges of Software Engineering remains the specification of design in such a way that computer programs can easily be understood, changed, and their components reused. Understandability is highly subjective, and depends on factors such as familiarity with the application domain and language of implementation. Nevertheless, there are attributes of any specification which can be measured that correspond with important aspects of its understandability.

Some of the most significant delays in software development occur when an individual attempts to understand a specification that they either did not write or wrote a long time ago. By better understanding the processes involved in human comprehension, and applying this understanding to software development, one can aim to reduce the influence of such human limitations. One goal would be to develop tools that help with the comprehension process. This includes evaluating software specifications for their expected understandability, and isolating potential problem areas before beginning to read the specification.

Improving the understandability of software specifications has two steps. Assuming that we can arrive at a definition of the characteristics of understandable software, we must then determine how to create software with those characteristics. Creating understandable software specifications implies a conception of understandability. This in turn necessitates a model of the thought processes associated with software development.

Cognitive ergonomics is a term that is used to conceptualize ease of thought. The software industry has been indirectly concerned with cognitive ergonomics from the beginning. When languages such as COBOL and FORTRAN were conceived, it was with the idea of making the thinking of programming solutions to particular classes of problems (business and scientific) more natural.

A successful approach has been to use metrics within a set of applications and for programmers to identify problem code and clean it up. Understandability is highly subjective. The only sure way to evaluate the understandability of a specification is to have the programmers who must understand them read the specifications to judge their understandability. Some work has been done toward automating the processes involved in software comprehension, which could point the way to implementing an automated comprehension effort estimation system.

1.1 Motivation

Program understandability has a significant effect on the effort required to maintain software, and the programmer's ability to produce correct software. The ability to predict program understandability would be an aid to the developer who wants his product to be maintainable and verifiable. It would also assist the manager who needs to estimate the manpower required to successfully complete a project on time.

When the programmers who must understand the code can verify it for understandability, one can be confident that their evaluation reflects their own comprehension process. Having a programmer read code simply to understand it is too costly and time consuming for most organizations. If one could reliably predict the understandability of a given code segment to a given programmer, then one could get valuable data to assist in estimating the difficulty and time required to understand a body of code.

Some model of understanding is essential if one is to design meaningful predictors of understandability. Works in the domain of psychology, as well as computer science, offer direction as to where solutions may come from.

1.2 Background

In the process of software development, every developer must eventually understand and modify a previously created artefact. Sometimes it will have been created by another developer. Often it will be the developer himself who created the artefact in question. Especially during large software projects, developers may be faced with comprehending work that is in progress in order to fulfil change requests, or to correct errors.

The success of the software developer in these situations will be influenced by the complexity of the specification he must understand. If the complexity of that specification can be known beforehand, or better yet constrained, the developer may be able to proceed in a more efficient way.

Description of the complexity of an artefact can be used in different ways. During development, it can be used to limit the complexity of new specifications to predefined organizational standards so that reuse and maintenance can be facilitated. With existing software artefacts that must be modified, problem areas can be identified at the earliest stages, creating the opportunity for the invention and application of strategies that can relieve these bottlenecks, minimizing secondary effects such as errors of interpretation.

To understand cognitive complexity metrics, there are three areas that must be covered. First, cognitive psychology provides us with an understanding of issues such as short-term memory capacity and attention span which help us to explain when a description reaches a level of complexity that most people cannot deal with efficiently. Next, complexity theory helps us to describe, in mathematical terms, how two documents may differ in complexity. Finally, metrics provides a set of software artefact attributes that can be computed.

1.3 Expected Benefits

If one can capture the effects of short term memory limitations in the form of a metric which corresponds to memory demands in such a way as to predict human error, then one may be able to improve the quality of software by simplifying where possible and isolating code which is complex due to the domain of application. By applying the metric as part of a quality control process, code can be constructed with an objective valuation of understandability, simplifying the job of producing code others can understand.

Metrics programs involve gathering data for about two years before they can be used to predict productivity trends and error rates. A metric which predicts error based on human factors may not require extensive data gathering. Data gathering could provide the basis for predicting debugging time, potentially very useful.

In software development, we work with different kinds of specifications: analysis documents, design documents, software artefacts and so on. By verifying the understandability of a specification, it should be possible to limit errors and improve software production.

A metric which predicts the understandability of code can probably be adapted to work on analysis, design, and other documents involved in software development. It may also find applications in other areas such as standards for technical documents and textbooks.

Chapter 2

Cognitive Complexity and Understandability

2.1 Cognitive Issues

Human beings have a limited capacity to deal with information. As the structure of the information becomes more intricate and as the amount of the information grows, the performance of human beings decreases. Understanding something about human limitations is essential to designing, choosing and using design paradigms (e.g. programming languages) in ways that minimize the effects of human limitations. Such knowledge can also be useful for designing software metrics.

There are a number of results from cognitive psychology that can serve to guide the design of measures that can estimate the amount of cognitive effort a task would require. They can also serve to help understand the behaviour of metrics.

Different models of thought processes have been researched. They generally involve testing the limits of human memory in some way, and then explaining the behaviour when errors occur.

2.1.1 Operationalizing Understandability

Chunking

Chunking is a term used to describe the process of grouping and understanding related single entities that can be processed more easily when grouped under one name.

Program segments with more than seven subparts will likely require the reader to do implicit chunking. Limiting the subparts reduces the cognitive work load [34].

Different programming languages offer different styles of chunking. The principle of limiting the size of the chunks can be applied to any programming language.

Tracing

In software development, the process of chunking often involves searching for definitions of variables or procedures. This process of finding relevant code is called tracing.

Tracing involves searching for an instance of a name. The search may be for a definition, the reading of a value or the assignment of a value. The search may involve a series of parameter declarations at different levels or in different modules, where a naming convention may or may not have been applied. The search may also be simply a look at another file where the definition is known to be. A case that is of high complexity arises in an object oriented program where inheriting code rather than interface is occurring [3].

Landscapes

The process of chunking and tracing can be viewed so that each trace from the top level takes us one level down. Each chunk is delineated by a pair of markers. In this way a visual representation of the chunking and tracing process can be used to evaluate the complexity of a given code segment [8].

2.1.2 Short Term Memory

Perhaps the most familiar result of research in cognitive psychology which has applications to computing is that of Miller where he found that subjects could remember on average 7 numbers or 5 words for up to 30 seconds[36]. When the number of items to remember exceeds the individual's limits he/she must use memorization and categorization techniques to perform chunking on the information. This process may depend on the patterns the subject is familiar with. Knowledge and experience govern the size and complexity of chunks that can be recalled.[17]

2.1.3 Long-Term Memory

Long-term memory is generally considered to be limitless in capacity[36]. Long-term memory distinguishes itself from short-term memory in that it is durable, and retrieval cues are needed in attention to access information[14].

2.2 Comprehension

Walter Kintsch is a psychologist who has extensively researched comprehension[27]. Kintsch puts forward a theory of comprehension, the construction-integration theory: "A context-insensitive construction process is followed by a constraint-satisfaction, or integration, process that yields if all goes well, an orderly mental structure out of initial chaos." This may be analogous to a bottom-up discovery process followed by a top-down resolution process. This is similar to the object oriented design process described by Wirfs-Brock et. al.[44].

A study by Ericsson and Kintsch[14] of how experts use memory has found that the ability to store domain specific information can exceed the usual limits of short term memory (STM), however this improved performance is always specific to the domain of expertise. Domain specific retrieval structures are developed with experience. This has some parallels with chunking, in the sense that an expert will have chunked a great amount of information. Patterns of information specific to their domain may become chunks which will facilitate their recognition. When experts read a text with the aim of understanding it, if it is in their domain, their immediate recall of what has just been read will be virtually perfect. For instance, chess masters exhibit the ability to follow as many as 20 games with their eyes closed. However, when an expert is asked to remember something out of his domain, his memory follows typical STM patterns.[27]

Domain knowledge plays an important role in text comprehension. When a reader encounters text which does not fit well with what they know, an appropriate retrieval cue must be generated and this may be a difficult problem-solving task.[27]

Tests have shown that when an expert on a subject writes a text which lacks some basic background, other experts will understand the text while readers unfamiliar with the missing background will not understand it. Providing explicit bridging material in text is necessary for the understanding and learning of readers without background knowledge. It is difficult to know exactly the right amount of coherence in a text, since too much may be confusing (e.g., legal documents)[27].

Readers with good domain knowledge learn better from texts which require them to relate what they know with the presented information. An experiment by McNamara et. al. (1996) pitted high knowledge and low knowledge readers reading high coherence and low coherence texts. An example of low coherence text would be:

> The heart is connected to the arteries. The blood in the aorta is red.

Low knowledge readers might not know the relationship between *arteries* and *aorta*[27]. A corresponding high-coherence text might be:

> The heart is connected to the arteries. The blood in the aorta, the artery that carries blood from the heart to the body, is red.

The subjects are then put to answer questions of four types: "(1) text-based questions, (2) elaboration questions that required relating text information to the reader's background knowledge, (3) bridging inference questions that required connecting two or more separate text segments, and (4) problem-solving questions that required applying text information in a novel situation."[27]

In the first two tests, both high knowledge readers and low knowledge readers performed better with the high-coherence text. In the third and fourth tests, high knowledge readers scored better with the low-coherence text than with the high-coherence text, and the low knowledge readers scored better with the high coherence text and very poorly with the low coherence text. The results suggest that learning for problem solving and deeper understanding is aided by low coherence rather than high coherence provided that the reader has adequate background knowledge. This is explained by the fact that the reader must make an effort to construct an internal representation of the meaning of the text when confronted with less than complete coherence. This internal representation then facilitates processes which require a deeper understanding.[27]

Comprehension is a simpler activity than problem solving, except for expert problem solvers who can recognize problems they have seen before and solve them easily. These experts rely on their long-term working memory for problem solving whose difficulty is reduced to that of text comprehension.[14]

2.3 Memory Capacity and Comprehension

Differences in language comprehension and differences in memory capacity have been studied together and reveal a significant relationship. In [24], Just and Carpenter propose a theory that links processing and storage potential with memory capacity, and how individual differences in memory capacity can affect performance in comprehension tests.

An empirical study demonstrated that when STM demands are coincidentally increased, processing is compromised. This suggests that STM is used for both storage and processing.

A study of college students found that their performance differences are small when the comprehension task is easy, but large and systematic when the comprehension task is demanding. This suggests that code which is easy to understand has a much better chance of being maintained without the introduction of errors than code which is complex.

The test used was designed to simultaneously draw on the processing and storage resources of working memory. After reading a pair of sentences, the subject tries to recall the final word of each sentence. The test determines the maximum number of pairs of sentences per set such that the subject can recall all the final words. Reading spans varied from 2 to 5.5 sentences.

The rationale behind the test is that subjects with greater memory capacity will be able to remember more words since the reading of the sentences would require less of their total capacity. The results were found to have a high correlation with other measures of reading comprehension such as the verbal standard aptitude test (SAT). The ability to recall a list of digits or unrelated words is not significantly correlated with reading comprehension.

Processing sentences with a single concept associated with two different roles simultaneously, such as "The reporter that the senator attacked admitted the error.", poses a difficulty in language comprehension. Individual differences were most evident when the subject read difficult or complex portions of a text. In a test modulating the complexity of the sentence read, much of the quantitative difference that results from working memory capacity can be localized to the parts that are more capacity demanding.

Individuals with lower memory capacity were not slower in all reading comprehension tests. When an ambiguous verb was used, the subjects with higher memory capacity spent more time. This supports the hypothesis that subjects with greater memory capacity maintain the ambiguity in working memory for a longer time than the other subjects. This result was found to persist when the sentences were embedded in paragraphs. The comprehension of subjects with low memory capacity was found to be lower than those with high memory capacity. This demonstrates that individual differences in memory capacity can lead to differences in comprehension.

When a series of words or digits is required to be retained while reading, it is called an extrinsic memory load. As the load increases, the reading rate and the ability to recall items is degraded. This is typical of what is involved in reading code.

If two related sentences have unrelated sentences between them, the text will take longer to read. This underscores the importance of reducing large blocks of code into distinct smaller ones that accomplish a single task. Otherwise, more time will be needed to understand a program segment.

When a sentence ends with an ambiguous word, better comprehenders ignored it while poorer comprehenders would retain it in memory.

2.3.1 Conclusion

Individual differences in memory capacity can explain differences in comprehension performance. When memory resources are completely occupied, additional processing cannot occur.

Just and Carpenter suggest that the results of this study have implications in areas other than language such as problem solving, complex decision making, and areas which involve sequential symbol manipulation. Individual differences within a task domain will be explained in large part in terms of working memory capacity. In tasks which involve no language use at all such as arithmetic, working memory capacity will have a lower correlation than with reading performance.

Other sources of individual differences include the speed of word decoding, vocabulary size, and higher level comprehension processes such as syntactic, semantic, and referential-level processes. Individuals with a larger memory capacity may also exhibit better vocabulary acquisition.

2.4 Modelling Working Memory

There are several models that have been used to describe working memory. One measures the load a subject can maintain while performing a task. Another involves matching production rules with currently available information. Both of these are an influence in a theory called ACT elaborated by Anderson, Reder, and Lebiere in[2].

In one experiment it was found that substituting fractions for integers in a simple arithmetic transformation increased the error rate. The increase in error was attributed to increased memory load.

A series of experiments with randomly generated arithmetic transformations were performed (e.g. solve for x when $x + 6 = 9$). Varying the number of digits found that when the number of digits is increased from 4 to 6 the error increases more than when the digits are increased from 2 to 4. Also that when the digits are increased from 6 to 8, the effect is greater than when they are increased from 2 to 6. More complex equations (e.g. $3/4 + x = -7/2$) generated slightly more errors.

2.5 Empirical Measurement of Software Comprehension

Essential to arriving at meaningful cognitive complexity measures, is the study of how programmers understand software. A study consisting of two experiments was undertaken by psychologists at George Mason University[6].

In the first experiment, subjects with little experience, some experience, and significant experience were given software comprehension tasks involving programs which were either object oriented or functionally decomposed. The purpose of the study was to better understand chunking activity. It was found that programmers developed hierarchical representations of the programs, with the more experienced programmers having more levels of nesting and a greater number of units in their internal representations. The study also found that programmers tend to chunk code based on program scopes identified with begin-end delimiters.

In the second experiment, inexperienced subjects were paired , and their activities were monitored as they studied the same two programs as in the first experiment. It was found that programmers made frequent use of variable tracing to construct a model of how a program worked.

2.6 Simulation of Memory Models

A number of models of human memory have been created whose behaviour closely resembles human behaviour in controlled experiments. Some experiments involve using the SOAR architecture, developed by Newell and his colleagues, which implements a model of learning based on productions and chunking[1].

2.6.1 Modeling Episodic Indexing

Altmann investigated programmer behaviour, then designed a simulation of mental indexing based on how programmers will scroll to off-screen targets while understanding a program[1]. He calls the behaviour *episodic indexing*, and its theory is implemented using SOAR and compared to actual programmer behaviour. The goal is to develop a model which can help explain tracing behaviour in programmers by maintaining a mental index to what objects exist in the environment. During a comprehension task, the view of the programmer is controlled to one screenful at a time and what he looks at and for how long it is monitored.

2.6.2 Modeling Software Design

Using the SOAR architecture and the study of how humans design algorithms, a theory of the algorithm design process is implemented.[42] The SOAR architecture constrains the system to learn by chunking. The system successfully designs some algorithms such as Insertion sort, which required 35 minutes on a Sun3/260.

2.6.3 Capacity Constrained Comprehension

A simulation of language comprehension with constrained memory capacity is compared with the performance of human test subjects. The premise for the simulation is that cognitive processing requires resources, which if unavailable, results in cognitive error. The results of the simulation closely match human performance and demonstrate the hypothesis that qualitative and quantitative differences in language comprehension can be accounted for by differences in working memory capacity.[24]

2.6.4 Conclusion

Some cognitive activities involved in various stages of software development have been simulated closely enough to suggest that it is feasible to implement an automated model of comprehension to assist in evaluating software specifications. Such a program could reveal ambiguities, contradictions, and other aspects of bad design in its attempt to construct a consistent and correct internal representation of a specification. It could be a step in further automating software development, especially in domains which are stable.

2.7 Software Understandability

The influence of software understandability in software engineering is manifold. As an internal product quality it is linked to evolvability and verifiability[19]. This suggests that understandability is important during the initial product development when it is evolving and being tested, as well as during the product's lifetime as it is maintained and reused.

Recognizing cognitive limitations in some quantitative form is something that has become more prevalent in software development as the size and complexity of applications increases. Notably, user interface design has become more responsive to cognitive limitations.[17]

Long term memory is involved in software development in remembering such things as computer languages and how to use software development tools. Some short term memory is involved in the use of software development tools. The less short term memory that is needed to use the tool, the more that will be available for understanding and reasoning about the specification at hand.

A specification is like an interface to an implementation. It provides the programmer with a hierarchical structure to use as a guide to navigate the details of the implementation. Depending on the methodology, differing amounts of short term memory will be required to understand the various parts of a specification. Much like user interfaces give access to an application, implementations can be made more accessible by specifications designed with human limitations in mind.

Minimizing the short term memory demands of the software developer, will enhance his comprehension, and reduce the error rate of the software he produces. It will also improve his efficiency where the most difficult tasks are concerned. It is by relieving the point of greatest stress in the software development process, that the conscious application of standards for cognitive ergonomics can improve the efficiency of the software developer.

This improvement in efficiency is significant when one considers the improvement in software quality that is realized. Errors will be much more easy to identify, since often errors are hidden inside a complex sequence of instructions whose understanding is unreliable.

If one is to talk of understandability, then one should have a definition to work from so that ambiguity can be limited. If a software system is understandable, then each of its components is understandable, as is the structure that relates the components. The domain of application should be part of the specification, and the structure of the system domain-oriented.

2.8 Levels of Understandability

A programmer developing a large program will typically use an architecture, a methodology, and a programming style. Each will play a role in the understandability of the final product.

The architecture has a significant influence on the understandability of large systems. Software architecture is defined as the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.[23] An architecture is essentially a software design pattern which is used when constructing software of a known design, such as a compiler, where a particular architecture, such as pipes and filters, works well.[41]

The process of understanding software has many levels. At the lowest level, the programmer must understand the programming language, data types, variable definitions, module specification, and so on. At higher levels of abstraction, the programmer must relate the program to the problem which is being solved.

2.8.1 Conceptual Modelling

Understanding a system involves the building of a conceptual model. We build a conceptual model by starting with a familiar analogy, and learning the differences from the system. The conceptual model is an aid in learning about, and predicting the behaviour of systems [10].

Limiting the complexity of the conceptual models used in designing and coding a system results in improved understandability.

2.8.2 Formal Methods

Formal methods reduce the complexity of software development in the sense that large blocks of specification can be abstracted to a correct interpretation of a higher level specification.

2.8.3 Paradigm Shifts

Kuhn has identified 'paradigm shifts' as discontinuous, disruptive changes in the evolution of scientific thinking [29]. This is of particular concern to software developers since many paradigms are involved in a typical project. There may be different processes for analysis, design, and implementation. Several languages may be used in an implementation.

2.9 Object Orientation and Understandability

Hsia et al. found that software is easier to maintain when inheritance hierarchies are broad and that reuse is easier when the hierarchies are deep[23]. Deep inheritance hierarchies result in smaller classes, a more complex design, and greater dependency among objects. Isolating a problem area is more difficult when the level of inheritance is high. Broad inheritance hierarchies result in simpler debugging than deep hierarchies since problem areas are easier to isolate.

2.9.1 Object/Action Model in GUI

The user is supplied with a set of objects on which he can perform a set of actions. Use of this model has resulted in the observation that "it leads to relatively simple models of complex systems, which are flexible to use, and easy to change (e. g. by adding new objects, adding or changing actions)" [10].

2.9.2 Hierarchies

Hierarchies have become an important part of computer programming. Object oriented libraries are built using inheritance hierarchies. Analysis and experiments have shown that when backtracking across abstraction levels is required to arrive at a concrete-level solution, hierarchical problem solving is less effective [3]. This is the case where a trace through a hierarchy does not end with an immediate parent, but rather with another child of an ancestor. This suggests that the use of inheritance hierarchies with anything but abstract inheritance should be limited in the interest of maintaining understandability.

The idea of backtracking is analogous to the process required in object oriented programming when chunking and tracing to understand an object's behaviour. Clearly, there will be more chunking and tracing when an inheritance hierarchy is large and involves inheriting from concrete classes. Consequently, cognitive complexity is increased.

Gamma et al. suggest object composition or black-box reuse is a style of programming that limits the possibility of hierarchies with dependencies due to parent class internals being visible to the child class, that require the programmer to trace through hierarchies to understand system behavior[18].

2.9.3 Functional Modelling vs. Object Oriented Modelling

A study by Kim and Lerch published in 1992 concluded object orientation (OOD) offers a better mapping of problem entities than the procedures and data structures utilized in traditional functional decomposition (TFD) methodologies. A mental simulation is defined as a cognitive process where either the programmer is traversing the problem space in the task domain or running logical design constructs in the solution domain. The results of a pilot study suggest that a tenfold reduction in time spent in mental simulation was achieved using OOD over TFD [26].

2.10 Conclusion

We know that all specifications are not equally understandable. We also have some knowledge of the comprehension process. Specifications which are hard to understand take more time to read and correctly interpret than specifications which are easy to understand. They are also more likely to result in human error, especially when time is constrained.

We have seen that individuals' short term memory (STM) capacity is a limiter on comprehension skill, and that it varies between individuals. When an individual becomes an expert in a domain, his long-term working-memory acquires structures which perform much like STM for the domain of expertise. We should therefore respect individual differences if we expect different individuals to understand a specification.

Another factor in the way we understand specifications is the abstraction methodology employed to create the specification. If different paradigms are used together, the reader must translate between paradigms to understand the document. This added burden compromises comprehension.

Experienced software developers think of applications in terms of the domain of application. If the software system can be structured like the domain of application, it removes an important paradigm shift.

Object orientation has proved to be a paradigm which can be used to represent software systems in terms of domain concepts. It has been observed that programmers are better able to perform mental simulations which are part of the design process when using object orientation.

Inheritance, a feature of object orientation, has been shown to be a hindrance to the understandability of programs when code is inherited.

When we are choosing or designing specification standards such as design methodologies or programming languages, we can use knowledge of the human comprehension process to improve the understandability of the specifications we create. This can result in fewer errors, time saved, and a more agreeable experience for the user of the specification.

Chapter 3

Metrics

When estimating the duration and cost of a software project, it is helpful to have measures of software size and complexity. The time or effort required to complete a given project can depend on many factors. These include program size, program complexity, language of implementation; the programmer's familiarity with the language, with the domain of application, and perhaps familiarity with an existing implementation, to name a few.

Metrics are used at various stages of the software lifecycle.

3.1 Weyuker's axioms for Complexity Metrics

Axiom sets to aid in the evaluation of complexity metrics have been proposed by Weyuker[28].

Weyuker states that the axioms specify a desirable model for complexity measures, adding that things which are not complexity measures may satisfy the axioms. If something does not satisfy the axioms, then it is not a complexity measure. In mathematics axiom sets are necessary and sufficient to define a set. Weyuker's axioms are necessary but not sufficient.

Terminology: The complexity of a program x is $|x|$. The set of programs is P. The concatenation operator is ";".

Axiom 1: There exist x and y in P such that $|x| \neq |y|$.

Axiom 1 excludes measures which give all programs the same value.

Axiom 2: For a nonnegative number c, there are only a finite number of programs of that complexity.

Properties 2 and 3 ensure sufficient resolution.

Axiom 3: There exist x and y in P such that $x \neq y$ and $|x| = |y|$.

Axiom 4: There exist x and y in P such that their behaviour is equivalent, and $|x| \neq |y|$.

Axiom 4 suggests that different implementations of the same functionality, such as different sorting algorithms, can have different complexity.

Axiom 5: For all x, y in P, $|x| \leq |x;y|$ and $|y| \leq |x;y|$.

Axiom 5 expresses that the complexity of the sum of two parts is at least as great as the complexity of either part.

Axiom 6: There exist x, y, and z in P, such that $|x| = |y|$ and $|x;z| \neq |y;z|$.

Axiom 6 states that the complexity of the concatenation of code segments x and y of the same complexity, with a third code segment z, will sometimes result in code segments of different complexity.

Axiom 7: There exist x and y in P such that y can be derived by permuting the statements of x yet $|x| \neq |y|$.

Axiom 7 states that changing the order of the statements can change the complexity.

Axiom 8: For programs x and y in P, if y is a renaming of x then $|x| = |y|$.

Axiom 8 states that changing the names in a program does not change its complexity.

Axiom 9: There exist programs x and y in P such that $(|x| + |y|) < |x;y|$.

Axiom 9 states that in some cases, the concatenation of two programs will result in a program more complex than the sum of the complexities of the components.

Fenton notes that Weyuker's axiom attempt to describe aspects of both psychological and structural complexity, and suggests that these are fundamentally incompatible and cannot be represented by the same set of axioms[28]. In particular, Fenton is concerned that Axiom 5 is specific to complexity and Axiom 6 is appropriate for comprehension[22].

Axioms 6, 7, 8 and 9 seem most easily supported when semantics are a factor, yet axiom 8 seems to explicitly nullify the effect of semantics. The axioms seem to suffer from the weaknesses of recursive definition since complexity is being defined in terms of complexity. A psychological model is needed to provide an unambiguous perspective where complexity is defined.

3.2 A Moller and Paulish Guide to Using Metrics

Moller and Paulish survey active metrics programs in [32]. Metric are used for the improvement of the development process. By identifying the origin of a fault, it is possible to improve the process which caused the fault. In one survey, companies which implemented a metrics program saw a 25% reduction in production costs and a 10% improvement in production time. It normally takes about two years for a company to acquire enough data about their processes and products to begin to see the benefits of a metrics program.

Metrics offer a means of measuring software quality. Improving software quality using metrics leads to reduced production times, reduced fault rates, reduced costs, and more satisfied customers. The first step to using metrics is to identify specific goals for improving software quality and productivity. For example, to reduce high-level design faults by half.

Software quality and productivity can be improved with software metrics. This leads to better profit margins and a faster time to market.

Metrics can be applied to the various phases of software development. By comparing previous trends with a current project, estimation and planning of resource requirements can be more accurate.

If a high number of faults are found at an early stage, efforts can be made to improve the output of that stage to help ensure a timely completion.

It is suggested that initially, a small number of metrics be chosen. Metrics that measure program size such as lines of code or function points, are used to set other metrics into perspective such as faults per thousand lines of code. They are often counted by in-house tools. They may be indicators of complexity since bigger programs are usually more complex.

Effort can be measured in terms of person-days, and is a productivity indicator. Elapsed time can be measured for each phase. The number of faults can be counted during each phase of software development. The definition of a fault and the period during which faults are counted must be specified. Fault counts are a primary indicator of product quality.

The effort spent due to faults can be tracked to help assess the importance of different kinds of faults.

Additional metrics can be defined and implemented as the need becomes more clear.

Various software development teams in Siemens have benefited from the application of metrics. The recognized improvements include product quality, staff productivity and motivation, and customer satisfaction. In providing a perspective that identifies the effect of quality between phases, the software development process as well as communication between developers is improved.

Most metrics used by Siemens are based on faults found during various stages, schedule information, and product size data. With these values productivity (i.e. size/time) and quality (i.e. faults/size) metrics are computed and used to track the progress of the project.

Moller and Paulish demonstrate the value of using metrics. Even simple metrics such as LOC/Time and Faults/KLOC provide feedback which can motivate improvements in the way programmers work. These improvements result in reduced development times, increased product quality, improved customer satisfaction, and better programmer morale. The two year investment that is needed to start a metrics program seems to be a good one.

3.3 Lines of Code: LOC

LOC is the oldest software measure. It was first used to measure the size of fixed-format assembly languages. Since programming languages have evolved to support multiple instructions per line, and abstraction as with high-level languages, the use of LOC as a metric has become problematic. For instance, a compound conditional statement will be counted as a single line of code of a high level language, but would require many assembly instructions, whereas an assignment statement would require two assembly language statements. In each case, one line of code is counted.

Some studies suggest that for some language/application situations, LOC is at least as good as Halstead's metric and McCabe's cyclomatic complexity metric.

The most accepted definition of LOC is as follows:

> A line of code is any 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. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements.

However, LOC is not useful as an effective measure for understandability since it does not account for the complexity of a line of code[22].

3.4 Halstead's Software Science

Halstead developed a series of metrics based in part on a study of highly refined algorithms in early forms of Algol, PL/1, FORTRAN, and assembly language[21]. He offers experimental results which are surprisingly supportive of his conclusions.

His metrics are computed from the total and unique numbers of operators and operands. Operands are defined as "variables or constants". Operators are defined as "symbols or combinations of symbols that affect the value or ordering of an operand". He treats opening and closing parenthesis, as well as BEGIN and END program grouping pairs as single operators.

$$\begin{aligned} n_1 &= \text{the number of unique operators} \\ n_2 &= \text{the number of unique operands} \\ N_1 &= \text{the total number of operators} \\ N_2 &= \text{the total number of operands} \end{aligned}$$

The vocabulary of a program, n, is defined as the sum of the operators and operands ($n = n_1 + n_2$), and the length of the program, N, is equal to the sum of the total number of operators and operands ($N = N_1 + N_2$).

Using these basic metrics, Halstead defines several different metrics, some of which are intended to correspond with or predict human behaviour measures. These are the difficulty D, the intelligence content I, and the programming effort E.

Difficulty (D) $\quad D = \left(\frac{2}{n_1}\right) \times \left(\frac{N_2}{n_2}\right)$

Intelligence (I) $\quad I = \frac{2}{n_1}\frac{n_2}{N_2} * (N_1 + N_2)\log_2(n_1 + n_2)$

Effort (E) $\quad E = \frac{n_1}{2}\frac{N_2}{n_2} \times (N_1 + N_2)\log_2(n_1 + n_2)$

The difficulty is the inverse of level: $L = 1/D$. L is intended to give a relative measure which is greater for succinct expressions of algorithms. Consequently, more voluminous representations have a higher value of D.

Intelligence content, I, is a value that Halstead speculated might have the same value for any particular algorithm expressed in different languages.

Programming effort E, represents the "total number of elementary mental discriminations...required to generate a given program." Halstead assumes that human beings use binary search in their mental discriminations and that all mental discriminations require the same amount of time.

3.4.1 Conclusion

The idea that a programmer must think about how operands and operators relate while coding algorithms is a sensible beginning to creating metrics. However, we know that individuals vary widely in their performance both in time and error rate.

In one of Halstead's experiments, he had several programmers familiarize themselves with certain programs over a period of months, and then asked them to perform tasks which involved understanding the code. This negates the learning difficulty which could arguably be the most significant component of what is measured as complexity.

Halstead believed that metrics could be developed to measure the complexity of technical documents by classifying words as operators and operands, and applying the same metrics he used to measure software.

A type of token, the function that returns a value, is not given special treatment by Halstead's metrics which are intended for the analysis of algorithm implementations. A function can be both an operator and an operand in the sense that it takes arguments and has a value in an expression.

A study by Fitzsimmons and Love (1978) found that Halstead's effort metric did correlate well with programmer performance on various software comprehension tasks, averaging 0.75 across 7 studies[5].

3.5 McCabe's Cyclomatic Complexity

McCabe's metric is based on an analysis of the control flow graph of a program and is defined as

$$C = e - n + 2p$$

where e is the number of edges, n is the number of nodes, and p is the number of connected components of the graph (usually 1). The metric can also be calculated by counting the number of decision statements (predicate nodes) in the program and adding 1.[19]

A limit for the cyclomatic complexity of a code segment of ten is proposed by McCabe. Code segments with larger values should be split. The current trend is to write smaller code segments and lower values are preferred. Intuitively, any code segment with ten decision statements could be hard to understand.

McCabe's measure does not account for the complexity of the condition. Since there are several ways of writing conditional statements, the metric is not neutral to programming style, as it should be. However, if the metric is applied on programs from the same set of programmers, the utility of the metric is improved.

For instance, the following code segment

```
IF (A and B)
  I := 1
```

is logically equivalent to

```
IF A
  IF B
    I := 1
```

McCabe's cyclomatic complexity would suggest that the second code segment is more complex than the first even though they achieve the same purpose. This compromises the value of the metric.

3.6 Function Points

The function point counting is used to estimate development effort. It is considered to be somewhat language independent. It has been found to be useful in predicting system size early in the development life cycle[22]. It is based on the following weighted items, where the weighting is determined through a sample of existing applications:

Number of external inputs
Number of external outputs
Number of external inquiries
Number of internal master files

Number of external interfaces

Function points have been criticized for not being orthogonal and for being more complex than other predictors which are as good. Function points have been used mainly in business applications where the domain is relatively contained.

The accuracy of function point metrics is dependent on the similarity between the domain and set of programmers involved in the sample to determine the weights. As with most metrics, as time passes, the samples become out of date, and the accuracy of the metrics are degraded. In order to maintain accurate function point metrics, weights should be recomputed with current samples that reflect the set of programmers and the domain of application. With function point metrics, more work is involved in such maintenance.

3.7 Chunking Metrics

In one study, an operational definition for chunks is used to compute several metrics[11].

> A chunk is a sequence of one or more contiguous program statements $S_i, S_{i+1}, ..., S_{i+n}$ with the property that there is no explicit transfer of control to any statement in the sequence other than S_i.

There are two interchunk dependencies which form the basis for the metrics. Data dependencies occur when a variable can be updated in one chunk and is referenced in another. Control dependencies between chunks exist if there is a potential transfer of control between chunks.

Metrics are tested for debugging time, error rate, and construction time. A data dependency exists between two chunks if a variable that can be set in one chunk can be read in another. The metric DATFAN, the total number of data dependency connections between chunks seemed to follow a non-linear relationship with the number of errors. After exploring transformations of the number of errors, DATFAN was found to correlate well (r = 0.92) with the square of the number of error occurrences.

3.8 Object Oriented Metrics of Mark Lorenz

Lorenz describes two categories of metrics: project and design metrics[30]. Project metrics are used to predict staffing needs and total effort. Design metrics look at the quality of the way the system is constructed.

3.8.1 Project Metrics

Three kinds of metrics are offered. pertaining to application size, staffing size, and scheduling.

Application size metrics include NSS, the number of scenario scripts; NKC, the number of key classes; NSC, the number of support classes; and NOS, the number of subsystems.

A scenario script is "a sequence of steps the user and system take to accomplish some task. Each step consists of an initiator, an action, and a participant." Scenario scripts are useful as a form of specification which both the user and developer can understand and agree on. Lorenz suggests at least one script per subsystem contract.

A key class is "a class that is central to the business domain being automated. A key class is one that would cause great difficulties in developing and maintaining a system if it did not exist." Key classes are a good indicator of how much effort will be required. They are specific to the problem domain of the client and are often targeted for reuse. Lorenz suggests that 20-40 percent of classes should be key classes.

A support class is "a class that is not central to the business domain being automated but provides basic services or interface capabilities to the key classes." Support classes are discovered later in the development process. Their number is an indication of effort. High numbers may be an indication of poor factoring into classes. There should be at least as many support classes as key classes.

A subsystem is "a group of classes that work together to provide a related group of end-user functions." It should be possible to develop a subsystem independently of other subsystems. The number of subsystems can help to plan the development schedule. The subsystems can define the architecture of the system.

Staffing size metrics include PDC, person-days per class and CPD, classes per developer.

Lorenz estimates ten to fifteen days per class. The value comes down as the ratio of OO experts to novices increases. Reuse is also a factor. The number of classes per developer depends on the duration of the project and the experience per developer. Lorenz suggests giving key classes to the most experienced developers.

Scheduling metrics include NMI, number of major iterations, and NCC, number of contracts completed.

An iteration is "a single circle of an iterative process, consisting of planning, production, and assessment phases over a multimonth period of time." Iterations improve designs and allows user feedback to be a stronger influence on the final product. Lorenz suggests three to six major iterations.

A contract is "a simplifying abstraction of a group of related public responsibilities that are to be provided by subsystems and classes to their clients." Contracts correspond well to end-user functionality. Lorenz believes that contract based estimation is more accurate than class based estimation.

3.8.2 Design Metrics

There are many kinds of design metrics described. They include measures for method size, method internals, class size, class inheritance, method inheritance, class internals and class externals. They are used to improve the quality of the product, especially during development.

Method size metrics include NOM, the number of message sends, the number of statements, and LOC, the lines of code.

There are three types of messages. Unary messages are messages with no arguments, binary messages have one argument separated by a selector name such as math functions, and keyword messages have one or more arguments. Their counts are added together to make NOM. It is a less biased measure of method size than LOC which may be affected by style differences. Methods with 9 or more message sends should be checked for poor design.

Methods with more than 7 statements are considered candidates for review.

LOC is a measure which does not take coding style into account. C++ methods with 24 lines or more should be reviewed.

An average method size of no more than 9 message sends is recommended.

Metrics for method internals include MCX, method complexity, and SMS, strings of message sends.

Object oriented methods are shorter, have no case statements, and fewer IF statements than functional programs. Consequently, traditional measures of complexity do not apply. Lorenz uses a system of assigning weights which have been derived through experience to compute method complexity. He considers it a practical though temporary measure until a better solution is found. The weights are as follows:

- API calls 5.0
- Assignments 0.5
- Binary expressions 2.0
- Keyword messages 3.0
- Nested expressions 0.5
- Parameters 0.3
- Primitive calls 7.0
- Temporary variables 0.5
- Unary expressions 1.0

Based on past experience, a threshold of 65 is used for MCX.

SMS is a proposed metric to help in assessing a method's ability to recover from errors. When messages are strung together, the possibility of trapping errors may be compromised. This metric has not been empirically tested and no quantitative recommendations are offered.

Class size metrics include PIM, the number of public instance methods; NIM, the number of instance methods; NIV, the number of instance variables; NCM, the number of class methods; and NCV, the number of class variables.

The number of public instance methods, PIM, is a good indication of the amount of work being done by the class. A threshold of 20 is suggested.

The number of instance methods, NIM, can be used as an indicator of classes which should made smaller since too many responsibilities have been assumed. A threshold of 40 for user interface classes and 20 for other classes is suggested. An average of 25 for user interface classes and twelve for other classes is suggested.

The number of instance variables, NIV, is recommended threshold of 9 for user interface classes and 3 otherwise. Reuse is facilitated by a low value.

NCM, the number of class methods, is the number of methods for use within the class. A threshold of 20% of NIM is suggested.

The number of class variables or objects common to all instances of a class, NCV, are suggested a threshold of 3 and an average less than 0.1.

Class inheritance metrics include HNL, the hierarchy nesting level; the number of abstract classes; and MUI, multiple inheritance.

The hierarchy nesting level, HNL, is suggested to be limited to 6. Deep nesting makes testing more difficult.

The number of abstract classes is suggested a threshold of ten to fifteen percent.

Multiple inheritance, MUI, is a flag to indicate its presence. It is not recommended due to the extra effort required to understand a class and the risk of name collisions.

Method inheritance metrics include NMO, the number of methods overridden; NMI, the number of methods inherited; NMA, the number of methods added; and SIX, the specialization index.

Larger numbers (> 3) of the number of methods overridden metric, may indicate subclassing by convenience, something to be avoided.

The number of methods inherited, NMI, is an indicator of good subclassing. A high value is recommended.

The number of methods added has a threshold on a declining scale based on the class hierarchy nesting level. Empirical results are not detailed. The author suggests at least one and no more than 4 when the nesting level is at the suggested maximum of 6.

The specialization index, SIX, is computed as:

$$\frac{NumberOfOverriddenMethods * ClassHierarchyNestingLevel}{TotalNumberOfMethods}$$

Lorenz uses 15% as an anomaly threshold. This metric is intended to promote subclassing where a new type of object is-a superset of the superclasses.

Metrics for class internals include CCO, class cohesion; GUS, global usage; PPM, parameters per method; FFU, friend functions; FOC, function-oriented code; CLM, comment lines per method; PCM, percentages of commented methods; and PRC, problem reports per class.

The purpose of the class cohesion metric is to aid in identifying classes that should be split up. No actual metric is proposed.

Global usage should be minimized. Lorenz suggests justifying any globals other than one system global required to bootstrap the system at start-up..

Based on project results, an upper threshold of 0.7 parameters per method is suggested. Gross differences in the use of parameters between classes may suggest different design styles.

The use of friend functions for anything other than mathematical operators should be justified.

The use of function oriented code is considered undesirable and should be justified.

There should be at least one comment line per method.

At least 65 percent of methods should have a descriptive comment.

If more than one problem per contract or two per class is reported, a closer look at the situation is suggested.

Metrics for class externals include CCP, class coupling; CRE, class reuse; and NCT, number of classes thrown away.

No empirical results are available for CCP which is intended to indicate the degree of collaboration with other classes. Reuse encourages lower levels of coupling and inheritance encourages higher levels of coupling.

No numbers are recommended for CRE at the time of writing, only that there should be some reuse in every project, and each project should produce some reusable classes.

Classes being thrown away (NCT > 0) is a sign that the iteration process is refining the design. If none are being thrown away then it is possible that functionality is simply being added. Lorenz believes that an application under development will undergo refinement which will result in the restructuring of classes so that some classes are thrown away. If no classes are thrown away, then the developer may simply be patching and not thinking about reuse, compromising the potential maintainability of the system.

3.8.3 Conclusion

Lorenz covers many aspects of object oriented software development with his metrics. His recommendations as to thresholds appear to be intuitive. Inexperienced developers would probably benefit from following his guidelines, provided that their domain of application is not much different. Metrics use is always tempered by experience and the local development environment. Developers would probably refine his recommendations for their own purposes as their experience grows.

One would expect that a project where every metric is at or near the suggested threshold would not be ideal. Some guidelines in that regard would be useful.

3.9 Object Oriented Metrics of Ebert and Morschel

Ebert and Morschel focus on metrics which can be used as quality indicators during the development process[13].

Correcting faults during module testing and code reading is at least ten times less expensive than making the same corrections at system test time. Twenty percent of modules contain forty percent of errors, of which sixty percent can theoretically be detected before system integration. Therefore, one can predict the early detection of twenty four percent of faults yielding a twenty percent cost reduction for fault correction.

Metrics are divided into five categories: volume, structure, cohesion and coupling, inheritance, and class organization. An empirical study was conducted comparing metrics with human performance.

3.9.1 Volume

The number of object attributes or instance variables per class makes one metric. The number of methods per class is another, with a suggested maximum of 30.

3.9.2 Structure

Message passing is suggested a limit of 30, cyclomatic complexity a limit of 5, and nesting depth within methods a limit of 5, for each method. The number of parameters per method and the number of temporary variables per method are also measured. Structure metrics are helpful in forecasting debugging and testing effort.

3.9.3 Cohesion and Coupling

The existence of a cohesion violation, a variable access that does not use an accessor method, is noted. The number of external messages passed per method is suggested a limit of 5. A Smalltalk specific metric which counts the number of external accesses of private methods is suggested a limit of 5.

3.9.4 Inheritance

The number of predecessors is limited to 5. The number of successor classes has a minimum of 2 and a maximum of 10. The number of changes to inherited methods is recommended 0 though a limit of 5 is given.

3.9.5 Class Organization

These metrics are intended to capture comprehensibility. The number of characters of an identifier should be at least 7. There should be about one comment line for every five lines. The editing distance of identifiers should be at least 3.

3.9.6 Discussion

Ebert and Morschel provide an example of metrics use. Their set of metrics is smaller yet they report a reduction of faults. It serves as evidence that metrics can help raise the quality of object oriented software.

3.10 Object Oriented Metrics of de Champeaux

In [12], Dennis de Champeaux discusses many metrics. He categorizes them in different ways, notably as they apply to the analysis, design, and implementation phases.

3.10.1 Analysis Metrics

Function point analysis is a popular choice for estimating development effort. However, it uses historical data which is based mostly on MIS applications, and is not recommended for estimating OO project metrics.

Analysis Effort Estimation

The following terms are used:

t_0 = start time of the analysis
t_e = end time of the analysis
T = a time "now " between the start and end
$A(T)$ = actual development cost from t_0 to T

$B(T)$ = estimated development cost from T to t_e

$C(T) = A(T) + B(T)$, effort cost function

The functions $A(T)$ and $B(T)$ are computed based on a model of the OO analysis process. There are several types of artefacts that must be completed that comprise the analysis. $A(T)$ consists of the sum of completed efforts measured in person-hours. $B(T)$ is the sum of predicted artefact effort requirements.

The set of artefact types is {Use Case, Scenario, Subsystem, Vocabulary, Inheritance Diagram, Subsystem Diagram, Class-Relationship Diagram, Class, Relationship, Instance, Scenario Diagram}. There are dependencies between artefacts which are used to estimate the effort requirements of incomplete artefacts using known or already estimated efforts.

The set of dependencies is as follows:

< Scenario,Use Case >

< Vocabulary, Scenario, Subsystem, Requirements Document >

< Inheritance Diagram, Vocabulary >

< Subsystem Diagram, Vocabulary, Subsystem >

< Class Relationship Diagram, Vocabulary, Subsystem Diagram >

< Class, Inheritance Diagram, Vocabulary, Class Relationship Diagram >

< Relationship, Class Relationship Diagram, Class >

< Instance, Vocabulary, Class, Relationship >

< Scenario Diagram, Scenario, Class, Instance >

The functions used to estimate one artefact based on the values of others is determined from historical data in the implementation environment.

Artefacts which are partially complete can be estimated as the sum of the time spent and the remaining fraction multiplied by the estimate.

The development of a new artefact is, statistically, not a linear function of its size, but rather a quadratic or exponential one.

Discussion

By including a set of dependencies in the documentation process, de Champeaux is able to predict effort with a finer granularity than simply counting classes, for instance. His approach is based on his OO analysis model which relies in part on Use Cases, a proven methodology.

Analysis Artefact Metrics

Theory

A metric is a function that maps from the artefact domain to the domain of positive real numbers. Certain characteristics are desirable but not necessary for a metric. Chidamber and Kemerer refined those of Weyuker.

Order property

We would like our metric to correspond with our intuitive sense of ordering, so that a value for artefact a_1 would be less than that for a_2 if that is what our intuition tells us.

Relative metric

The numeric values of our metric do not have an absolute interpretation. This makes it easy to scale metrics according to their influence when combining them.

Artefact separation

Different artefacts will generally produce different values. Weyuker calls this *granularity* and Chidamber and Kemerer call it *non-coarseness.*

Incidental collusion

It is acceptable if two different artefacts happen to score the same on a particular metric.

Artefact composition

In certain cases, it may be meaningful to combine measures. We expect the combination of two parts to be at least as large as the artefacts by themselves. We must be careful to avoid combining measures of components which are identical in some way.

Artefact structure

Weyuker requires that the metric of an artefact change when program statements are permuted.

Artefact renaming

Weyuker argues that renaming an artefact should not affect its complexity.

Form versus meaning

Metrics depend on the syntactic and not the semantic aspects of artefacts.

Monotonicity

The values of an artefacts metrics do not decrease when something is added to the artefact.

Composition instability

If two artefacts have the same value of a particular metric, adding the same thing to each of them may not result an equal change in the metric.

Metrics

Use case

The number of sentences that describe the use case is the metric.

Scenario

Scenarios are structurally the same as use cases and the same metric applies.

Subsystem

One metric is to count the number of subsystems, so each subsystem gets a value 1.

Another involves quantifying template attributes such as "number of children subsystems" and combining them in a linear polynomial with appropriate weighting coefficients.

Vocabulary

The vocabulary is a central artefact and describes relationships, object classes, ensemble classes, (parametric) relationship instances, (parametric) objects, and ensembles (which correspond to subsystems). The metric design can be similar to that of subsystems. Either counting the number of entries, or exploiting fields that can be quantified.

Inheritance diagram

A metric for the inheritance diagram can be constructed by counting the number of nonroot nodes in the inheritance lattice, and if multiple inheritance is used, weighting each node by an exponential function.

Subsystem diagram

Two metrics are offered. The first is a weighted sum of the number of nodes and the number of arcs. The second consists of the sum of the number of links each subsystem has with its peers. No empirical data exists to decide which is more informative.

Class relationship diagram

The graph consists of relationship nodes and class nodes, where the arity of a relationship is at least 2. One metric involves the counting of class nodes and adding the arity of each relationship. A more sophisticated measure would involve weighting the contributions of the class nodes and the relationship nodes.

Class

Several metrics apply to classes. Notably the attribute count, the number of class constraints, the number of disjoint transition networks, and the numbers of event and service requests in the class's transitions.

Depth of Inheritance Tree This metric is defined as the distance from the class to the root. In the case of multiple inheritance, the longest path is used.

Number of Ancestors This metric is a count of the ancestors. In the case of single inheritance, this is the same as the previous metric. It provides a more accurate indication of the class's dependencies on other classes in the case of multiple inheritance.

Number of Children This is defined as the number of immediate subclasses.

Number of all Children This metric counts all immediate and indirect subclasses.

Coupling Between Object Classes This is defined as the number of couples the class has with other classes. An instance of coupling is defined as a service request, the sending of an event, the transmission of an instance, or the reception of an instance.

Response For a Class This metric is the size of the set of service requests and events.

Lack Of Cohesion This metric is intended to measure how "splittable" a class is, and its definition is suggested as a topic for research.

Relationship

A metric for relationship can simply be the sum of the arities of the relationships. A more sophisticated approach is to assign a weight to each relationship depending on its cardinality.

Instance

Two metrics are defined. One for the number of stable or enduring objects, and the other for the number of transient objects.

Scenario diagram

A scenario diagram consists of class instances and their transition networks. A measure is to add the number of objects and the number of arcs using weights to compensate for their relative importance.

Discussion

Weyuker's principles are controversial, even after the refinements of Chidamber and Kemerer. In particular the requirement that the metric of an artefact change when program statements are permuted is questionable. If one reverses the order of two initialization statements, has the complexity changed?

Also, the composition of metrics is fraught with pitfalls. Metrics are intended to be used to compare software artefacts. When metrics are composed the reliability is compromised. We are pleased when a metric correlates by a factor of 0.75. If we compose two metrics of similar accuracy then we may be lucky to get a correlation of 0.75 * 0.75, though it could be better or worse.

There is no discussion of experience to support or justify the set of metrics, however it appears to be thorough and the mere documentation process is likely to be of value.

3.10.2 Design Metrics

Design metrics relate more to quality aspects of software.

Effort Estimation

Effort estimation for design can use what is known from the analysis phase. A simple estimation would be to multiply the analysis effort by an empirically determined multiplier. A finer estimation could be determined using the number of classes and relationships.

The level of granularity can be further increased. For instance, an effort estimate could be the weighted sum of the number of transition diagrams. For predetermined constants $\{c_i\}$, an estimated design time for a class could be

$c_1 * (\#attributes) + \sum_{TDaTransitionDiagram} (c_2 * (\#states_in_td)^2 + c_3 * (\#TransitionsInTD))$

Estimating the design effort of other components is similar. Factors affecting design effort include:

1. The use of design tools
2. Domain complexity
3. Domain familiarity
4. Availability of domain specific library components
5. Availability of low level library components

Discussion

The metric of estimated design time hinges on accurately determined constants. There is no specification as to how these constants would be determined and no experience to back up the accuracy of the metric.

Quality Assessment

Size

In general, smaller is better.

Flexibility

Also called robustness, flexibility is a desirable software quality. For instance the ability to add numbers of different types in C++ is an example of flexibility.

Rule based systems are inherently more flexible. No metric to measure flexibility is proposed.

Modifiability

Consistency of descriptions is one attribute which facilitates maintenance. Various tools can improve modifiability. These include version control systems and integrated tools with some kind of hyperlinks.

Modifications happen inside objects, at the object level, and systems wide.

Changes inside objects

Transition diagrams are transformed into functions as part of the design process. Factors affecting the modifiability of a member function include its size, control complexity, and to a lesser degree the number and complexity of arguments, the complexity of a return value, whether a function is inherited, and the number of calls to other member functions.

Changing objects

Changes to objects can be internal, or can affect how they interact in the system.

First we consider objects in isolation. Larger classes are more difficult to grasp than smaller ones. The size of the local members and the size when factoring inherited members can be considered separately. Some proposed measures:

$u^1_{ClassMB}(class) =$ distance of *class* from the furthest root class

$u^2_{ClassMB}(class) = \#$ direct and indirect super classes of *class*

Other metrics can be designed. A desirable feature which is difficult to measure is splitability. Ideally a class should have a minimal size.

Object interaction can be measured in terms of communication connections. A high fan-in/fan-out may suggest that some objects have too much responsibility, especially if cohesion is low. A proposed metric is

$u_{systemContext}(class) = c_1 * u_{fanIn}(class) + c_2 * u_{fanOut}(class) + w(class)$

where $w(class)$ is a value which is high when garbage collection is explicit, such as with C++.

Changing the system

System maintenance is mainly a function of size. The number of classes is one measure. More elaborate approximations can be developed using class characteristics such as class size complexity.

Discussion

The system context metric is a composite metric which combines fan-in and fan-out. A high fan-in implies a class which does too many things whereas a high fan-out implies a class where reuse of other classes has been applied. While de Champeaux notes the importance of the distinction the proposed metric seems insensitive to it, contrary to his stated intention.

Otherwise, he points out important aspects of quality, though no experience using metrics to achieve quality is offered.

3.10.3 Implementation Metrics

Since no single metric always provides all the useful information about an implementation, several are computed.

Metrics incorporated in the CPPSTATS tool used inside IBM include:

Method details LOC in a method, the method's access scope, return type and parameters.

Class instance variables The variable's name, type, and access scope.

Class LOC

File LOC If no class is defined, the LOC of the file is recorded.

Superclass Subclass/superclass relationships.

Method complexity This is the McCabe complexity measure.

An OO tool from McCabe Associates provides another set of metrics:

Encapsulation

- Lack of Cohesion in Methods

- Percent of Public and Protected
- Access to Public Data

Polymorphism

- Percent of Non-overloaded Calls
- Weighted Methods per Class
- Response for a Class

Inheritance

- Number of Roots
- Fan-in
- Number of Children
- Depth

Quality

- Maximum Cyclomatic Complexity of Methods in a Class
- Maximum Essential Complexity of Methods in a Class
- Number of Classes Dependent on Descendants

Effort Metrics

If design efforts have been recorded for each class, an estimate of implementation effort could be obtained with the function:

$efes\,(ic) = u_1 * efm\,(dc) + u_2 * pr\,(dc) + v$

where ic is the implementation class, dc is the design class, $efm\,(dc)$ is the actual effort used to develop dc, $pr\,(dc)$ is a product metric that takes aspects of the design into account; and u_1, u_2, and v are empirically determined constants.

Product Metrics

Member function metrics

Size of a member function

Different metrics for size exist:

1. LOC
2. The number of statements
3. The number of semicolons in a member function
4. The number of input and output parameters
5. The number of member variables

6. Halstead's length

Complexity of a member function

McCabe's cyclomatic complexity is used.

Fan-in of a member function

This is the number of instances that the function is invoked outside the class.

Fan-out of a member function

This is the number of external objects referred to within the function.

Class metrics

Class attributes

There are different ways to count attributes. We have

locally defined attributes = private + protected + public attributes

total attributes = inherited attributes + friendship attributes + locally defined attributes

or

total attributes = object-valued + non-object-valued attributes

From these secondary metrics can be defined:

attribute closure = (private attributes) / (total attributes)

attribute external dependence = (inherited attributes + friendship attributes) / (total attributes)

attribute OO-ness = (object valued attributes) / (total attributes)

attribute incorporation = (value attributes) / (total attributes)

The existence of public attributes violates encapsulation and should be noted.

Number of static attributes of a class

The static attributes of a class are shared by all instances of the class. The ratio of static versus regular attributes is offered as a possibly useful metric.

Number of member functions of a class

Of interest is the access control, the type of any returned value, and whether the function is inherited, inherited and modified, or locally defined.

Weighted methods for a class

Chidamber and Kemerer suggest the sum of weighted methods as a metric. They are not committed to any particular weighting scheme.

Fan-in of a class

This is the sum of the fan-ins of the methods.

Fan-out per class

This is the sum of the fan-out of the member functions, referred to as coupling by Chidamber and Kemerer.

Response for a class

This is the size of the set of member functions and the member functions that can be invoked directly by them.

Lack of cohesion in methods

This is the difference between the number of member functions that share at least one variable and member functions that share no variable. While it is useful to look at which member functions share which variables, this metric will only trap specific cases and is not generally meaningful.

Number of member function clusters for a class

This metric clusters member functions where any two share a variable. This avoid the problems of the previous metric.

Number of comment lines

This can be counted for attributes, member functions, and classes.

Inheritance metrics

Inheritance depth of a class

This is the distance to the root class. The maximum length is used when there is multiple inheritance.

Number of ancestors of a class

This is equal to the depth when single inheritance is used.

Number of direct subclasses of a class

This indicates how many direct descendants will be affected by modifications to the class.

Number of direct and in-direct subclasses of a class

This measures all descendants.

Implementation system metrics

Number of global enduring instances

These are created during initialization and exist during program execution.

Min, max, median, and average number of attributes per class

Lorenz suggests an average less than 6.

Min, max, median, and average number of member functions per class

Lorenz suggests an average less than 20.

Min, max, median, and average size of member functions

Lorenz suggests an average less than 15 for C++.

Min, max, median, and average complexity of member functions

This number should be minimized.

Min, max, median, and average inheritance depth of the classes

Lorenz suggests a maximum less than 6.

Min, max, median, and average fan-in of the classes

High fan-in classes should be carefully checked for correctness.

Min, max, median, and average fan-out of the classes

High fan-out classes should represent system-specific features.

Number of classes

Of interest are

- Number of reused as-is classes
- Number of reused but modified classes
- Number of newly developed classes

Reuse metrics can be defined:

pure reuse rate = (reused as-is) / (number of classes)

leveraged reuse rate = (reused as-is + modified classes) / (number of classes)

Number of root classes

This metric gives an idea of system size. A derived metric

1 - (number of root classes) / (total number of classes)

gives a perspective on how much inheritance is involved in the system.

Average number of comment lines

Lorenz suggests averages for classes, attributes, and member functions greater than 1.

3.10.4 Discussion

De Champeaux gives us a broad view of OO metrics. He does not refer to any of his own experiences which would help in deciding which metrics are most valuable. This is understandable in part due to the lack of experience using OO metrics in the software industry at the time of writing.

3.11 Object-Oriented Cognitive Complexity Metrics

The most comprehensive model of cognitive complexity metrics to date is by Brian Henderson-Sellers[22]. As of this writing, empirical tests have not yet been completed and implementation specifications are not complete. They serve as an indication of the directions research has taken so far. The following is a summary.

3.11.1 Factors Affecting R: The Complexity of the Immediate Chunk

The complexity of the immediate chunk, R, is a composite of several metrics:

$$R = (R_S, R_C, R_E, R_R, R_V, R_D, R_F)$$

where R_S is the size, R_C is the difficulty of comprehending the control structure the chunk is contained in, R_E is the difficulty of comprehending Boolean expressions contained in the chunk, R_R is the recognizability, R_V is the effect of the visual structure or layout of the program, R_D represents the disruptions caused by dependencies, and R_F is the familiarity that influences speed of recall.

Empirical research is needed to validate this hypothesis.

Chunk Size (R_S)

This metric was is computed as follows

$$\begin{gathered} R_S = aS_i, \text{ if } S_i <= L_{\max}, \\ \text{and } R_S = aS_i + b\left(\frac{S_i - L_{\max}}{L_{\max}}\right), \\ \text{if } S_i > L_{\max} \end{gathered}$$

where R_S is the complexity resulting from chunk size, S_i is the size of the chunk, $L_{\max}$ is some programmer dependent limit on the size of a chunk that may be estimated, and a and b are empirically determined coefficients.

Type of Enclosing Control Structure (R_C)

Complexity varies with the control structure.

$$R_C = \sum_{j=1}^{\inf} C_i^{(j)} p^{2^{j-1}-1}$$

where $C_i^{(j)}$ is the complexity of the ith chunk after the jth iteration and p is the probability of failure after the first iteration.

Difficulty of Understanding Complex Boolean and Other Expressions (R_E)

This metric is defined as

$$R_E = b_1 \sum_{AllBooleanExpressions} B_i$$

where B_i is the number of predicates in the ith Boolean and b_1 an empirical constant. McCabe assumes that each conditional within a Boolean expression adds the same level of difficulty as an entire control structure. This important issue is not settled. Further research including empirical evaluation is needed to better estimate the influence of program complexity.

Recognizability of a Chunk (R_R)

Recognizability is based on the program's conformance with rules of discourse and cohesion.

$$R_R = r_R + r_C$$

where r_R represents rules and r_C represents cohesion. Computing this value is considered to involve a lot of work such as compiling the rules of discourse.

Effects of Visual Structure (R_V)

Chunks that are separable from the rest of the program reduce the difficulty of comprehending a chunk. The metric is defined by

$$R_V = a_1 V$$

where V is one of $\{1, 2, 3\}$ representing three levels of difficulty and a_1, an empirical constant to be evaluated experimentally.

Disruptions in Chunking Caused by Dependencies (R_D)

When tracing a call or definition is required, the current process of understanding is interrupted. The metric's computation is offered as

$$R_D = d \sum_{jinN} C_j + e \sum_{jinN} T_j$$

where N is the set of chunks on which the ith chunk is directly dependent for a given task; T is the difficulty of tracing a particular dependency; and d and e are empirically determined constants.

Speed of Review or Recall (R_F)

The more often a programmer reviews a chunk, the more familiar he is with it and less effort is needed to understand or recall it. For the ith chunk, the recall factor

$$R_{F_i} = \sum_{jinN} f^j$$

where f is a review constant (about 2/3) which represents the speed up of understanding on successive reviews. R_F is the only multiplicative factor.

3.11.2 Factors Affecting T, the Difficulty of Tracing

The difficulty of tracing is defined as

$$T = T\left(T_L, T_A, T_S, T_C, T_F\right)$$

where T_L is the localization of the dependencies, T_A is the ambiguity of the dependency, T_S is the spatial dependency of the dependency, T_C is the level of cueing of the dependency, and T_F is the familiarity of the dependency.

Localization (T_L)

The degree to which a dependency may be resolved locally. Three levels are embedded, local, and remote.

$$T_L = a_2 L$$

where L is one of $\{1, 2, 3\}$, and a_2 is an empirical coefficient to be determined.

Ambiguity (T_A)

Where there are several alternative chunks on which a section of code may be dependent. T_A has value 0 when no alternatives exist, otherwise the value is a_3, an empirical coefficient to be determined.

Spatial Distance T_S

This metric is expressed as

$$T_S = b_2 dS$$

where b_2 is an empirical constant, and dS represents the distance between two chunks for which there is a dependency.

Level of Cueing (T_C)

The name of a procedure is easy to find in the declaration, but may be obscured at the site of a call. The metric is expressed as

$$T_C = a_4 B$$

where B is 1 if there are obscure references, 0 otherwise.

Dependency Familiarity (T_F)

Similar to chunk familiarity, the multiplicative factor is defined as

$$T_{Fi} = \sum_{jinN} f^j$$

where f is the review constant (2/3).

3.11.3 Discussion

Henderson-Sellers' metrics relie heavily on empirically derived constants, yet there is no guarantee that the values that would make the metrics meaningful are in fact constants.

He makes frequent use of composition in computing metrics. For instance R_C, the type of enclosing control structure is a sum of the complexities of chunks multiplied by probabilities. Evidently empirical testing is needed to determine if the metric is meaningful when two different programs have the same value for this metric.

A lot of testing is needed to support the value of the proposed metrics.

3.12 Conclusion

As none of Henderson-Sellers metrics have been tested, it is difficult to draw conclusions about their accuracy in predicting human performance. However, they do provide insight into the kinds of factors that need to be considered. Combining measurements also leads to inaccuracy since no single measure is completely reliable, weakening the potential value of compound metrics and making them more difficult to validate.

Complexity metrics are useful when they predict the fault rate of a program, and guide us to the changes that improve the reliability of software. Complexity measures such as McCabe's cyclomatic complexity and Halstead's effort, show a high correlation with program size and are therefore weak indicators of complexity[33].

The science of software metrics is still immature. The goal of metrics is to estimate human performance, yet human performance can vary from day to day, not to mention from one individual to another. Nevertheless, organizations applying a metrics program have reaped considerable benefits in terms of improved product and productivity[16].

If one is to estimate human performance, then one should restrict ones estimations to humans whose performance is known so that our estimations can be calibrated accordingly. In organizations a given set of experienced programmers will perform in a more predictable way since they will have similar kinds of knowledge and experience.

As far as specifying how to implement cognitive complexity measures, there is little that is well defined. While it may be difficult to predict how long it would take an individual to perform a cognitive task, it is feasible to identify passages in a specification which are likely to be hard to understand, and are therefore candidates for simplification.

Chapter 4

Analysis: Comprehension Constrained Software Engineering

Software development is a thought intensive activity, and the products of software development include many documents which must then be understood. If their understanding can be improved by respecting human performance limitations, then many benefits can be realised. Linking research in comprehension performance to software engineering is an opportunity that should not be missed, therefore we will develop rules for developing software that respect the limits of human comprehension.

Software engineering methodology has evolved according to the experience of software engineers. When a different method has been found to work well in one project, it has been tried by others seeking to improve their productivity. A trend has been that methods which have gained acceptance have improved certain aspects of software comprehensibility which made it easier to create larger and more complex software systems. We will now re-examine the current understanding of human comprehension and see how software development is vulnerable to human limitations.

4.1 Understanding the Comprehension Process

The first limiter of comprehension is short term memory (STM). Short term memory is estimated as being 7 digits or 5 words for about 30 seconds, on average. It is estimated that a human being can maintain 4 chunks in STM[14].

4.1.1 Limiting Errors Due to Statement Size

With such limitations, one would expect that processes involving an increasing number of words or chunks may result in an increasing number of errors on the part of human subjects. This was in fact demonstrated in a study of problem solving by Anderson et. al.[2]. Their study found that arithmetic problems with more digits produced more errors in the responses of the subjects, and these errors increased at a greater than linear rate with respect to the number of digits. This suggests a specification rule:

1) PROGRAM STATEMENTS SHOULD BE AS SIMPLE AS POSSIBLE.

When memory was more scarce, programmers might avoid the declaration of a variable and write statements that would fill a line. Such programs are difficult to decipher. Correctness has a strong association with clarity, therefore statement simplicity is highly desirable when time is short.

4.1.2 Comprehension Affected by Size of Block of Text

A different measure of memory called reading span is measured by putting subjects to read a series of sentences and remember something, usually a word, from each sentence. A study by Just and Carpenter found that individuals vary in performance from 2 to 5.5 sentences[24]. This type of memory was not found to correlate with the subject's ability to remember a list of unrelated digits or words. They found that when STM is loaded with something the subject must remember while performing a comprehension task, comprehension performance was degraded. This suggests that even modestly large specifications can be difficult to understand when the size reaches a subject specific threshold. This suggests a second specification rule:

2) KEEP GROUPS OF STATEMENTS SMALL.

Occasional comments facilitate chunking and reduce the effect of size. This rule also motivates modularization.

4.1.3 Ambiguity Affects Comprehension

Just and Carpenter also found that when an ambiguous word occurs at the end of a sentence, better comprehenders ignored it while poor comprehenders would retain it in memory. It was also found that an ambiguous reference in the text would cause better comprehenders to slow down and arrive at the correct conclusion, while poorer comprehenders would not be slowed down and would be more likely to arrive at an incorrect conclusion. This suggests a third specification rule:

3) LIMIT THE SCOPE OF NECESSARY AMBIGUITY, AND AVOID UNNECESSARY AMBIGUITY.

Compilers warn the programmer of unused variables and unreachable code. These represent a form of ambiguity it is preferable to avoid. A programmer may search for the use of a variable from the point of declaration. If the variable use is not immediate, one may consider that code as having undesirable ambiguity. When code that accomplishes different tasks is mixed, conceptual ambiguity can result.

4.1.4 Complexity Magnifies Differences in Comprehension Skills

Differences in comprehension performance were found to be most pronounced when reading difficult or complex portions of text. This precipitates the fourth rule:

4) AVOID UNNECESSARY COMPLEXITY.

Necessary complexity is determined by the domain. When computers were slower and had little memory, programmers were often preoccupied with making programs do as much as possible in the least space. Hence "obfuscation" became a popular term.

4.1.5 Encapsulate to Improve Simplicity

When two related sentences are separated by unrelated sentences, the text takes longer to read than when all the text is related. This suggests a fifth rule:

5) DO NOT MIX UNRELATED SPECIFICATIONS IN THE SAME BLOCK.

This is one motivator for modularization. For instance, we do not need to know all the low level details of displaying text on the screen, and if we showed them, the intent of the program segment would be obscured in detail.

4.1.6 Optimize Value of Expertise Through Simplicity

Studies, especially by K. A. Ericsson, into how experts use memory found that with experience, experts develop domain-specific retrieval structures which work like

STM[14]. This he refers to as long-term working memory (LT-WM). LT-WM can provide a significant difference in performance between experts and novices, for instance within an experts domain, problem solving occurs at the speed of text comprehension, about ten times faster than problem solving in an unfamiliar domain. It is estimated that about ten years of experience is required to achieve expert performance. Since an important part of the domain of software development is the paradigm that one works with, a sixth rule is derived:

6) USE A GENERAL PARADIGM WHERE PATTERNS IN THE DOMAIN OF APPLICATION CAN BE REPRESENTED.

The use of design patterns as suggested by Gamma et. al. [18] and Pree [35] demonstrate the value of using familiar patterns. Using an abstraction paradigm that encourages the repetition of patterns such as object orientation, entity diagrams, or data flow diagrams, promotes the portability of expertise from one domain to another.

4.1.7 Documentation Should be Adequately Detailed

Such large differences between expert and non-expert performance, and the fact that the higher performance levels are highly domain-specific, can have a significant impact on software development. For instance, when an expert writes a text which lacks background information, other experts will understand it completely, but non-experts will not. This suggests the seventh rule:

7) INCLUDE NECESSARY BACKGROUND INFORMATION IN THE DOCUMENTATION.

Determining what is necessary and what is superfluous is probably domain specific. All users of the documentation should participate in setting specification standards since in many cases, one user does not know what the other is doing.

4.1.8 Documentation Should Include Expert Level Summary

Tests involving high-coherence text, or text with supporting background, and low-coherence text, or text lacking supporting background, with high-knowledge and low-knowledge readers found that low knowledge readers fare better with high-coherence text, whereas high-knowledge readers fared better in some activities with high-coherence text, and fared better in other activities such as problem solving with low-coherence text[27]. This leads to the eighth rule:

8) PROVIDE BOTH NOVICE-LEVEL AND EXPERT-LEVEL DOCUMENTATION.

When documentation is written an expert for himself, a novice will likely be frustrated by the lack of detail. Conversely, when a novice has made the documentation, it may include details that are only useful during the early stages of learning the application. Hypertext provides the means of having more than one level of detail in the same document, and may be a solution.

4.1.9 Applications Should be Domain Oriented

Altmann's study of programmer behaviour found that the programmer's understanding of the program is organized around domain knowledge[1]. This leads to the ninth rule:

9) ORGANIZE THE PROGRAM AND ITS DOCUMENTATION ACCORDING TO THE DOMAIN OF APPLICATION.

Object orientation tends to organize the program around concepts in the domain of application. This enables the programmer to plan the program according to the domain, and to quickly find the code that pertains to a part of the domain of application.

4.1.10 Variable Tracing is the Main Comprehension Activity

A study at George Mason University of software comprehension processes found that programmers made frequent use of variable tracing to construct a model of how a program works[6]. This motivates the tenth rule:

10) MINIMIZE THE TRACING DISTANCE OF VARIABLES (E. G. KEEP DECLARATION AND USES CLOSE TOGETHER).

This rule motivates encapsulation. Encapsulation keeps related data and functions together and limits the tracing required to understand the program segment. This also enforces the principle of not using global variables which represent a worst-case tracing scenario.

4.1.11 Function use Source of Comprehension Difficulty

Just and Carpenter found that when a single concept is associated with two different roles simultaneously, that reader had difficulty in comprehending the text. Functions that take arguments and return values are performing two roles simultaneously. If in addition, the function has side effects, then three roles are associated with the function, and the likelihood of error is increased. This motivated the next two rules:

11) AVOID WRITING FUNCTIONS WITH SIDE EFFECTS.

When a function call results side effects such as the hidden change of the value of a variable, a programmer unaware of this dependency is likely to make an error before they discover it.

12) LIMIT USE OF FUNCTIONS.

For example

```
a = b + c
```

is better than

```
a = add(b,c)
```

Of course, function use is a part of the programmers toolkit, so they must be familiar with it. Function usage can also improve clarity in cases where a code segment would otherwise become large and complicated. They also improve conciseness by providing a means to avoid needless repetition. Nevertheless, one can go overboard with function use. Writing short single-use functions at every opportunity will result in a lot of programmer tracing and increase the time to understand a program, and increase the risk of misunderstanding.

4.1.12 Conclusion

It is clear from these derived rules that comprehension capacity is a significant parameter in the success of software specification practises. There exists a substantial amount of research in comprehension which can aid in evaluating and choosing new and existing software specification languages. Most of all, it helps to understand how there can be so many convincing differing opinions about software methodology when one realizes that experts can be much more efficient in their domain of expertise (e.g. using a particular methodology).

Here is a summary of the derived rules of software development:

```
1)  Program statements should be as simple as possible.
2)  Keep groups of statements small.
3)  Limit the scope of necessary ambiguity, and avoid unnecessary ambiguity.
4)  Avoid unnecessary complexity.
5)  Do not mix unrelated specifications in the same block.
6)  Use a general paradigm where patterns in the domain of application can be
    represented.
7)  Include necessary background information in the documentation.
8)  Provide both novice-level and expert-level documentation.
9)  Organize the program and its documentation according to the domain of
    application.
10) Minimize the tracing distance of variables (e. g. keep declaration and
    uses close together).
11) Avoid writing functions with side effects.
12) Limit use of functions.
```

4.2 Visual Representation of Programs

An experimental investigation of the value of flowcharting techniques at Indiana University published in 1977 concluded that detailed flowcharts are not useful in program composition, comprehension, debugging, or modification [39]. Consequently, flowcharting was ignored by the software engineering community[15].

Scanlan, motivated by his teaching experience with flowcharts, performed an experiment to determine if flowcharts assist in the comprehension of complex algorithms[37]. He found that significantly less time was required to understand algorithms as flowcharts than as pseudocode, and that students understood the algorithms more correctly. He also found that some students comprehended simple algorithms faster with pseudocode, but their number was reduced to zero as the algorithm increased in complexity.

Scanlan's study considered only comprehension and not development. Nevertheless, his results are important in that he measured the time subjects spent using a flowchart or pseudocode in understanding a program. No such measurement was made in the University of Indiana study. Scanlan identifies other weaknesses in their experiment.

This result would suggest that the flowchart should be a standard part of program documentation, to facilitate comprehension.

The success of graphical user interfaces demonstrates that human beings are more efficient when a visual rather than a textual representation is used. Various studies have found that visual or graphical representations of programming languages make abstractions easier to understand [9]. Representing specifications graphically also naturally limits the complexity of the structure in question, since very complex things are messy when represented visually. Complexity in textual specifications is not immediately identified since one must read the specification to realize its complexity.

4.3 Comprehension and Software Metrics

Software metrics such as size/time are used in practise to evaluate human performance. The process of software development necessarily involves the comprehension process.

From the research of expert memory processes, it is apparent that individuals will develop an expertise particular to their experience, will perform an order of magnitude better in tasks that fall in their domain, and will perform in an average way in other areas. This helps to explain why measures such as program-size/time will vary a lot, why metrics programs must be within company, and why quantitative results from one study are not generally portable to other software development environments.

Since programmers think of applications in terms of the application domain, a part of their expertise will be application domain specific. Therefore, placing these otherwise productive programmers into projects outside their domain will not result immediately in the same levels of productivity.

Most errors in software other than syntactical occur due to a lack of comprehension of the domain, or the application environment. Therefore one would expect that metrics which are good indicators of faults are also good indicators of software that is hard to understand. It is known that large specifications are harder to understand than small specifications, so size metrics will indicate fault rate.

More interesting are errors due to cognitive complexity. Defining cognitive complexity completely is difficult. Some complexity will be due to semantic issues in the domain, and the complexity may be due to the programmer's ignorance of the domain. However, some aspects of complexity should be predictable. Another challenge in accurately measuring the effect of complexity, is that a complex code sequence may be embedded in a large program, and its influence may be difficult to identify.

Adding different metrics together, given the nature of human performance variation, is unlikely to produce the desired effect except in special circumstances. For instance, when de Champeaux adds states and transitions in a transition diagram as part of an effort estimation metric, the implicit assumption is that a transition and a state require the same amount of effort. There is also the assumption that larger transition diagrams require a linear increase in effort. There is the additional assumption that two transition diagrams with different number of states and transitions but the same total number of states and transitions require the same effort. Each of these assumptions must be carefully verified with the individuals whose performance is being measured to have the necessary confidence to add these values together.

The accuracy of complexity metrics is related to the granularity of the measurements made. If one would like to identify certain effects of complexity such as time spent understanding complex code sequences, then one must measure the time spent on each program component. This may be an important determinant as to the set of metrics that an organization gathers.

4.4 Development Tools

Software development tools are generally constructed using the latest graphical interface techniques. A tool designer can use the understanding of human thought processes to reduce the cognitive load associated with the use of the tool and searching through the specification, so that the developer has more freedom of thought to understand what he is doing and thinks less about how he is doing it. The executable icon is one example of such an application.

In graphical user interface design, the user's cognitive limitations are respected to ensure that the system will be easy to use. Respecting the developer's cognitive limitations in choosing understandability standards for system specifications, should result in specifications that are as "easy to use" as current knowledge of human limitations allow.

The most meaningful set of metrics will vary depending on the individuals and applications in question. Style can play a significant role in the complexity of software. Measures such as Fan-in and Fan-out of Yourdon and Constantine help identify poorly designed modules. [25]

Paradigm shifts during and in between the phases of software development have been observed to be a source of error and complexity. Ideally, a single methodology would encompass the entire development lifecycle, simplifying the job of the software developer. If this methodology were formally validatable, that would be helpful. If, in addition the specifications are executable and the representations are visual then the methodology is nearly ideal. A product that nearly achieves this ideal in addition to concurrency is a product called ROOM (Real-Time Object-Oriented Modelling) [40].

4.5 Other Implications for Software Engineering

Software specification languages, including programming languages, can be designed and/or chosen according to their suitability to the task and their facility of comprehension. Since language acquisition ability is a factor of memory capacity, the simplicity of the languages used is important. The simplicity refers not only to the language but also to the specifications created with the language.

When functions and types are defined, the language is in fact extended. Therefore abstraction competes with language complexity to reduce overall complexity.

Perhaps a solution to the problems of hierarchies reported earlier that would yield reuse benefits without a great penalty would be to use deep hierarchies only in the portion of the application, such as the interface, where reuse is intense, and broad or no hierarchies elsewhere.

In prototype-based object-oriented languages there are no classes. Objects are created by cloning with no parental ties. Reuse is accomplished by delegation.[20]

4.6 Cognitive Issues in Programmer Productivity

When programmers are in the process of understanding an application, they are processing information and learning how the application is written. If we want to capture a measure of a programmer's cognitive behaviour, we need a model of it.

From earlier discussions, it has been observed that the quantity of information that a human being can retain at a given moment varies between individuals. Memory limitation has been shown to affect an individual's ability to learn language. A program, by defining new variables and functions, is extending the programming language and is therefore new to the programmer. A measure which reflects a programmer's performance would be related to the short-term memory demands associated with understanding an application.

There are a few areas where the complexity that relates to programmer comprehension can be observed: the domain complexity, the problem complexity, the complexity of the language of implementation, and the complexity of the implementation. Programmer performance is then related to a number of independent factors such as their own memory capacity, domain knowledge, familiarity with the language of implementation, related programming experience, and familiarity with the implementation.

Programmer experience is one influence that changes with time. Also, the kind of experience may play a strong role in how efficient a particular programmer is with a particular problem. For instance a network operating system programmer may have trouble with graphical user interfaces, or a business system programmer may have trouble with flight simulator programming. The effects can be more subtle within a particular domain.

It would be useful to have complexity metrics for not only the implementation, but also for the requirements and for the design. This would help in localizing the complexity and possibly reducing or at least controlling it before the implementation phase.

Chapter 5

Analysis: Designing Metrics

A widely used framework for the design of metrics is the goal/question/metric (GQM) paradigm of Basili and Rombach[22]. The first step is to identify the goal, such as to identify error-prone code. Next, appropriate questions must be chosen such as how to limit the likelihood of error in code. From these questions, metrics are defined.

In this case, the goal is to identify error prone code through metrics. We expect that code segments which pose a comprehension challenge will result in a greater error rate. Therefore metrics whose determination is closely related to factors involved in the comprehension process would hopefully predict error.

University examination results represent data of human comprehension performance. We will use them to identify a metric that predicts human error.

5.1 Comprehension Metrics

Boehm-Davis observed that programmers understand a program by tracing references [6]. One metric would involve counting all the traceable references, that is, the number of occurrences of names defined in the scope. This would include variable names, constant names, function names, procedure names, and class names. Another could involve only the names that are associated with a data type, that is, the number of occurrences of variables, constants, and function names where the function returns a value.

Of possible interest is the density of defined names. With a greater density of defined names, chunking would be more difficult since chunks would tend to contain more traceable terms.

For data, we have the results of final examinations on a question by question basis. Results that could be meaningful would include a close association between a high density of traceable terms and a low mean score. Another relationship of interest would be a close association between a large standard deviation and an increasing density of traceable terms. Just,and Carpenter observed that individual differences were most pronounced when the subjects read difficult or complex passages. Therefore, more difficult questions should result in a wider range of marks and a larger standard deviation. If a metric follows the standard deviation then it should be an indicator of complexity.

5.2 An Empirical Study

It would be useful if one could predict code segments which pose a comprehension challenge. This would make it possible to improve the maintainability of code where complex code can be identified and simplified. It could also speed up debugging efforts by finding code segments where errors are more likely. In addition, it may help in the evaluation of the maintainability of legacy software by predicting the likelihood of error, and therefore the debugging time.

In a university we have access to final examinations and the results, and these results constitute measures of human performance. Here we will look at how examination results correspond to various metrics of the solutions.

As we have seen, studies of comprehension processes have shown that human beings can become better at specific activities, such as digit memorization, with practise[14]. Consequently, differences in the material that students are exposed to is likely to create differences in the way classes of students perform. Also, the similarity of the exam questions to class material varies from class to class and year to year.

The hoped for result of this metrics study is to identify one or more metrics that reliably predict error rate. The predictive property that is required in this case is a metric which increases when scores go down.

5.2.1 The Experiment

The course in question is the first programming course in the undergraduate curriculum. The data comes from two sections from each of three semesters. The first semester used a different book and the course took place during a compressed summer term. The two sections of the last semester were different in that one consisted entirely of graduate students seeking to be admitted to a graduate degree program through their performance in the course, the other section consisted of undergraduate students who had a full course load. The middle two sections were mixed graduate and undergraduate students.

The first of the three exams was judged to be the most difficult. The second was much easier, and the third was moderately difficult. The grades of students who did not pass the final were omitted from analysis. Finals were marked one question per marker, as usual, at a time to minimize the effect of marking style differences. The decision to use the exams for this purpose was made after the exams were corrected so neither students nor markers were aware of the experiment.

Due to the textbook, the students who wrote the first exam had a strong preparation in class design and a weak preparation in function design. Students who wrote the second and third exams had a strong preparation in function design and a weak preparation in class design. Teachers had freedom as to how material was covered, consequently different sections received slightly different emphasis on various parts of the material. The exams reflect the course content. The mean and standard deviation are computed.

Since it is the metrics of the solutions which are computed, the influence of style differences cannot be measured. The students are not restricted as to the time they spend on any given question, so the effect of program size is diminished. Thus program complexity factors independent of size should be the dominating theme of students' errors.

5.2.2 Computed Metrics

Since the questions are small, many metrics, such as fan-in and fan-out do not apply. Size metrics such as LOC, and Halstead metrics effort E, difficulty D, volume V and vocabulary are computed and can be found in the appendix.

Several metrics which capture different aspects of computer programs were tested.

EXAM: 1

QUESTION	LOC	E	D	V	ID	MEAN1	STDDEV1	MEAN2	STDDEV2
1ab	36	30980	.038	150	1.00	.78	.18	.79	.16
2a	18	21027	.067	98	1.61	.64	.16	.68	.26
2bc	17	7394	.114	72	1.53	.51	.34	.48	.27
3abc	45	44400	.151	207	.96	.71	.24	.62	.24
4a	10	2034	.095	34	1.00	.88	.14	.83	.21
4b	17	5859	.121	69	1.29	.86	.16	.79	.17
5ab	36	16458	.031	113	1.08	.70	.16	.60	.17

Table 1: Scores and metrics of first exam.

Initially, variable density was tested. The results were interesting but not inspiring. It occurred to me that other identifiers such as function and class names would also require tracing. A new metric, identifier density (ID), the number of identifiers divided by LOC, was tested and found to behave in a more consistent way, frequently predicting when code is easy or hard to understand. This represents the discovery of a cognitive complexity metric through both psychological theory and empirical testing.

Several metrics were tested

5.2.3 Results

First Data Set

The first exam was held during the summer, when the term is compressed from 13 to 7 weeks. The range of experience of the students was from undergraduates with no programming background to people with graduate degrees in other disciplines and some programming background. The course emphasized class design over fundamentals. See table 1.

The tables contain the metrics LOC, E, D, V, and ID as well as the mean and standard deviation for each of the two sections that wrote the exam.

Questions 4a and 4b which have contiguous solutions have means and standard deviations which behave as expected with the metrics LOC and DTN. Namely, higher

EXAM: 2

QUESTION	LOC	E	D	V	ID	MEAN1	STDDEV1	MEAN2	STDDEV2
1a	27	13534	.058	133	1.48	.89	.20	.91	.22
1b	17	7661	.096	77	.94	.86	.26	.95	.17
2	12	7672	.065	63	1.25	.50	.35	.38	.27
3	23	14332	.078	89	1.35	.87	.13	.89	.21
4	20	4531	.060	57	.80	.97	.12	.95	.12
5abc	26	27792	.042	125	1.81	.80	.16	.83	.17

Table 2: Scores and metrics of second exam.

values of the metrics correspond with lower means and higher standard deviations.

Second Data Set

The students for these sections were a mix of undergraduate Computer Science and students with a degree in another subject. The duration of the course was the normal 13 weeks of lecture. A new text was used which emphasized fundamentals more than program design.

The exam was perceived as easy. The results of the second question should be ignored since the correct answer was beyond the scope of the course and the marking did not reflect the students' abilities. See table 2.

Questions 1b, 3 and 4 have a single block of code for their solutions. It is interesting to note that the smallest values of ID occur with the highest mean (4), and the highest values correspond with the lowest score (5abc).

Third Data Set

For this exam, the first class consisted entirely of undergraduate students and the second entirely of students with a Bachelors or higher. Students in the second group were competing for a limited number of graduate diploma positions. The exam was of moderate difficulty. See table 3.

Here again, the highest scores for ID correspond with the lowest mean (3b) and the lowest scores for ID correspond with the highest mean (1c). Questions 1(a), 1(c),

EXAM: 3

QUESTION	LOC	E	D	V	ID	MEAN1	STDDEV1	MEAN2	STDDEV2
1a	8	2688	.130	32	1.50	.82	.38	.75	.38
1b	29	9257	.042	119	.76	.82	.24	.84	.22
1c	26	9057	.040	98	.42	.97	.08	.90	.28
1d	9	3879	.135	72	1.44	.85	.14	.84	.15
2a	30	29700	.032	120	1.37	.93	.15	.83	.28
2b	9	2059	.103	36	1.22	.81	.37	.82	.33
3a	12	15388	.117	88	2.50	.83	.24	.83	.23
3b	11	9206	.111	74	2.64	.78	.23	.64	.27
3c	9	8847	.079	62	2.11	.80	.21	.67	.33
4	35	23471	.044	165	1.34	.90	.12	.87	.19
5	30	26021	.025	127	1.47	.85	.17	.65	.26

Table 3: Scores and metrics of third exam.

2(b). 3(a) ,3(b) and 3(c) have contiguous code.

Analyzing the Data

LOC was not found to predict error in general, as would be expected for such small(10-80 line) programs. The Halstead metrics proved erratic as well, the most promising being E which showed a Pearson correlation in the -.3 to -.5 range on three of the six exams.

ID is most useful when applied to individual methods, as the density of tracable names is most meaningful when considering a particular scope. The metrics correlated well with the third exam with 11 questions. giving -.6279 with p = .039 for the first section and -.6233 with p=.040 for the second. For other exams, the correlations were -.6091 with p=.147 for exam 1, section 1; -.4146 with p=.355 for exam 1, section 2; -.7805 with p=.119 for exam 2, section 1; and -.9358 with p=.019 for exam 2, section 2.

The first exam involved many solutions without contiguous code. This may explain in part why the metric was not as accurate in predicting error.

```
void perfect(int p)
{
  int sum=0,n=1;
  while (n <= p/2)
  {
    if ((p % n) == 0)
      sum += n;
    n++;
  }
  if (sum == p)
    cout << "Input number is perfect" << endl;
  else
    cout << "Input number is not perfect" << endl;
}
```

Figure 1: Solution of question 4, exam 2.

Understanding the Results

It may be instructive to look at programs which rated easy and those which rated difficult. We will look at an easy solution and a difficult solution.

The solution in figure 1 comes from the second exam, and solves question 4.

The solution requires reasoning with three variables, sum, n, and p. It is not a difficult solution. There are 16 defined names over 20 lines of code.

The solution in figure 2 comes from question 3(b) of the third exam, and was the most difficult of that exam.

This solution has a nested loop and involves copying data elements from one array location to another. It is obviously more complex than the easy solution. There are 29 defined names over 11 lines of code.

5.2.4 Conclusion

ID, the density of tracable names is interesting since it is designed based on the study of how programmers think, it often predicts good or poor performance in final examinations, and it is relatively easy to compute. If one is to make a recommendation

```
void Transpose(int **X, int n)
{
  int i,j,t;
  for(i=0;i<(n-1);i++)
    for(j=(i+1);j<n;j++)
    {
      t=X[i][j];
      X[i][j]=X[j][i];
      X[j][i]=t;
    }
}
```

Figure 2: Solution of question 3(b), exam 3.

as to how this metric can be used, one can notice that most solutions with a ID greater than 1.50 resulted in averages lower than 0.80 and relatively high standard deviations suggesting they are of higher complexity.

Style was not an issue in this experiment, though it can easily be significant in software comprehension. Further testing is needed to establish the utility of the metric in industry. It may prove to be a good indicator of fault rate and debugging time.

The results of the experiment should be of interest to psychologists researching comprehension. They may wish to test this metric in general text comprehension experiments. Such a study could yield information on how to structure text to minimise error and improve learning.

This metric is supported by cognitive theory and empirical study. As such it has the priviledge of have a theoretical basis. It may prove to be at least as valuable and less domain sensitive than metrics such as LOC, which would give it credibility. It may also be possible to use this metric without extensive data gathering since it is sensitive to differences in domain and style.

Chapter 6

Future Directions

Experiments with the kind of measurement of time and observation of what the subject is viewing as employed by Boehm-Davis [6] and Scanlan [37] could probably yield more interesting and accurate results than those possible with final examinations.

Simulation of thought processes could predict with greater precision how and where errors occur, could be used to identify code which is likely to produce error, and help programmers understand unfamiliar code by identifying comprehension requirements such as global declarations.

It would be very useful to have a virtual programmer which could report information such as the knowledge required to be memorized in order to understand a given module. This would make it possible to set standards that would control the human memory requirements associated with a set of modules, and thereby improve human performance.

6.1 Cognitive Modelling

Addressing the issue of cognitive complexity will be greatly improved when thought processes are modelled in software, so that one can test models of thinking by implementing them and running them on programs for which empirical data exists. In this way, refinement as to the description and identification of complexity in software will move forward with more tangible points of reference, namely, the cognitive models.

This sets the stage for what might be called the thought-space or code-space that a programmer can work with. On the one hand, the capacity of short term memory

(STM) is a limiter. On another, the language skill is a factor. The language skill is a function of memory capacity, previous knowledge, and time spent working with the language. The program a programmer can understand is a factor of variables such as the programmer's STM capacity, his related knowledge, his language skills, and the time available for the task.

Domain knowledge is a factor, as to the complexity of a program in the sense that a more complex problem involving more domain knowledge results in a more complex program. Ideally, a program's complexity at the highest level should not exceed the complexity of the problem, and should mirror the problem as closely as possible. This allows the programmer to think about the program in terms of the domain without paradigm shifts.

The time available is a factor in the same way it causes a variation in performance in timed exams. If the exams had no time limit, more students would answer more questions correctly. In a practical situation, the same individuals are likely to produce more errors when time is constrained. Reducing the complexity of the implementation can thus reduce the error rate when time is constrained.

With so many independent factors affecting the performance of programmers, if measures are to be at all reliable, then some variables must be controlled. In practical situations many factors will be constant. The set of implementations to be tested can be restricted to the same domain, programming language, and programmers. The remaining variables are then the problem complexity and the implementation complexity. Ideally, one should have separate complexity measures for the domain aspects and the implementation aspects. Implementation complexity which is independent of domain complexity may be possible to reduce, while domain complexity may be fixed. Additionally, knowledge of the domain complexity can be used to choose people to work on a project. Some programmers may be particularly familiar with certain important domain aspects, while others may be familiar with certain important implementation details. Such factors can make a big difference when choosing team members to work on a given project, and can influence performance metrics.

Many metrics measure a combination of domain complexity and implementation complexity. Since these may not be equally related in all applications, the relationship of such measures to human performance is less reliable.

6.2 Directions in Computer Languages

Languages that facilitate more expression with respect to software architecture are evolving. The language Genoa studied at Stanford involves the use of situation theory to improve the flexibility of high-level components. Finding a balance between simplicity and expressiveness that minimizes complexity is easier when we understand more about simplicity and complexity.

6.3 Directions in Cognitive Complexity Metrics

Studies in programmer behavior which observe tracing activities and record the time spent on each name may reveal trends as to the relative importance of function names versus variable names and so on. This could result in more precise comprehension metrics.

Inter module tracing could be linked to intermodule dependencies. Measurement of this effect could help in defining intermodule metrics. If the usual tracing behavior is measured at the same time and a sufficient variety of programmers are tested, then we may be able to define intermodule metrics which can be added to metrics such as ID defines in this thesis.

Domain complexity metrics could also be highly useful, as well as structuring techniques such as object orientation to simplify domain knowledge.

Bibliography

[1] Erik M. Altmann, Modeling Episodic Indexing of External Information, Cognitive Science, (in press), Nov. 1997.

[2] John R. Anderson, Lynne M. Reder, and Christian Lebiere, Working Memory: Activation Limitations on Retrieval, Cognitive Psychology Vol. 30, 221-256, 1996.

[3] Faheim Bacchus, Qiang Yang, Downward refinement and the efficiency of hierarchical problem solving, Artificial Intelligence, Elsevier Science, 1994, pp. 43-100.

[4] James M. Bieman, Metric Development for Object-Oriented Software, Software Measurement, International Thompson Computer Press, 1996.

[5] Deborah A. Boehm-Davis, "Software Comprehension", Handbook of Human-Computer Interaction, Elsevier Science Publishers B. V., 1988, pp.107-121.

[6] Deborah A. Boehm-Davis, Jean E. Fox, Brian H. Phillips, Techniques for Exploring Program Comprehension, Empirical Studies of Programmers: Sixth Workshop, pp3-37, Ablex Publishing, 1996.

[7] L. Briand, C. Bunse, J. Daly, C. Differding, An Experimental Comparison of the Maintainability of Object-Oriented and Structured Design Documents, IEEE Conference on Software Maintenenance, 1997, pp. 130-138.

[8] S. N. Cant, D. R. Jeffrey and B. Henderson-Sellers, A Conceptual Model of Cognitive Complexity of Elements of the Programming Process, Information and Software Technology, 1995, pp. 351-362.

[9] Dimitris N. Chorafas, Visual Programming Technology, McGraw-Hill, 1997.

[10] K. Cox, D. Walker, User Interface Design, Prentice Hall, 1993.

[11] John Stephen Davis and Richard J. LeBlanc. A Study of the Applicability of Complexity Measures, IEEE Transactions on Software Engineering, Vol. 14, No. 9, 1988.

[12] Dennis de Champeaux, Object-Oriented Development Process and Metrics, Prentice Hall, 1997.

[13] Christof Ebert, Ivan Morschel, Metrics for Quality Analysis and Improvement of Object-Oriented Software, Information and Software Technology, vol. 39, pp497-509, 1997.

[14] K. Anders Ericsson and Walter Kintsch, Long-Term Working Memory, Psychological Review, 1995, Vol 102, No. 2, pp. 211-245.

[15] Norman Fenton and Shari Lawrence Pfleeger, Science and Substance: a Challenge to Software Engineers, IEEE Software, July 1994, pp. 86-95.

[16] Norman Fenton, Shari Lawrence Pfleeger, Software Metrics: A Rigorous and Practical Approach, International Thomson Computer Press, 1996.

[17] Wilbert O. Galitz,User-Interface Screen Design, QED Publishing Group,1993.

[18] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns: elements of reusable object oriented software, Addison-Wesley, 1994.

[19] Carlo Ghezzi, Mehdi Jazayeri, Dino Mandrioli, Fundaamentals of Software Engineering, Prentice Hall, 1991.

[20] Bob Hallman, Are Classes Necessary?, Journal of Object-Oriented Programming, September 1997, pp. 16-21.

[21] Maurice H. Halstead, Elements of Software Science, Elsevier North-Holland, Inc., 1977.

[22] Brian Henderson-Sellers, Object-Oriented Metrics: Measures of Complexity, Prentice Hall, 1996.

[23] P. Hsia, A. Gupta, C. Kung, J. Peng, S. Liu, A Study on the Effect of Architecture on Maintainability of Object-Oriented Systems, IEEE Conference on Software Maintenenance, 1995, pp.4-11.

[24] Marcel Adam Just and Patricia A. Carpenter, A Capacity Theory of Comprehension: Individual Differences in Working Memory, Psychological Review, 1992, Vol. 99, N0. 1, pp. 122-149.

[25] Stephen H. Kan, Metrics and Models in Software Quality Engineering, Addison-Wesley Longman, Inc. 1995.

[26] J. Kim, F. J. Lerch, Towards a Model of Cognitive Process in Logical Design: Comparing Object-Oriented and Traditional Functional Decomposition Software Methodologies, CHI '92, 1992.

[27] Walter Kintsch, Comprehension: a paradigm for cognition, Cambridge University Press, 1998.

[28] B. A. Kitchenham and J. G. Stell, The danger of using axioms in software metrics, IEE Proceedings - Software Engineering, Vol 144, No. 5-6, October-December 1997.

[29] Thomas S. Kuhn, The Structure of Scientific Revolutions, Chicago: University of Chicago Press, 1962.

[30] Mark Lorenz, Jeff Kidd, Object Oriented Software Metrics, Prentice Hall, 1994.

[31] Austin Melton, editor, Software Measurement,International Thomson Computer Press, 1996.

[32] K. H. Moller and D. J. Paulish, Software Metrics: A practitioner's guide to improved product development, IEEE Press, 1993.

[33] John D. Musa, Anthony Iannino, Kazuhira Okumoto, Software Reliability: Measurement, Prediction, Application, McGraw Hill, 1987.

[34] P. R. Newsted, The Principle of Chunking in Programming and Design: a Threshold for Acceptable Complexity, University of Calgary, 1977.

[35] Wolfgang Pree, Design Patterns for Object-Oriented Software Development, Addison-Wesley, 1995.

[36] Stephen K. Reed, Cognition, Brooks/Cole Publishing Company, 1996.

[37] David A. Scanlan, Structured Flowcharts Outperform Pseudocode: An Experimental Comparison, IEEE Software, September 1989, pp. 28-36.

[38] Stephen R. Schach, Software Engineering, Richard D. Irwin, Inc., and Asken Associates, Inc., 1993, pp. 38-40.

[39] Ben Schneiderman, Richard Mayer, Don McKay, and Peter Heller, Experimental Investigations of the Utility of Detailed Flowcharts in Programming, Communications of the ACM, June 1977, Vol 20, pp. 373-381.

[40] Bran Selic, Garth Gullekson, and Paul T. Ward, Real-Time Object-Oriented Modeling, John Wiley & Sons, 1994.

[41] Mary Shaw, David Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice-Hall, 1996.

[42] David Steier, Automating Algorithm Design within a General Architecture for Intelligence, Automating Software Design, Lowry and McCartney editors, MIT press, 1991, pp. 577-602.

[43] M. Tambe and P. S. Rosenbloom, The Soar Papers, vol. 2, The Problem of Expensive Chunks and its Solution by Restricting Expressiveness, The MIT Press, 1993.

[44] Rebecca Wirfs-Brock, Brian Wilkerson and Lauren Wiener, Designing Object-Oriented Software, Prentice Hall, 1990.

Appendix A

Examination Questions

First Exam

Question 1. (Arrays and strings.) The declaration of the class Person is shown below.

```
class Person {
    public:
        char *GetName();          // Returns the person's name
        int VowelsInName();       // Returns the number of vowels in the name
    private:
        char name[25];
};
```

The main program sets up and initializes an array of 20 Persons to represent a group (the initializing code is not shown):

```
void main () {
    Person group[20];
    ..... // Code to initialize the group
}
```

(a) [10%] Implement the member function `int VowelsInName()` that counts and returns the number of vowels in the name of the person. The vowels of English are 'a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O', and 'U'.

(b) [10%] Write a code segment which finds the name with the most vowels in the group array, and displays it on the screen.

Question 2. (Coding)

(a) [8%] The code below is a partial declaration for a class of matrices.

```
class Matrix {
    public:
        Matrix (int n);
    private:
        int M[20][20];
};
```

The constructor creates a matrix of size $n \times n$. You can assume that the argument corresponding to n will never be greater than 20. Write a definition for the constructor, given that the elements of this class Matrix are as follows. By default the element M_{ij} of the Matrix M is zero. If the sum of i and j is divisible by i, then add 1 to the element M_{ij}. If the sum of i and j is divisible by j then add 2 to the element M_{ij}. Perform these calculations for all the elements M_{ij} of the matrix M, where $i = 0, 1, \ldots, n-1$ and $j = 0, 1, \ldots, n-1$.

(b) [8%] What value will p have after the following code has been executed?

```
int p = 0, m = 15, n = 7,a=2, b=1;
while (m || n)
{
    if (m % a ==  b)
    {
        p = p + b;
        m -= b;
    }
    if (n % a == b)
    {
            p = p + b;
            n -= b;
        }
        b = a;
        a *= 2;
    }
```

(c) [4%] What does the code in part (b) compute for general values of m and n?

Question 3. (Design and implementation) Design and implement a class for storing and manipulating dates. A user of the class must be able to:

(a) [6%] set the value of a date,

(b) [6%] display a date in the format "21 Dec 96", and

(c) [8%] decrement the date by one day.

Use a constant array with the value

$$\{31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31\}$$

to store the number of days in each month. Note, however, that February has 29 days in a leap year. The year Y is a leap year if Y is a multiple of 4 but is not a multiple of 400. For example, 1900 was a leap year but 2000 will not be a leap year.

Question 4. (Coding)

(a) [10%] Write initialization statements and a while statement that have the same effect as the following statement:

```
for (j = x; j >= y; j--) {
    cout << x * ( j + y) << '\n';
```

(b) [10%] Write a program that uses a `for` statement to compute the sum

$$displaystylefrac11 + \frac{1 \cdot 2}{2 \cdot 2} + \frac{1 \cdot 2 \cdot 3}{3 \cdot 3 \cdot 3} + \frac{4!}{4^4} + \ldots + \frac{n!}{n^n}$$

Question 5. (Design and implementation) Design and implement a class called `InventoryItem`.

(a) [10%] For an `InventoryItem` object, one should be able to find out the quantity of the item, buy items (increment the quantity), and sell items (decrement the quantity). Declare the member functions that enable the user to perform these actions and then write definitions for them. You should also declare the necessary data members for the class.

(b) [10%] There is a minimum and maximum quantity for an `InventoryItem` object. When the quantity falls below its minimum, an order is placed to restore the quantity to the maximum. Add the necessary member data and member functions to make this possible.

— End of Examination —

Second Exam

Question 1. (Reading code.)

(a) [10%] Write down the output that would be obtained by running the following C++ program.

```
#include <iostream.h>

void f(int & x, int y) {
   int t = x;
   x = y;
   y = t;
   cout << "f(int ,int) says x is" << x << endl;
   cout << "f(int ,int) says y is" << y << endl;
   return;
}
 void f(int x, int y, const int z} {
    if (x > y) f(y, x-z);
        else}  f(x, y-z);
    return;
}

int main()  {
    int i = 5;
    int j = 16
    int k = 6;
    cout << " f(int, int) is called with values" << i << j << endl;
    f(i,j);
    cout << " f(int, int, const int) is called with values"
         << k << j << i << endl;
    f(k, j, i);

}
```

(b) [10%] Write down the output that would be obtained by running the following C++ program.

```
#include <iostream>

int main() {
int number = 161;
int rem = 0;
int index = 0;
cout << "the input number is" << number << endl;
   if (number % 2 = 1) {
     rem = 1;
     number = number -1;
   }
   while (number % 2 = 0) {
     number = number % 2:
     index = index + 1;
   }
cout << "index is" << index << "multiplier is" << number
     << "aader is" << rem << endl;
}
```

Question 2. [20%] (Iteration.) Write a program that uses a `while` statement to compute the sum

$$1 - \frac{1}{2} + \frac{1}{3} - \cdots + (-1)^n \frac{1}{n}.$$

such that the value of sum is accurate to three places of decimals.

Continued . . . .

Question 3. [20%] (Arrays.) Implement a function `int Outlier(double list[10])` that expects an array of positive real numbers, and returns 1 if there is at least one number in the array that "significantly deviates" from the other numbers in the array. We say that x_i "significantly deviates" from the set of positive real numbers $x_0, \ldots, x_9$ if the difference between x_i and the average of the set is larger than the average itself:

$$x_i - \frac{1}{10}\sum_{j=0}^{9} x_j > \frac{1}{10}\sum_{j=0}^{9} x_j$$

If there is no such number in the set (i.e., all the numbers are between 0 and twice the average), the function returns 0.

Question 4. [20%] (Program.) A positive integer n is said to be "perfect" if the sum of its proper divisors equals the number itself. (Proper divisors include 1 but not the number itself.) For example, 6 is a perfect number, since the proper divisors of 6 are 1, 2, and 3, furthermore, 6=1+2+3.

Write a program that reads a positive integer, and determines if the integer is perfect. If the input is a perfect number then the program should print out a message "Input number is perfect", otherwise it should print out a message "Input number is not perfect". You can assume that the input integer is greater than 1 and less than 32500.

Continued . . . .

Question 5. [20%] (Class.) Class `Point` is designed to represent points on the plane. A point on the plane is represented by its two coordinates, x and y. The class declaration is the following:

```
class Point
{
    public:
        Point(double xi, double yi);
        void Rotate(double arc);
        friend double Dist(Point p1,Point p2);
        double x;
        double y;
};
```

(a) [4%] Write a definition for the constructor. The constructor initializes the data members so that $x = xi$ and $y = yi$.

(b) [8%] Write a definition for the friend function `double Rotate(double angle)`. The function should rotate the point by *angle* degrees. The coordinates x' and y' of the rotated point are given by the following expressions:

$$\begin{aligned} x' &= x\cos(angle) + y\sin(angle), \\ y' &= -x\sin(angle) + y\cos(angle). \end{aligned}$$

You can use the library functions `double cos(double d)` and `double sin (double d)` to compute the sine and the cosine of an arc.

(c) [8%] Write a definition for the member function `double Dist(Point p1,Point p2)`. The function should calculate the distance between two points. The distance of the two points $p_1(x_1, y_1)$ and $p_2(x_2, y_2)$ is given by the following formula:

$$d(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

You can use the library function `double sqrt(double x)` to compute the square root of a number.

Third Exam

Question 1. (Program Tracing)

(a) [4%] What value will m have after the following code has been executed?

```
int m = 35, n = 14;
while (m != n)
{
    if (m > n)
        m = m - n;
    else
    n = n - m;
}
```

(b) [4%] What is the output produced by the following program?

```
#include <iostream.h>
int i;
void f();
void main(){
cout << i << endl;
i = 5;
f();
}
void f() {
cout << i << endl;
char i;
i = 'a';
cout << i << endl;
cout << ::i << endl;
::i = 3;
{ cout << i << endl;
```

```
  int i = 9;
  cout << ::i << endl;
  cout << i << endl;
}
cout << i << endl;
cout << ::i << endl;
cout << i << endl;
::i = 90;
cout << i << endl;
cout << ::i << endl;
}
```

(c) [4%] Give the exact range of values of `X` for which the following code segment prints the string `Belgian` .

```
  int X;
  cout << "Enter an integer: " << flush;
  cin >> X;

  char ch;

  if (X <= 300)
      if (X < 200)
          if (X <= 100)
            ch = 'A';
          else
            ch = 'B';
      else
          ch = 'C';
  else
      ch = 'D';

      switch(ch)
```

```
{
    case 'D': cout << "Danish" << endl;
      break;
    case 'C': cout << "Canadian" << endl;
      break;
    case 'B': cout << "Belgian" << endl;
      break;
    case 'A': cout << "American" << endl;
      break;
}
```

(d) [8%] Consider the following prototype:

```
void Msg(int I, double D = 78.9, char C = 'F', string S = "Hello");
```

For each function invocation below indicate whether or not it is legal.

```
1.  Msg(11, 78.9, 'F', "Hello");
2.  Msg(11, 6.5, 'H', "Bye");
3.  Msg(6.5, 'H', "Bye");
4.  Msg(11, 'H', "Bye");
5.  Msg(11,6.5,'H');
6.  Msg(11,'H');
7.  Msg(11,6.5);
8.  Msg(11);
```

Question 2. (Coding)

(a) [12%] The code below is a partial declaration for a class of matrices.

```
class Matrix {
  public:
       Matrix (int n);
       void DisplayMatrix(int n);
  private:
       int M[20][20];
       int n;
};
```

The constructor creates a matrix of size n x n. You can assume that the argument corresponding to n will never be greater than 20.

1. Write a definition for the constructor, given that the elements of an n x n matrix A are given by

$$A_ij = \begin{array}{ll} i, & \text{if i = j;} \\ i + j, & \text{otherwise} \end{array}$$

for $i = 0, 1, ..., n-1$ and $j = 0, 1, ..., n-1$.

2. Write a definition for the member function DisplayMatrix(), which outputs the elements of the matrix row by row.

(b) [8%] Design and implement a C++ function `int SumofDivisors(int n)` that returns the sum of the divisors of n. If $n \leq 0$, the function should return 0.

HINT: For $n = 15$, the divisors are 1, 3, 5, 15. The function should return 24.

Question 3. (Arrays) An integer array is defined as
$A[i][0] = i + 1; A[0][j] = j + 1; A[i][j] = A[i][0] + A[0][j], for i, j > 0$
For example, using the above definition the $9x9$ array is shown below:

1	2	3	4	5	6	7	8	9
2	4	5	6	7	8	9	10	11
3	5	6	7	8	9	10	11	12
4	6	7	8	9	10	11	12	13
5	7	8	9	10	11	12	13	14
6	8	9	10	11	12	13	14	15
7	9	10	11	12	13	14	15	16
8	10	11	12	13	14	15	16	17
9	11	12	13	14	15	16	17	18

Write the following declarations and function definitions. Each function below takes two parameters. The first parameter is an array `X` , and the second parameter is an integer `n` , where `n` is the size of the array. That is, the number of rows equals the number of columns `n` .

(a) [4%] Write a function definition `InitArray` which initializes the array `X` according to the definition above.

(b) [8%] Write a function definition `Transpose` which modifies the array `X` in which the rows and columns are respectively the columns and rows of the original array. For example if $n = 3$

and the array `X` is

7	-10	26
3	19	0
18	-6	21

then, after the invocation of `Transpose(X, n)` , the array `X` should be

7	3	18
-10	19	-6
26	0	21

(c) [8%] Write a function `IsAscending` which returns `true` if every row of array `X` is in *increasing* order. For example, every row in the array `A` defined above is in increasing order. Hence, `IsAscending(A,n)` should return true; however, the function `IsAscending` should return false for the array in part (b) above.

Question 4. (Design)

(a) [4%] A truck company that moves parcels across towns wants a design for its boxes. Consider `Box` as a class. Give the attributes and member functions of the class `Box` so that (1) box objects may be constructed; (2) the dimensions of a box can be displayed; (3) the color of a box can be modified; (4) the volume of a box, the surface area of a box, and the sum of its edges can be calculated.

(b) [6%] Give an interface description of the C++ class `Box` .

(c) [10%] Give definitions for all the member functions listed in the `Box` class.

Question 5. (Complete Program)

Write a complete C++ program with headers, preprocessor directives, and in correct syntax to implement the following:

1. Define a class `Distance` for distances; `Distance` has a private part containing two data items, one is `feet` of type `int` , the other is `inches` of type `int` . The public part of the class `Distance` contains a constructor which will be executed automatically whenever an object of type `Distance` is created, and three other member functions. The first function called `get()` is for the user to input, through the input stream `cin` , `feet` and `inches` of an object of class `Distance` . The second member function called `show()` is to output to the screen the distances in `feet` and `inches` . The third member function called `add()` is to add two distances. The first two functions `get()` and `show()` should be defined within the class while the third function `add(Distance, Distance)` should be declared within the class declaration but defined outside.

2. Write the member function `add(Distance u, Distance v)` to add two distances `u` , and `v` . You should add `feet` to `feet` and `inches` to `inches` and convert `inches` to `feet` if the resulting `inches` is greater than or equal to 12.

3. Write function `main()` which creates three `Distance` objects `u` , `v` , and `w` . Then use the member function `get()` to input the data to these three objects. Use the member function `add()` to add `u` and `v` to produce `w` . Use `show()` to output `u` , `v` and the sum `w` in feet and inches.

Appendix B

Solutions

B.1 Exam 1

```
#include <iostream.h>
class Person {
  public:
      char *GetName();
      int VowelsInName();
  private:
      char name[25];
};
// 1.(a)
int Person::VowelsInName()
{
  int v=0;
  for(int i=0;((i<25) && (name[i] != '\0'));i++)
  {
      switch(name[i]) {
          case 'a': case 'e': case 'i': case 'o': case 'u':
            v++;
      }
  }
  return v;
```

```
}
void Question1b()
{
  Person group[20];
  // init group
  // 1.(b)
  int max=0,v=0,mi=0;
  for(int i=0;i<20;i++)
  {
      v=group[i].VowelsInName();
      if(v>max) {
          max=v;
          mi=i;
      }
  }
  cout << group[i].GetName() << endl;
}

// 2.(a)
class Matrix {
  public:
      Matrix (int n);
  private:
      int M[20][20];
};

Matrix::Matrix(int n)
{
  for (int i = 0;i<n;i++)
      for(int j=0;j<n;j++)
      {
          M[i][j]=0;
          if(((i+j)%i)==0)
            M[i][j]++;
          if(((i+j)%j)==0)
            M[i][j]+=2;
      }
}

void Question2bc()
{
  int p=0, m=15, n=7, a=2, b=1;
  while (m || n)
  {
      if (m % a == b)
      {
          p = p + b;
          m -= b;
      }
      if (n % a == b)
      {
          p = p + b;
          n -= b;
      }
      b=a;
      a *= 2;
  }
}

// 3

class Dates {
  public:
      void SetDate(int, int, int);
      void Display();
      void Decrement();
  private:
      int Year;
      int Month;
      int Day;
};

void Dates::SetDate(int d, int m, int y)
{
  Year=y;
  Month=m;
  Day=d;
}
// 3.(b)
void Dates::Display()
{
  char *Days[12]={"Jan","Feb","Mar","Apr","May","Jun",
                         "Jul","Aug","Sep","Oct","Nov","Dec"};
  cout << Day << " " << Days[Month-1] << " " << Year << endl;
}
// 3.(c)
void Dates::Decrement()
{
  int Mdays[]={31,28,31,30,31,30,31,31,30,31,30,31};
  int ly=0;
  if((Year % 4==0) && (Year % 400 !=0))
      ly=1;
  Mdays[1]+=ly;
  if (Day > 1)
      Day--;
  else
  {
      if (Month>1)
        Month--;
      else
      {
          Month=12;
          Year--;
      }
      Day=Mdays[Month-1];
  }
}

void Question4a()
{
  // 4. (a)
  int x=1,y=2;
  int j=x;
  while(j>=y)
  {
      cout << x \times (j + y) << '\n';
      j--;
  }
}

void Question4b()
{
  long nn;
  long nf;
  float s=0;
  long ni=1;
  long ln=10;
  for(i=1;i<=ln;i++)
  {
      nf=1;
      nn=1;
      for(ni=1;ni<=i;ni++)
      {
          nf*=ni;
          nn*=i;
      }
      s+=(float)nf/(float)nn;
  }
}

// 5
class InventoryItem {
  public:
      InventoryItem(int, int, int);
      int GetQty();
      void AddQty(int);
      int SellQty(int);
  private:
      int Qty;
      int Min;
      int Max;
};

InventoryItem::InventoryItem(int q, int min, int max)
{
  Qty=q;
  Min=min;
  Max=max;
}

int InventoryItem::GetQty()
{
  return Qty;
}

void InventoryItem::AddQty(int a)
{
  Qty += a;
}
```

```
int InventoryItem::SellQty(int s)
{
  if (s <= Qty)
  {
      Qty-=s;
      if (Qty < Min)
      cout << "Quantity below minimum, order " << (Max - Qty) << endl;
      return s;
  }
  return 0;
}
```

B.2 Exam 2

```
#include <iostream.h>
#include <math.h>

void f(int &x, int y) {
  int t = x;
  x = y;
  y = t;
  cout << "f(int, int) says x is " << x << endl;
  cout << "f(int, int) says y is " << y << endl;
  return;
}

void f(int x, int y, const int z) {
  if (x > y) f(y, x-z);
  else f(x,y-z);
  return;
}

void Question1a() {
  int i = 5;
  int j = 16;
  int k = 6;
  cout << " f(int, int) is called with values " << i
      << " and " << j << endl;
  f(i, j);
  cout << " f(int, int, const int) is called with values "
      << k << ", " << j << ", and " << i << endl;
  f(k, j, i);
}

void Question1b() {
  int number = 161;
  int rem = 0;
  int index = 0;
  cout << "The input number is " << number << endl;
  if (number % 2 == 1) {
      rem = 1;
      number = number - 1;
  }
  while (number % 2 == 0) {
      number = number / 2;
      index = index + 1;
  }
  cout << "index is " << index << ", multiplier is " << number
      << ", adder is " << rem << endl;
}

float Question2() {
 float n=1.0,s=0.0,p=1.0;
 while (n<1000.0)
 {
     s=s+p/n;
     p=-p;
     n+=1.0;
     if (fmod(n,100)==0)
     cout << "n = " << n << " s = " << s << endl;
 }
 return s;
}

int Outlier(double list[10])
{
  float diff,x,avg=0.0;
  int i;
  for(i=0;i<10;i++)
      avg=avg+list[i];
  avg/=10.0;
  for(i=0;i<10;i++)
  {
       x=list[i];
       diff=x-avg;
       if (diff < 0)
            diff=-diff;
       if(diff>avg)
            return 1;
  }
  return 0;
}

void Question3()
{
  double list[10]={1.0,2.0,3.0,4.0,5.0,1.5,2.5,3.5,4.5,15.5};
  cout << "Outlier = " << Outlier(list) << endl;;
}

void perfect(int p)
{
  int sum=0,n=1;
  while (n <= p/2)
  {
       if ((p % n) == 0)
            sum += n;
       n++;
  }
  if (sum == p)
       cout << "Input number is perfect" << endl;
  else
       cout << "Input number is not perfect" << endl;
}
```

```
void Question4()
{
  perfect(28);
  perfect(6);
  perfect(25);
}

// 5

class Point
{
  public:
      Point(double xi, double yi);
      void Rotate(double angle);
      double Dist(Point p1, Point p2);
      double x;
      double y;
};

Point::Point(double xi, double yi)
{
  x=xi;
  y=yi;
}

void Point::Rotate(double angle)
{
  double xp,yp;
  xp=x*cos(angle)+y*sin(angle);
  yp=-x*sin(angle)+y*cos(angle);
  x=xp;
  y=yp;
}
```

```
double Dist(Point p1, Point p2)
{
  return (sqrt(pow(p1.x-p2.x,2)+pow(p1.y-p2.y,2)));
}

void Question5()
{
  Point p1(3.0,4.0), p2(2.0,5.0);
  p1.Rotate(1);
  p2.Rotate(2);
  cout << "Distance = " << Dist(p1,p2) << endl;
}
```

B.3 Exam 3

```
#include <iostream.h>

void Question1a()
{
  int m = 35, n = 14;
  while (m != n)
  {
      if (m > n)
          m = m - n;
      else
      n = n - m;
  }
}

// 1 (a)

int i;
void f();
void main(){
cout << i << endl;
i = 5;
f();
}
void f() {
cout << i << endl;
char i;
i = 'a';
cout << i << endl;
cout << ::i << endl;
::i = 3;
{ cout << i << endl;
  int i = 9;
  cout << ::i << endl;
  cout << i << endl;
}
cout << i << endl;
cout << ::i << endl;
cout << i << endl;
::i = 90;
cout << i << endl;
cout << ::i << endl;
}

// 1 (c)

void main()
{
  int X;
  cout << "Enter an integer: " << flush;
  cin >> X;

  char ch;

  if (X <= 300)
      if (X < 200)
          if (X <= 100)
            ch = 'A';
          else
            ch = 'B';
      else
          ch = 'C';
  else
      ch = 'D';

      switch(ch)
      {
          case 'D': cout << "Danish" << endl;
            break;
          case 'C': cout << "Canadian" << endl;
            break;
          case 'B': cout << "Belgian" << endl;
            break;
          case 'A': cout << "American" << endl;
            break;
      }
}

// 1 (d)

void Msg(int I, double D = 78.9, char C = 'F', string S = "Hello");
void main()
{
  Msg(11, 78.9, 'F', "Hello");
  Msg(11, 6.5, 'H', "Bye");
  Msg(6.5, 'H', "Bye");
  Msg(11, 'H', "Bye");
  Msg(11,6.5,'H');
  Msg(11,'H');
  Msg(11,6.5);
  Msg(11);
}
// Spring 98 2.(a)
#include <iostream.h>

class Matrix {
  public:
      Matrix (int n);
      void DisplayMatrix(int n);
  private:
      int M[20][20];
      int n;
};

Matrix::Matrix(int n)
{
  for (int i = 0;i<n;i++)
      for(int j=0;j<n;j++)
      {
          if(i==j)
            M[i][j]=i;
          else
            M[i][j]=i+j;
      }
}

void Matrix::DisplayMatrix(int n)
{
  for (int i = 0;i<n;i++)
  {
      for(int j=0;j<n;j++)
      {
          cout << M[i][j] << flush;
      }
      cout << endl;
  }
}

// 2 (b)

int SumofDivisors(int n)
{
  int s=0;
  for(int i = 1;i==n;i++)
  {
        if((n%i)==0)
            s += i;
  }
}

// 3 (a)

void InitArray(int **X, int n)
{
  int i,j;
  for(i=0;i<n;i++)
  {
      X[i][0]=i+1;
      X[0][i]=i+1;
  }
  for(i=1;i<n;i++)
      for(j=1;j<n;j++)
          X[i][j]=X[i][0]+X[0][j];
}

// 3 (b)

void Transpose(int **X, int n)
{
  int i,j,t;
  for(i=0;i<(n-1);i++)
      for(j=(i+1);j<n;j++)
      {
          t=X[i][j];
          X[i][j]=X[j][i];
          X[j][i]=t;
      }
}
```

```
// 3 (c)

int IsAscending(int **A, int n)
{
  int i,j;
  for(i=0;i<n;i++)
      for(j=0;j<(n-1);j++)
          if(A[i][j]<=A[i][j+1])
            return 0;
  return 1;
}

// 4

class Box {
public:
  Box(int h,int l,int w,int c);
  void Display(void);
  void SetColor(int c);
  void BoxStats(void);
private:
  int h;
  int l;
  int w;
  int c;
};

Box::Box(int ih, int il, int iw, int ic)
{
  h=ih;
  l=il;
  w=iw;
  c=ic;
```

```
}

void Box::Display(void)
{
  cout << "Height = " << h << endl;
  cout << "Length = " << l << endl;
  cout << "Width = " << w << endl;
}

void Box::SetColor(int ic)
{
  c=ic;
}

void Box::BoxStats(void)
{
  cout << "Volume = " << h*l*w << endl;
  cout << "Surface area = " << 2*(h*l+h*w+l*w) << endl;
  cout << "Sum of edges = " << 4*(l+h+w) << endl;
}

// 5

class Distance {
  public:
      Distance(int f=0, int i=0);
      void get(void){cin >> feet >> inches;};
      void show(void){cout << feet << " feet " << inches << " inches" << endl;};
      Distance add(Distance u, Distance v);
      int GetFeet(void) {return feet;};
      int GetInches(void) {return inches;};
  private:
      int feet;
      int inches;
```

```
};

Distance Distance::add(Distance u, Distance v)
{
  int tf, ti;
  tf = u.GetFeet() + v.GetFeet();
  ti = u.GetInches() + v.GetInches();
  if (ti >= 12)
  {
      tf++;
      ti-=12;
  }
  return Distance(tf,ti);
}
```

Appendix C

Test Scores

Exam 1, Section 1

STUD	MARK	Q1AB	Q2A	Q2BC	Q3ABC	Q4A	Q4B	Q5AB
101	65.00	13.00	4	9	12.00	8	8	11.00
103	53.00	11.00	6	3	8.00	5	7	13.00
106	87.00	19.00	7	12	16.00	9	10	14.00
107	81.00	17.00	5	9	14.00	9	10	17.00
109	56.00	12.00	4	1	6.00	8	7	18.00
110	82.00	19.00	5	6	19.00	10	10	13.00
111	53.00	10.00	5	8	7.00	6	8	9.00
112	68.00	15.00	4	4	15.00	8	9	13.00
113	74.00	19.00	6	1	10.00	10	10	18.00
114	52.00	6.00	6	2	9.00	10	5	14.00
115	50.00	12.00	4	2	3.00	8	9	12.00
117	82.00	18.00	4	8	18.00	9	10	15.00
118	85.00	16.00	6	6	20.00	8	10	19.00
119	69.00	15.00	4	3	18.00	8	6	15.00
121	59.00	14.00	6	0	11.00	10	8	10.00
122	87.00	18.00	6	9	19.00	10	10	15.00
124	76.00	14.00	4	8	15.00	10	9	16.00
125	91.00	20.00	7	12	18.00	8	10	16.00
126	79.00	18.00	4	12	16.00	10	7	12.00
128	70.00	13.00	5	8	15.00	10	8	11.00
130	64.00	16.00	4	3	17.00	8	10	6.00

131	78.00	19.00	4	3	19.00	8	10	15.00
132	92.00	19.00	8	12	19.00	10	9	15.00
133	99.00	20.00	7	12	20.00	10	10	20.00
136	61.00	17.00	3	1	11.00	10	6	13.00

Exam 1, Section 2

STUD	MARK	Q1AB	Q2A	Q2BC	Q3ABC	Q4A	Q4B	Q5AB
201	87.00	18.00	8	7	18.00	10	9	17.00
202	80.00	16.00	8	5	17.00	10	10	14.00
203	69.00	13.00	6	4	15.00	8	9	14.00
204	59.00	14.00	2	4	11.00	8	7	13.00
205	67.00	15.00	1	4	11.00	9	10	17.00
206	51.00	14.00	6	3	6.00	3	7	12.00
208	75.00	19.00	5	6	14.00	10	7	14.00
209	80.00	20.00	8	8	17.00	9	8	10.00
210	63.00	8.00	7	4	15.00	8	10	11.00
211	51.00	17.00	2	3	4.00	8	7	10.00
212	68.00	12.00	4	9	14.00	8	10	11.00
213	59.00	10.00	5	3	14.00	8	7	12.00
215	58.00	16.00	4	0	17.00	2	7	12.00
216	56.00	16.00	5	6	2.00	9	7	11.00
218	61.00	19.00	7	6	11.00	8	9	1.00
220	63.00	18.00	6	3	9.00	9	8	10.00
223	61.00	18.00	4	3	11.00	10	3	12.00
224	54.00	16.00	4	4	5.00	10	6	9.00
225	69.00	13.00	5	9	17.00	8	6	11.00
227	92.00	20.00	7	12	22.00	9	10	12.00
228	83.00	20.00	7	11	10.00	9	8	18.00
229	82.00	15.00	8	12	14.00	10	9	14.00

Exam 2, Section 1

STUD	MARK	Q1A	Q1B	Q2	Q3	Q4	Q5ABC
301	81.00	10	10	5	20	20	16.00
302	81.00	10	10	6	20	17	18.00
303	83.00	10	10	12	14	20	17.00
304	65.00	5	4	8	15	20	13.00
305	81.00	10	10	4	19	20	18.00
306	70.00	10	10	0	18	20	12.00
307	80.00	10	10	5	18	20	17.00
308	82.00	10	10	6	18	20	18.00
309	61.00	10	4	0	14	17	16.00
311	83.00	10	10	5	18	20	20.00
312	69.00	5	10	10	18	20	6.00
314	78.00	10	10	8	14	20	16.00
315	94.00	7	10	18	20	20	19.00
316	78.00	10	4	10	16	20	18.00
317	98.00	10	10	20	20	20	18.00
318	93.00	10	10	20	20	20	13.00
319	97.00	10	10	20	19	20	18.00
320	57.00	7	10	8	11	7	14.00
321	88.00	10	4	20	18	20	16.00
322	100.00	10	10	20	20	20	20.00
323	98.00	10	10	20	20	20	18.00
324	72.00	5	4	15	14	20	14.00
325	96.00	10	10	20	18	20	18.00
326	92.00	10	10	12	20	20	20.00
327	64.00	4	5	5	16	20	14.00
328	84.00	10	10	8	20	20	16.00
329	82.00	10	10	5	19	20	18.00
330	65.00	5	10	0	14	20	16.00
331	66.00	10	10	0	14	20	12.00
332	67.00	8	3	10	16	20	10.00

Exam 2, Section 2

STUD	MARK	Q1A	Q1B	Q2	Q3	Q4	Q5ABC
401	82.00	10	10	10	19	19	14.00
402	60.00	1	10	8	6	20	15.00
403	90.00	10	10	12	20	20	18.00
404	98.00	10	10	20	19	20	19.00
406	85.00	10	10	6	20	20	19.00
407	84.00	10	10	6	20	20	18.00
408	90.00	10	10	12	20	20	18.00
409	92.00	10	10	12	20	20	20.00
410	72.00	5	10	6	20	20	11.00
412	78.00	10	10	0	20	20	18.00
413	100.00	10	10	20	20	20	20.00
414	61.00	10	10	0	14	19	8.00
415	85.00	10	10	5	20	20	20.00
416	85.00	10	10	10	20	20	15.00
417	72.00	8	10	8	20	8	18.00
419	79.00	10	10	12	15	20	12.00
420	69.00	10	10	0	16	17	16.00
421	88.00	10	10	8	20	20	20.00
422	79.00	10	10	4	18	20	17.00
423	80.00	10	4	6	20	20	20.00
424	88.00	10	10	16	18	18	16.00
425	90.00	10	10	10	20	20	20.00
426	69.00	10	10	0	15	20	14.00
427	88.00	10	10	12	19	19	18.00
428	80.00	10	10	0	20	20	20.00
429	92.00	10	10	12	20	20	20.00
430	50.00	5	2	0	14	18	11.00
431	85.00	10	10	8	20	19	18.00
432	69.00	3	10	6	18	20	12.00
433	70.00	7	10	0	20	18	15.00
434	82.00	10	10	5	19	18	20.00
435	67.00	10	7	8	14	12	16.00

436	58.00	10	10	8	0	20	10.00
437	86.00	10	10	10	18	20	18.00
438	85.00	10	10	6	20	20	19.00

Exam 3, Section 1

STUD	MARK	Q1A	Q1B	Q1C	Q1D	Q2A	Q2B	Q3A	Q3B	Q3C	Q4	Q5
502	86.00	1.0	3.50	3.0	7.0	11.0	8.0	3.0	5.5	4.0	20.0	20.0
503	89.50	.0	2.50	4.0	8.0	12.0	8.0	4.0	8.0	7.0	18.0	18.0
504	93.00	.0	4.00	4.0	8.0	11.0	8.0	4.0	8.0	8.0	18.0	20.0
505	89.00	.0	3.50	4.0	8.0	12.0	8.0	4.0	5.0	7.5	19.0	18.0
506	99.00	4.0	4.00	4.0	7.0	12.0	8.0	4.0	8.0	8.0	20.0	20.0
507	98.50	4.0	3.50	4.0	7.0	12.0	8.0	4.0	8.0	8.0	20.0	20.0
508	94.00	4.0	3.00	3.0	8.0	12.0	8.0	4.0	8.0	8.0	18.0	18.0
510	95.00	4.0	3.50	4.0	7.0	12.0	8.0	3.5	8.0	7.0	18.0	20.0
511	94.00	4.0	3.00	4.0	6.0	12.0	8.0	4.0	8.0	8.0	19.0	18.0
512	92.00	4.0	4.00	4.0	7.0	12.0	8.0	3.0	4.0	7.0	19.0	20.0
513	94.50	4.0	4.00	4.0	7.0	12.0	8.0	3.5	6.0	7.0	19.0	20.0
515	92.00	4.0	4.00	4.0	7.0	12.0	8.0	3.0	6.0	7.0	20.0	17.0
516	94.00	4.0	4.00	4.0	6.0	12.0	8.0	4.0	7.0	8.0	19.0	18.0
517	66.00	1.5	3.50	4.0	7.0	7.0	.0	2.0	5.0	4.0	18.0	14.0
518	95.00	4.0	3.50	4.0	7.0	12.0	8.0	3.5	7.0	7.0	19.0	20.0
519	94.50	4.0	3.50	4.0	8.0	12.0	8.0	4.0	8.0	5.0	18.0	20.0
520	85.50	4.0	2.50	4.0	7.0	12.0	8.0	3.0	3.0	7.0	18.0	17.0
521	98.00	4.0	4.00	4.0	7.0	11.0	8.0	4.0	8.0	8.0	20.0	20.0
522	93.50	4.0	3.50	4.0	7.0	12.0	8.0	4.0	8.0	5.0	20.0	18.0
523	97.00	4.0	4.00	4.0	6.0	12.0	7.0	4.0	8.0	8.0	20.0	20.0
524	87.00	4.0	3.50	3.0	7.0	12.0	8.0	4.0	7.0	5.5	17.0	16.0
525	88.50	4.0	3.00	4.0	7.0	12.0	8.0	3.5	6.0	7.0	20.0	14.0
526	92.00	4.0	3.50	4.0	7.0	11.0	8.0	3.5	6.0	7.0	18.0	20.0
527	69.00	4.0	4.00	4.0	2.0	12.0	.0	2.0	5.0	5.0	18.0	13.0
528	67.00	4.0	3.00	4.0	4.0	11.0	2.0	4.0	5.0	4.0	18.0	8.0
529	95.00	4.0	4.00	4.0	7.0	12.0	8.0	3.0	6.0	7.0	20.0	20.0
530	55.00	4.0	.00	3.0	7.0	7.0	2.0	2.0	5.0	3.0	13.0	9.0
531	57.50	4.0	3.50	4.0	7.0	6.0	2.0	3.0	3.0	4.0	10.0	11.0
532	92.00	4.0	4.00	4.0	7.0	11.0	8.0	4.0	7.0	8.0	18.0	17.0
533	70.50	4.0	3.50	4.0	7.0	12.0	5.0	1.0	1.0	3.0	18.0	12.0
534	76.00	4.0	.00	4.0	6.0	11.0	8.0	1.0	6.0	7.0	14.0	15.0
535	95.00	4.0	3.00	4.0	7.0	11.0	8.0	4.0	8.0	8.0	19.0	19.0

536	91.00	4.0	4.00	4.0	7.0	12.0	.0	4.0	8.0	8.0	20.0	20.0
537	71.00	.0	2.00	4.0	7.0	12.0	.0	3.0	5.0	5.0	18.0	15.0
538	60.50	.0	2.50	4.0	7.0	6.0	8.0	1.0	3.0	4.0	12.0	13.0

Exam 3, Section 2

STUD	MARK	Q1A	Q1B	Q1C	Q1D	Q2A	Q2B	Q3A	Q3B	Q3C	Q4	Q5
602	92.00	4.0	4.0	4.0	8.0	12.0	8.0	4.0	8.0	8.0	20.0	12.0
603	51.50	4.0	3.5	4.0	7.0	7.0	.0	2.0	.0	.0	18.0	6.0
604	62.00	4.0	3.0	4.0	7.0	8.0	5.0	3.0	5.0	2.0	15.0	6.0
605	55.00	4.0	2.0	4.0	7.0	6.0	5.0	1.0	1.0	2.0	15.0	8.0
606	50.50	4.0	3.0	4.0	7.0	9.0	8.0	2.5	5.0	.0	8.0	.0
608	89.00	1.0	4.0	4.0	7.0	12.0	8.0	4.0	6.0	7.0	20.0	16.0
609	72.50	4.0	3.0	.0	7.0	11.5	8.0	2.0	5.5	3.5	14.0	14.0
610	96.00	4.0	4.0	4.0	7.0	12.0	8.0	4.0	7.0	8.0	20.0	18.0
612	50.00	4.0	2.5	4.0	4.0	10.0	8.0	3.0	5.5	2.0	5.0	2.0
613	87.00	1.0	3.0	4.0	7.0	10.0	8.0	4.0	7.0	6.0	19.0	18.0
614	91.00	4.0	4.0	4.0	7.0	12.0	7.0	4.0	4.0	7.0	20.0	18.0
615	64.50	4.0	2.0	1.5	4.0	11.0	.0	3.0	4.0	2.0	20.0	13.0
619	99.00	4.0	4.0	4.0	7.0	12.0	8.0	4.0	8.0	8.0	20.0	20.0
620	91.00	.0	4.0	4.0	7.0	12.0	8.0	4.0	8.0	8.0	20.0	16.0
621	60.50	4.0	4.0	4.0	7.0	.0	6.0	2.5	3.0	4.0	16.0	10.0
623	82.50	4.0	4.0	4.0	7.0	11.0	.0	3.5	4.0	7.0	20.0	18.0
625	51.00	1.0	3.5	1.5	7.0	.0	8.0	2.0	1.0	2.0	18.0	7.0
626	93.00	4.0	4.0	4.0	7.0	12.0	8.0	4.0	4.0	8.0	20.0	18.0
627	92.00	4.0	4.0	4.0	7.0	12.0	8.0	4.0	8.0	8.0	17.0	16.0
628	66.50	1.0	2.5	4.0	7.0	10.0	8.0	1.0	4.0	5.0	14.0	10.0
629	94.00	4.0	4.0	4.0	8.0	12.0	8.0	4.0	4.0	8.0	20.0	18.0
631	85.50	1.0	4.0	4.0	7.0	10.0	8.0	3.5	7.0	5.0	20.0	16.0
632	71.50	4.0	.0	4.0	7.0	12.0	8.0	3.0	5.5	7.0	9.0	12.0
633	96.50	4.0	3.5	4.0	7.0	11.0	8.0	4.0	8.0	7.0	20.0	20.0
634	70.00	4.0	3.5	4.0	7.0	11.0	2.0	2.5	5.0	4.0	15.0	12.0
635	86.00	4.0	3.5	4.0	7.0	11.0	8.0	3.5	5.0	8.0	19.0	13.0
636	86.50	1.0	4.0	4.0	7.0	12.0	8.0	3.5	7.0	7.0	20.0	13.0
637	77.00	.0	3.5	4.0	6.0	11.0	8.0	4.0	4.0	3.0	18.5	15.0
638	66.00	1.0	3.0	4.0	8.0	3.0	3.0	4.0	5.0	5.0	20.0	10.0
639	73.50	4.0	3.5	4.0	2.0	11.0	8.0	4.0	4.0	5.0	18.0	10.0
640	85.00	1.0	3.0	.0	7.0	12.0	7.0	4.0	8.0	8.0	19.0	16.0
641	91.50	4.0	3.5	4.0	8.0	12.0	8.0	4.0	4.0	8.0	18.0	18.0

Appendix D

Halstead Metrics and LOC

UOPER = Unique operators

UOPRA = Unique operands

TOPER = Total operators

TOPRA = Total operands

HVOCAB = UOPER + UOPRA

HSIZE = TOPER + TOPRA

EXAM: 1

QUESTION	HDIFF	HEFFORT	HSIZE	LOC	TOPER	TOPRA	UOPER	UOPRA	HVOCAB
1ab	.038	30980.74	150	36	97	53	29	21	50
2a	.067	21027.30	98	18	60	38	19	8	27
2bc	.114	7394.63	72	17	40	32	14	10	24
3abc	.062	31606.14	182	81	99	83	27	39	66
4a	.095	2034.05	34	10	21	13	13	6	19
4b	.121	5859.35	69	17	40	29	12	9	21
5ab	.031	16458.83	113	36	85	28	21	10	31

EXAM: 2

QUESTION	HDIFF	HEFFORT	HSIZE	LOC	TOPER	TOPRA	UOPER	UOPRA	HVOCAB
1a	.058	13534.86	133	27	89	44	17	19	36
1b	.096	7661.01	77	17	46	31	14	10	24
2	.065	7672.54	63	12	39	24	19	9	28
3	.078	14332.14	89	23	51	38	19	11	30
4	.060	4531.17	57	20	37	20	18	11	29
5abc	.042	27792.52	125	26	88	37	20	8	28

EXAM: 3

QUESTION	HDIFF	HEFFORT	HSIZE	LOC	TOPER	TOPRA	UOPER	UOPRA	HVOCAB
1a	.130	2688.00	32	8	18	14	12	4	16
1b	.042	9257.57	119	29	95	24	12	8	20
1c	.040	9057.21	98	26	71	27	19	14	33
1d	.135	3879.71	72	9	43	29	10	12	22
2a	.032	29700.00	120	30	87	33	24	8	32
2b	.103	2059.20	36	9	23	13	11	5	16
3a	.117	15388.83	88	12	50	38	13	6	19
3b	.111	9206.32	74	11	43	31	13	7	20
3c	.079	8847.51	62	9	38	24	16	6	22
4	.044	23471.57	165	35	118	47	18	15	33
5	.025	26021.57	127	30	94	33	28	12	40

Appendix E

Correlations

```
Here are the bivariate Pearson correlations.
EXAM:      1, Section 1
                        - -  Correlation Coefficients  - -
              MEAN       STDDEV     LOC        HEFFORT    ID
MEAN           1.0000     -.7859     -.1107     -.1551     -.6091
             (     7)   (     7)   (     7)   (     7)   (     7)
             P= .       P= .036    P= .813    P= .740    P= .147
STDDEV        -.7859     1.0000      .1208      .1375      .3255
             (     7)   (     7)   (     7)   (     7)   (     7)
             P= .036    P= .       P= .796    P= .769    P= .476
LOC           -.1107      .1208     1.0000      .8746     -.5462
             (     7)   (     7)   (     7)   (     7)   (     7)
             P= .813    P= .796    P= .       P= .010    P= .205
HEFFORT       -.1551      .1375      .8746     1.0000     -.3679
             (     7)   (     7)   (     7)   (     7)   (     7)
             P= .740    P= .769    P= .010    P= .       P= .417
ID            -.6091      .3255     -.5462     -.3679     1.0000
             (     7)   (     7)   (     7)   (     7)   (     7)
             P= .147    P= .476    P= .205    P= .417    P= .
```

```
EXAM:     1, Section 2
                    - -  Correlation Coefficients  - -
              MEAN        STDDEV      LOC         HEFFORT     ID
MEAN          1.0000      -.6216      -.2411      -.0994      -.4146
             (     7)    (     7)    (     7)    (     7)    (     7)
             P= .        P= .136     P= .603     P= .832     P= .355
STDDEV        -.6216      1.0000      -.2108      -.0219       .5837
             (     7)    (     7)    (     7)    (     7)    (     7)
             P= .136     P= .        P= .650     P= .963     P= .169
LOC           -.2411      -.2108      1.0000       .8479      -.5462
             (     7)    (     7)    (     7)    (     7)    (     7)
             P= .603     P= .650     P= .        P= .016     P= .205
HEFFORT       -.0994      -.0219       .8479      1.0000      -.2993
             (     7)    (     7)    (     7)    (     7)    (     7)
             P= .832     P= .963     P= .016     P= .        P= .514
ID            -.4146       .5837      -.5462      -.2993      1.0000
             (     7)    (     7)    (     7)    (     7)    (     7)
             P= .355     P= .169     P= .205     P= .514     P= .
```

```
EXAM:     2, Section 1
                    - -  Correlation Coefficients  - -

             MEAN        STDDEV      LOC         HEFFORT     ID

MEAN         1.0000      -.3061      -.3622      -.8452      -.7805
            (     5)    (     5)    (     5)    (     5)    (     5)
            P= .        P= .616     P= .549     P= .071     P= .119

STDDEV       -.3061      1.0000      -.3013      -.1427      -.1015
            (     5)    (     5)    (     5)    (     5)    (     5)
            P= .616     P= .        P= .622     P= .819     P= .871

LOC          -.3622      -.3013      1.0000       .7068       .8604
            (     5)    (     5)    (     5)    (     5)    (     5)
            P= .549     P= .622     P= .        P= .182     P= .061

HEFFORT      -.8452      -.1427       .7068      1.0000       .9503
            (     5)    (     5)    (     5)    (     5)    (     5)
            P= .071     P= .819     P= .182     P= .        P= .013

ID          -.7805      -.1015       .8604       .9503      1.0000
            (     5)    (     5)    (     5)    (     5)    (     5)
            P= .119     P= .871     P= .061     P= .013     P= .
```

```
EXAM:     2, Section 2
                    - -  Correlation Coefficients  - -

              MEAN        STDDEV      LOC         HEFFORT     ID

MEAN          1.0000      -.3031      -.7349      -.9785      -.9358
             (     5)    (     5)    (     5)    (     5)    (     5)
             P= .        P= .620     P= .157     P= .004     P= .019

STDDEV        -.3031      1.0000       .5230       .2931       .5284
             (     5)    (     5)    (     5)    (     5)    (     5)
             P= .620     P= .        P= .366     P= .632     P= .360

LOC           -.7349       .5230      1.0000       .7068       .8604
             (     5)    (     5)    (     5)    (     5)    (     5)
             P= .157     P= .366     P= .        P= .182     P= .061

HEFFORT       -.9785       .2931       .7068      1.0000       .9503
             (     5)    (     5)    (     5)    (     5)    (     5)
             P= .004     P= .632     P= .182     P= .        P= .013

ID            -.9358       .5284       .8604       .9503      1.0000
             (     5)    (     5)    (     5)    (     5)    (     5)
             P= .019     P= .360     P= .061     P= .013     P= .
```

```
EXAM:     3, Section 1
                      - -  Correlation Coefficients  - -

               MEAN        STDDEV      LOC         HEFFORT     ID

MEAN           1.0000      -.7202       .6636       .4799      -.6279
              (    11)    (    11)    (    11)    (    11)    (    11)
              P= .        P= .012     P= .026     P= .135     P= .039

STDDEV         -.7202      1.0000      -.5993      -.5367       .2347
              (    11)    (    11)    (    11)    (    11)    (    11)
              P= .012     P= .        P= .051     P= .089     P= .487

LOC             .6636      -.5993      1.0000       .7605      -.5238
              (    11)    (    11)    (    11)    (    11)    (    11)
              P= .026     P= .051     P= .        P= .007     P= .098

HEFFORT         .4799      -.5367       .7605      1.0000       .0291
              (    11)    (    11)    (    11)    (    11)    (    11)
              P= .135     P= .089     P= .007     P= .        P= .932

ID             -.6279       .2347      -.5238       .0291      1.0000
              (    11)    (    11)    (    11)    (    11)    (    11)
              P= .039     P= .487     P= .098     P= .932     P= .
```

```
EXAM:     3, Section 2
                        - -  Correlation Coefficients  - -
              MEAN          STDDEV      LOC          HEFFORT     ID
MEAN           1.0000        -.3691       .3084       -.0303      -.6233
              (    11)      (    11)    (    11)     (    11)    (    11)
              P= .          P= .264     P= .356      P= .929     P= .040
STDDEV         -.3691        1.0000      -.3758       -.2811       .0502
              (    11)      (    11)    (    11)     (    11)    (    11)
              P= .264       P= .        P= .255      P= .402     P= .883
LOC             .3084        -.3758      1.0000        .7605      -.5238
              (    11)      (    11)    (    11)     (    11)    (    11)
              P= .356       P= .255     P= .         P= .007     P= .098
HEFFORT        -.0303        -.2811       .7605       1.0000       .0291
              (    11)      (    11)    (    11)     (    11)    (    11)
              P= .929       P= .402     P= .007      P= .        P= .932
ID             -.6233         .0502      -.5238        .0291      1.0000
              (    11)      (    11)    (    11)     (    11)    (    11)
              P= .040       P= .883     P= .098      P= .932     P= .
```