Properties of "Good" Java Examples

Nadeem Abbas

January 5, 2010
Master's Thesis in Computing Science, 30 ECTS credits
Supervisor at CS-UmU: Jürgen Börstler
Examiner: Per Lindström

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

Example programs are well known as an important tool to learn computer programming. Realizing the significance of example programs, this study has been conducted with a goal to measure and evaluate the quality of examples used in academia. We make a distinction between "good" and "bad" examples, as badly designed examples may prove harmful for novice learners. In general, students differ from expert programmers in their approach to read and comprehend a program. How do students understand example programs is explored in the light of classical theories and models of program comprehension. Key factors that impact program quality and comprehension are identified. To evaluate as well as improve the quality of examples, a set of quality attributes is proposed. Relationship between program complexity and quality is examined. We rate readability as a prime quality attribute and hypothesize that example programs with low readability are difficult to understand. *Software Reading Ease Score (SRES)*, a program readability metric proposed by Börstler et al. [5], is implemented to provide a readability measurement tool. *SRES* is based on lexical tokens and is easy to compute using static code analysis techniques. To validate SRES metric, results are statistically analyzed in correlation to earlier existing well acknowledged software metrics.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Learning something new is a human instinct and great fun, but it often involves complications and challenges. Being students of computer science we learn lot of new concepts, tools and technologies, and often face problems as well. Whenever there is a problem and a will to resolve it, there is a way, i.e., problem solving. Programming, in fact, is nothing but a problem-solving approach. However learning how to program in itself is a big problem confronted by many.

Programming is a foundational skill for all computing disciplines including computer science (CS), software engineering (SE), and information technology (IT). According to the Computer Science Curriculum 2008 [66], jointly composed by IEEE Computer Society and ACM, good programming knowledge is a prerequisite to the study of most of computer science. Computer science students, at undergraduate level, are supposed to gain competency in at least one programming language. *Programming Fundamentals (PF)* and *Programming Languages (PL)* are the main subjects to impart programming knowledge. There are many programming languages and paradigms available for educators to choose for their students. Among these java and object oriented programming have become most popular choices during the last few years. This study is not concerned with pros and cons of object orientation or java as a first programming language. We mainly study quality of the example programs, written in java, in terms of desired quality attributes and their impact on learning programming fundamentals.

The term's *example* and *example program* are interchangeable and object-oriented is the intended programming paradigm. Example program refers only to the source code excluding the supporting textual description, lecture notes or any visual aids, such as UML diagrams.

1.1 Problem Description

Learning by examples is a well established pedagogy [15]. While learning computer programming, example programs play a vital role and can be used as an effective tool to explain complex concepts which otherwise are difficult to comprehend. Example programs act as templates, guidelines and inspirations for learners [7]. Examples are generally believed to make learning easier by reinforcing fundamental concepts and eliminating confusion or misconception. However, examples are not always equally good for learning [48]. Badly designed examples mislead students to build myths and misconception. Use of good programming examples is extremely important as the hypothesis is supported by [7, 48, 5].

1.1.1 Problem Statement

What are the quality attributes for example programs? and how can we measure the quality of an example?

1.1.2 Goals

1. Review the literature to understand the relationship between example quality, comprehension and complexity; principles and guidelines for object-oriented pedagogy and software development in general.

2. Probe on the characteristics of example programs in an educational context, and propose a set of desired quality attributes for example programs that can be used to distinguish between "good" and "bad" examples.

3. Develop a program quality measurement tool based on Software Reading Ease Score [5] and Halstead's metrics of software science [29].

4. Evaluate the usefulness of SRES measures as compare to the existing software complexity measures.

1.2 Purposes

Overall purpose of this study is to facilitate educators, academics and students in determining the quality of example programs used in java programming education. It provides them with a light weight, specifically designed software tool that they can use to evaluate readability and quality of java example programs. Though there are already a number of open source code analyzers in java available, see `http://java-source.net/open-source/code-analyzers`. However, none of these provides measures for program readability. The only program readability metric found is the one defined by Buse and Weimer [11], but it differs from the SRES metric in its approach.

1.3 Related Work

Researchers in academia and industry have done plenty of work regarding software quality. A large number of software metrics and models [53, 13, 41, 11, 36, 32, 29, 50] have been proposed, which certainly helps to improve software quality and maintenance efforts. However, there is relatively little work done in the academic context, to determine desirable characteristics of example programs used as learning beacons in an academic world.

Kolling [39] urges educators and researchers to carefully design examples programs and be selective with their choice of examples. *Code Reading* is among the eight guidelines, discussed by Kolling, for teaching object oriented programming with java. Malan and Halland [48] claim to treat examples as products selling concepts to the novice learners. The authors identified four typical problems with example programs, particularly with respect to object oriented paradigm. The first issue pointed is that of too simple and abstract examples, which follow a non-realistic general approach, with no particular objective. The second problem identified is of example programs which are more realistic and too complex to understand. Such examples have unnecessarily complex constructs that create troubles for novice learners. The third issue is that of inconsistency, where example programs overthrow earlier taught concepts, instead of reinforcing them. The last issue concerns those

examples that are so badly constructed that they even undermine the concepts they are trying to teach. However Malan and Halland's work lacks empirical evaluation and does not provide any explicit solution or measures to improve the quality of example programs.

Program readability is a critical factor in program maintenance. Buse and Weimer [11] proposed a metric for program readability. The authors study readability with respect to program quality and its role in maintenance. According to the authors, readability is different from complexity, and it depends mainly on local, line-by-line elements of a program. Their findings are based on 12,000 human judgments regarding code readability, and report some interesting facts. To get readability judgment by human annotators, they use small java code snippets (with average code length of 7.7 lines), and ask the annotators to rate the code on an ordinal scale. The authors intentionally use small code snippets in order to distinguish which code elements are most predictive of readability. They have defined a snippet selection policy, however the code snippets fail to provide contextual information, or background knowledge that is very important according to models of program comprehensions, see Section 2.2. Their results show that code features like 'average line length' and 'average number of identifiers per line' are very important to predict readability. An interesting result reported is that the 'average identifier length' has almost no affect on readability measurement. This is contrary to the notion of "self describing code" that advocates use of long descriptive identifiers in place of short single character or abbreviation style identifiers.

Empirical research by Elshoff and Marcotty [23] supports that readability is a vital factor in program maintenance. The authors suggested inclusion of a special development phase in the software development cycle to produce more readable code. To enhance code readability, Haneef [30] argued to dedicate a special readability group in the development team.

This study is principally inspired from the work [7, 5, 6] of Börstler et al. In these articles, the authors investigate problems and desired attributes of "exemplary" examples that can be applied to formulate a measurement framework to indicate quality of the example. *Understandability, effective communication*, and *adherence to external factors* are the three basic desired properties described by the authors. The article [5] provides a good picture to distinguish good and bad examples by comparing two example programs: "Beauty" and "Beast". The authors consider readability as a basic prerequisite for understandability and are of the view that readability index can be used to evaluate the quality of an example. Börstler et al. propose a software readability metric called "Software Readability Ease Score" (SRES) based on the idea of Flesch Reading Ease Score [25]. This study further defines SRES readability metric and provides its implementation as well as evaluation. The article [7] defines an evaluation instrument to distinguish between good and bad examples. The evaluation instrument provides three check lists to the reviewers or evaluators. These three checklists are based on technical, object-oriented, and didactical quality attributes. The authors performed a test of their evaluation instrument on a set of five example programs, and the results show that such an evaluation instrument can be useful though not reliable to distinguish good and bad examples.

Nordström's licentiate thesis [55] is closely related to this work. It asserts importance of examples in learning, discusses principles of teaching, provides a survey of related literature, helps to identify characteristics and principles of object orientation, reviews heuristics and rules for object oriented design, and proposes heuristics for designing small-scale object oriented examples. It uses the keyword *exemplary* to refer good examples. According to Nordström a good example should facilitate the learners to recognize patterns and distinguish an example's superficial surface properties from those that are structurally or conceptually important. There are six design heuristics described by Nordström, as listed below:

1. **Model Reasonable Abstractions:** This is the most important design heuristics, and it has two main implications for the reasonable abstraction. The first is that an abstraction should promote object orientation, i.e., it shows the basic characteristics of an object: identity, state and behavior. The second is that an abstraction should strive for non-artificial classes and objects, keeping the problem appropriate in size and complexity.

2. **Model Reasonable Behavior:** Reasonable behavior is one that is not overly simplified and does not have distracting artificial operations. The demand of reasonable behavior makes certain common habits inappropriate. One such common habit is the use of *print* statements for tracing, this confuses the novice of how things are returned from methods and spoils the idea of having I/O separated from the functional parts of a system.

3. **Emphasize Client View:** Services and responsibilities of an object should be defined independent of its internal representation and implementation of the attributes. The interface of a class should be designed carefully, as complete as possible, and be consistent to the problem domain.

4. **Favor Composition over Inheritance:** Inheritance1 distinguishes object orientation from other paradigms, but is difficult to comprehend by novices. It is difficult to model a reasonable abstraction and behavior with early introduction to inheritance. Composition can be used to exemplify reuse characteristic instead of inheritance, and the strength of inheritance can be exemplified later.

5. **Use Exemplary Objects Only:** An important design heuristics is to take care with the choice of objects involved. One of the problem with example programs is the use of only one or two class references and object instances, obscuring the principle of objects interactivity. Another problem is the use of more than objects of a same class without any logical reason or requirement. Examples using nameless objects and anonymous examples are difficult in comprehension therefore small scale example programs designed for novices should avoid these constructs.

6. **Make Inheritance Reflect Structural Relationships:** The example programs should not confuse the students in terms of structural and hierarchical relationships between base and derived classes. "To show the strength and usefulness of inheritance it is essential to design examples carefully" [55]. Behavior should guide the design of hierarchies and the relationship must be clear and easy to understand.

It briefly describes the different software metrics but does not provide explicit mapping between proposed heuristics and software metrics to have a quantitative measure of example quality.

Chapter 2

Program Comprehension

2.1 Introduction

Program Comprehension is a cognitive process as a result of which one learns, acquires knowledge and gains understanding. Rugaber [59] describes program comprehension as a process through which one acquires knowledge about computer programs. Whereas Mayrhauser and Vans [70] define program comprehension as "a process that uses existing knowledge to acquire new knowledge". In literature, Program comprehension is referred as *Program Understanding* also. Therefore, program understanding and program comprehension are the two interchangeable term in this study.

There is a plenty of work done related to program comprehension, as a result of which number of program comprehension models have been proposed. These models will be briefly described in the next section 2.2. An interesting fact to quote is that almost all the work done for program comprehension targets modification and maintenance. Around 50% of the maintenance effort is spent just to comprehend the program [70]. It is a genral observation that novices spend ample time out of their studies just to comprehend example programs. Therefore, it would be usefult to see how novice learners interpret and understand programming examples. Understanding the way students understand example programs may help to understand the problems faced by students, and address those problems by designing easy to comprehend examples. There is very little work done on program comprehension with respect to novice's learning. An important point is to see how program comprehension with respect to novice learners differs from program comprehension with respect to maintenance and modification issues. Can we apply the same process of program comprehension on a novice learner as well as on a professional programmer or not, what are the main differences? How students comprehend programs, and how can we measure student's understanding or learning index?

2.2 Program Comprehension Models

2.2.1 Brooks' Hypothesis based Model

Brooks' theory of program comprehension [9] is based on the idea of problem domain reconstruction. It describes programming as a process of constructing mappings from a problem domain to a programming domain. These mappings from a problem domain to the programming domain involve several intermediary knowledge domains. All these intermediary

knowledge domains encoded by a program are reconstructed during program comprehension. The reconstruction process is controlled by creation, confirmation and successive refinement of a hypothesis. Hypothesis is built, in a top down manner, from the learner's existing knowledge of the problem domain and are iteratively refined to match specific lines of code in the program. At the top there exists a primary hypothesis, created just by hearing a program name or some description, without having a look on the actual program. It describes the overall program functionality and can not be verified against the actual code. Therefore, primary hypothesis branches into several subsidiary hypotheses until a level is reached where a subsidiary hypothesis can be matched and verified or validated against program code or documentation. Programming beacons play an important role in hypothesis validation. The programming beacons lead towards success or failure of a hypothesis and creation or modification of a hypothesis.

2.2.2 Soloway and Ehrlich's Top-down Model

Soloway and Ehrlich [64] describe *Programming Plans* and *Rules of Programming Discourse* as important elements in program comprehension. *Programming Plans* are the code fragments that specify conventional program constructs, whereas *Rules of Programming Discourse* are the standard code conventions or patterns, that results in expectations about a program's behavior. Top-down program comprehension approach typically applies when the code or type of code is familiar [70]. So this model is more applicable in case of expert programmers than novices. Similar to the Brook's hypothesis model, a comprehender establishes a high level goal and that further branches to several sub-goals, based on the observed programming plans and rules of discourse. Throughout the process, a mental model is constructed that consists of a hierarchy of goals and plans.

2.2.3 Shneiderman's Model of Program Comprehension

Shneiderman and Mayer [62] describe program comprehension as a critical subtask of debugging, modification, and learning. This comprehension model is based on the hypothesis that a comprehender constructs a multilevel internal semantic structure of a program with the help of existing syntactic knowledge. Comprehension at the highest level aims at overall functionality of the program: what does it do? Whereas the low level deals with recognizing individual code lines or statements. The comprehension process starts by recognizing functionality of smaller code chunks and creating their semantic representation in mind. These low level chunks are then combined to form larger chunks, along with establishing their counterpart internal semantic representation. This process continues until the entire program is comprehended.

2.2.4 Letovsky's Knowledge based Model

Letovsky [44] describes a knowledge based cognitive model of program understanding. According to the model, programmers are knowledge based understanders who establish three main components: a knowledge base, a mental model, and an assimilation process. Knowledge base contains background knowledge and the expertise of a programmer that he applies to understand a program. Types of knowledge identified by Letovsky include programming language semantics, programming plans, programming expertise, recurring computational goals and rules of discourse. Knowledge base contains a large set of solutions for the problems that a programmer has solved in the past. All these knowledge types help a programmer to construct a mental model of a program. Letovsky has identified three layers of mental

representation: a specification, an implementation, and an annotation layer. Specification layer is at the highest level and is more global to represent overall goals of a program. Implementation layer is at the lowest level and represents individual data structures and operations in a program. Whereas the annotation layer attempts to link each of the goals in specification layer to its corresponding program constructs in the implementation layer. This process of constructing a program's mental model by applying a knowledge base on a program is termed as an assimilation process by Letovsky. The assimilation process can be top-down or bottom-up, as it depends on the programmer' s characteristics that how can he attain better understanding about a program.

2.2.5 Pennington's Model

In order to analyze the program comprehension process, Pennington [57] categorized program information into six groups: operations, variables, control flow, data flow, state, and function. Pennington's model of program comprehension states that there are two representations of a program that are constructed during the comprehension process. First representation is named as *Program Model* and it corresponds to program text and control flow. The second representation is named as *Domain Model*.

A Program Model is constructed before a domain model. The domain model uses the program model to create a functional representation of a program. It is developed using a bottom-up program comprehension approach, where programming plan knowledge and code beacons are used to find key code controls and operations. It starts at a micro level, constructing a representation for smaller chunks and then continues to combine the smaller chunks and their internal representation, until a program model for the whole program is built. Then it proceeds to build a domain model of the program. The domain model is also constructed in a bottom up comprehension style and requires the knowledge of real world problem domain. Instead of programming plans and text structures it uses domain plan knowledge to construct a mental representation and describes program code in terms of real world objects and operations.

2.2.6 Littman Comprehension Strategies

Littman [46] identifies two program comprehension strategies: *Systematic Strategy* and *As-needed Strategy*. A comprehension strategy followed by a programmer greatly influences the knowledge he or she acquires. Comprehension strategy, programmer's knowledge about the program, and modification or maintenance success are interrelated. Systematic strategy helps a programmer to understand how a program behaves. It aims at the global behavior of a program by tracing control flow and data flow paths through the program. Systematic strategy requires comprehensive execution of control flow and data flow routines and sub routines to determine the causal relationship among different components of a program. That causal relationship greatly helps to understand the overall program behavior.

On the other hand, as-needed strategy is more localized as it aims at the local behavior of a program. It does not involve comprehensive tracking of control flow and data flow. The comprehender focus only on selected routines or procedures within a program. According to Littman, programmers who adopt a systematic strategy obtain better understanding as compare to those who adopt an as-needed strategy. The suggested reason for failure in case of as-needed strategy is the lack of knowledge about causal interactions among components of a program.

2.3 Important Factors for Program Comprehension

From above described various program comprehension models, we derive that in order to understand a program, the comprehender applies his existing knowledge along with available beacons and learning aids. To identify the factors that play an important role in program comprehension, we have distinguished categories of external and internal factors.

2.3.1 External Factors

External factors are all those skills, knowledge and helping aids that are not a part of actual program code.

Existing Knowledge One of the most dominant factors in program comprehension is a comprehender's existing knowledge. The existing knowledge includes computing specific knowledge, problem domain knowledge, and all the general knowledge that supports to understand a program [44]. Computing specific knowledge mainly includes knowledge about problem solving, data structures and algorithms, programming plans, and language specific skills. Problem domain knowledge is also very important, as it supports in the program comprehension process. Those who are familiar with problem domain tend to understand the program better than those who are not familiar with the domain. Novices are generally believed to have no or little domain knowledge. This factor greatly affects the comprehension approach.

Personal attributes Personal attributes cover the qualities and characteristics of a person who is going to understand a program. These personal attributes include: experience level, intellectual capabilities, knowledge base, motivation level, and behavioral characteristics [3]. Experience leads one to a high level of expertise that highly affects efficiency of program comprehension [70]. These personal attributes vary person to person and even among the persons with a same number of years as programming or learning experience.

Documentation Well documented programs are likely to have better understanding. "Different kinds of documentation are helpful at different stages of comprehension" [18]. Documentation provides details about individual modules and program constructs but at the same time too much documentation can create problems as well [18]. In case if a program segment is not clear to understand, one may consult the documentation to know more about class behavior, as well as underlying data and operations details.

UML Diagrams UML diagrams, an object oriented design artifacts, are really helpful in case of object oriented program comprehension. These artifacts provide quick insight about program constructs and behavior, specially in case of a larger project with many interactive modules. Flow charts are easy-to-understand diagrams that represent a process or a program algorithm. These are effective tools to communicate and understand how a process works. Flow charts are generally used in system analysis, design and documentation. However, they are also helpful in program comprehension, and can be combined with other tools to understand a program.

2.3.2 Internal Factors

Internal factors are all those internal elements, attributes, or characteristics of a program that help to understand it.

Program Complexity Complexity is regarded as an intrinsic attribute of programs and, in general, can not be avoided completely. It affects program readability and comprehension [18], where complex programs are difficult to read and understand. For example, very large programs are difficult to read because of their size. There are several different types of program complexity discussed in Section 3.3

Beacons Beacons are those program constructs or elements that provide hints about underlying program data structures and associated operations [3]. Effective use of beacons in a program greatly helps to recognize as well as understand a program. *Swap* is the most commonly referenced beacon in the related literature. Appropriate identifier names that describe the purpose of their use also serve as beacons. For example, an identifier *print* for a method that prints some values. Identifiers that reflect their corresponding objects and action in the real world, are easier to understand. Following a standard naming convention with meaningful identifier names may ease the job of program reading and understanding.

Comments Appropriate use of comments and meaningful identifiers are beneficial to program comprehension. However, unnecessarily verbose comments may add to the complexity of a code [18]. A program with a high ratio of comments to the actual code may make actual program code difficult to distinguish, read and understand as well.

Indentation and Formating Style Text readability is greatly affected by its formatting style, font options, white spaces, and indentation style. Poorly indented and formatted texts are hard to read and understand as well. The same is true in case of a program code.

2.4 Differences between Experts and Novices

Experts and novices differ in a number of ways in their skills, expertise, experience level and knowledge. Therefore, they differ in their program comprehension process and strategies as well.

- In comparison to novices, experts make a better use of *Programming Plans* and *Rules of Programming Discourse* to comprehend a program [64]. *Programming Plans* are the code fragments that specify conventional program constructs, such as stereotyped search or sort algorithms. Whereas *Rules of Programming Discourse* are the well established programming patterns and practices that provoke expectations about the functionality of a program. Defining variable names that agree to their functionality is an example of a programing discourse rule.

- Experts make more use of beacons to understand a program as compare to novices [3].

- Experts' knowledge is hierarchically organized that gives them better processing capability than novices [20].

- Experts mostly use a top-down approach while novices generally adopt a bottom-up approach to understand a program [3, 54]. Novices read a program sequentially, line by line in a physical order, while experienced readers generally follow control flow while reading a program in a top-down manner [54].

– Novices use surface features of a program, e.g., syntactic structure, to understand a program while experts use program semantics to understand a program [3, 20].

According to Fix et al. [24], an expert's mental representation of a program exhibits following five abstract characteristics that are absent in the mental representation constructed by novices:

1. Expert's mental model is hierarchal and multi-layered.
2. It has explicit mapping between the layers
3. It is based on the recognition of basic patterns.
4. It is well connected internally.
5. It is well grounded in the program text.

2.5 How Do Novices Read and Understand Example Programs?

As discussed in the Section 2.4, novice learners differ from expert programmers in their skills, knowledge, experience, behavior, as well as program reading and comprehension strategies [54, 34]. Most of the classical models of program comprehension, described in Section 2.2, deals with experienced programmers are not exactly applicable to novice learners. However, Program comprehension, both in case of experts or novices, requires a basic mechanism of constructing mental representations by processing the coded information [54, 20].

Program comprehension is generally viewed as a text comprehension, as most of the studies on program comprehension considered a program as a special form of a text [20]. According to the Verhoeven's model of reading comprehension [69], reading of a text starts with the identification of individual words. This process is supported by the reader's knowledge of words, orthography, phonology, and morphology. Verhoeven quoted studies of eye movements to claim that even skilled readers fixate on most of the words they read and the fixation tends to be longer at the end of sentences. These longer fixations in the end aims at sentence comprehension based on word-to-text integrations, where a reader connects identified words to a continuously updated representation of a text. Sentence comprehension employs both sentence structure and word meanings to formulate hypotheses about the meaning of a sentence. A reader then combines the meanings of each sentence with the overall text interpretation established so far on the basis of prior text. Major models of text comprehension state that apart from the information present in a text, readers use their prior knowledge as well to construct new knowledge as a mental representation of a text. Two models of representations are constructed: a text model, and a situation model. The text model is a linguistic representation of a text that is isomorphous to the text structure [10]. It reflects text contents, i.e., micro-structure and macro-structure of a text. It is constructed successively from the verbatim representation [10] by extracting basic meanings from sentences based on the existing knowledge [69]. The situation model models text domain, i.e., what the text is about and is isomorphous to the situation described by the text. It is built using reader's existing knowledge about a particular domain [10].

Students are introduced with example programs as a concrete model to enhance their learning. What students have earlier learned during lectures or tutorials adds to their existing knowledge. Comprehension is actually a cognitive process to connect or match

existing knowledge upon new information [69, 54, 49]. Existing knowledge, organized in long term memory, servers as a key resource during program comprehension [63, 44, 9]. It includes both computing specific knowledge as well as general knowledge about problem domain and related concepts. Initially, students' existing knowledge about object oriented programming is almost zero or very low, that tends to increase gradually as students learn more and more object oriented concepts and do more practice with examples. To provide novice learners with sufficient knowledge, fundamental programming concepts are taught at first during lectures and tutorial sessions. The purpose of supplying the associated programming theory or lecture notes at first is to add into their existing knowledge. Otherwise it would be very difficult for them to understand program constructs and concepts without having required knowledge.

Along with their specific knowledge of object oriented programming, novices also apply their general knowledge about mathematics and computing while understanding an example program [70]. Knowledge about problem domain plays a significant role in comprehension. Students are more likely to have better understanding about more popular and common real world problems. Above described program comprehension models talk about the use of programming plans, beacons and other rules of programming discourse, however, students at junior level have no idea about these learning aids [3, 64]. They develop their sense of programming plans, use of programming discourse rules, and beacons to some extent as they learn and practice more.

Novices generally read a program sequentially, line-by-line in a physical order starting from the top or first line, similar to reading a book [54]. Using a sequential reading strategy, novices follow a bottom-up approach, i.e., low level details first, and general structure in the end, since they lack in skills and knowledge required for the top-down approach. They hardly follow control-flow or data-flow while reading a program. In the beginning, they generally treat example programs similar to an ordinary text [54] as they do not know any specific strategy for program reading. They follow their practice of simple text comprehension because they are quite good and use to that practice. They read lines word-by-word in smaller syntactic chunks to establish their understanding of a program. A "program model is created by chunking micro-structures into macro-structures and by cross-referencing" [70]. Program's mental model is the first inner representation that a novice learner builds as a result of the comprehension process. Novices typically use program syntax to comprehend; they hardly follow program semantics, programming plans, or algorithmic knowledge to understand a program. If a student has already seen a similar code excerpt or algorithm then he or she will already have a partial mental model of the program. This partial mental model is likely to facilitate the understanding process, but it may mislead the novices in case of a complex or a bad example.

Brook's comprehension theory of hypothesis creation, validation and successive refinement can be partially applied to novices. As they create a hypothesis about program behavior by reading textual details, domain description and program identifiers. As a general learning process, students learn example programs in parts, setting up a hypothesis and then gradually refining it as the cognitive process proceeds. However, Brook's comprehension theory can not be fully applied to novices, as they usually lack in their knowledge of programming plans, beacons and problem domain. Beacons and Rules of discourse e.g., coding standards, naming conventions have a minimal (non obvious) impact on novices. These beacons and rules of discourse definitely help novice learners as well, but they do not explicitly look for such cues to comprehend the examples.

Chapter 3

Quality and Complexity of Example Programs

3.1 What is Good and What is Bad?

It is bit tricky to define or find an exact definition of a "good" or "bad" example. Börstler et al. has defined a good example as: *"An example that relates to the needs of novices in programming and object-orientation"* [6].

We derive a following definition of "bad" example from an article [5] by Börstler et al.:

Bad Example: An example program that imposes risk of misinterpretations, erroneous conclusions, and leads towards misconceptions causing ultimate problems in learning.

The above definition can be extended to define a good example as: *"An example program that does not have any risk of misinterpretations, misconceptions and erroneous conclusions"*.

"Good" and "bad", the two mutually exclusive adjectives used to describe example programs, in fact, are the measures of program quality on an ordinal scale. ISO8402 defines quality as: *the totality of features and characteristics of a product or service that bear on its ability to satisfy stated or implied needs. Not to be mistaken for degree of excellence or fitness for use, which meet only part of the definition.* In the context of this study, the product or service is an example program. Classification of good and bad examples can be simplified by identifying quality factors or attributes. These quality attributes combined with above definitions, result in a more comprehensive definition of "good" example, as below:

Good Example: An example that truly satisfies desired quality attributes and learning needs of novices without any risk of misinterpretations, misconceptions, and erroneous conclusions.

So there are three major challenges for a good example.

1. The first one is to satisfy the desired quality attributes, and these quality attributes are discussed with more details in Section 3.2.

2. The second challenge is to meet learning needs, goals, and objectives. These learning objectives are generally defined by academics in agreement to the course curriculum [66] defined by the joint task force of the ACM and the IEEE Computer Society.

3. The third challenge for examples is to avoid any misconceptions, false conclusion or misinterpretations. This means an example should have effective demonstration of a key concept clearly reinforcing earlier taught principles without resulting any confusion or ambiguities.

3.2 Desirable Quality Attributes of Example Programs

The quality attributes of an example is a set of desired properties, that a "good" example program is required to satisfy. The IEEE-1061 standard defines software quality as the degree to which a software possesses a desired combination of attributes. These attributes can be mapped to define quantitative measures of quality and can be used to evaluate quality of examples as well. Börstler et al. [6] classify example quality attributes into three categories of technical, object-oriented and didactical quality. *Technical quality* deals with technical aspects and considers three quality factors: problem versus implementation, content, and style. Object-oriented concepts and principles are taken into account under the category of *object-oriented quality*. While *didactical quality* considers instructional design, comprehensibility, and alignment with learning objectives [7].

A list of example quality attributes is given below. Since example program is a special case of software, so most of these attributes are inspired from general software quality attributes.

1. **Readability:** Readability comes first. In academic perspective, it is the most important and basic quality attribute for example programs to follow. Readability means that a program is well written and is easy to read. If an example program is not even readable, how it can be understandable? Novices tend to understand programs by reading them in smaller parts, similar to reading an ordinary text. Therefore basic syntactical units of an example program must be easy to spot and recognize, so that students can easily establish a meaningful relationship between program components [5]. Inappropriate code indentation, bad choice of identifier names, meaningless comments and non standard code conventions are the major problems that make example programs difficult to read. Readability of an example can be improved by using meaningful identifier names, good use of comments, proper code indentation, following standard code conventions and removing unused or noisy code elements. Students' familiarity of the concept, background knowledge, personal interest and motivation are the few external factors that may affect readability of an example program.

   One may consider readability as a sub unit of understandability or communication, because of its close relationship to these attributes. However, realizing the importance of readability, we count it as an essential prerequisite and an explicitly separate quality attribute. To have a quantitative measure of examples' readability, Börstler et al. [5] propose a readability metric, *Software Readability Ease Score (SRES)* based on the Flesh Reading Ease Score (FRES). SRES counts lexical units such as syllables(lexemes), words (tokens/syntactic categories) and sentences (statements or units of abstraction). Chapter 4 will provide more insight about program readability and SRES.

2. **Understandability:** A good example must be understandable by students and obviously by computers [5]. Understandability is a cognitive process during which students employ their knowledge, skills and available resources to recognize and understand the elements of an example program. It comes after the basic quality attribute of

readability. Readability is both a physical and a mental activity [58], whereas understandability is purely a cognitive process. Example programs having poor readability, meaningless identifiers names e.g., *A, B, x, y*, unnecessarily complex code structure and non standard code conventions make understanding really a difficult task for novice learners. Program comprehension is the other name for program understanding in literature, that has been discussed in Chapter 2. In future we aim to study cognitive processes involved in example program comprehension from students' point of view so that we may formulate an effective measure of program quality. There are several internal and external factors that affect understandability of examples, see Section 2.3

3. **Simplicity:** A good example should be as simple as possible, suitably abstract, neither too complex nor too much simple. It should not expose more or less elements, e.g., lines of code, concepts and identifiers, than what is actually required [7]. However readability, understandability, and other quality attributes should not be compromised for the sake of simplicity. Simplicity should not be misused to write too much abstract examples without any real life application. "Too abstract" or a simple example is one of the four problems reported by Malan and Halland [48]. In order to attain simplicity, sometimes example programs are written without any particular application or relevance to any real life objects' interaction. Such examples may help students to learn syntax of the programming language but they do not really help them to learn the principle of object oriented or any other programming paradigm of choice. Therefore simplicity here does not mean that educators should use too much abstract example, rather simplicity here means that examples should not be unnecessarily complex above the cognitive capabilities of target students.

4. **Communication:** A good example must have effective explanation of the key concept, reinforcing earlier taught principles without resulting any confusion, ambiguities or misinterpretations [5]. One of the essential properties of a good example is that it does not let students to end in erroneous conclusions. Rather a good example should facilitate students to clearly comprehend complex concepts and principles. It might be difficult for novices to learn different uses of inheritance or polymorphism by reading lecture notes or text book. But a purposefully designed example can effectively communicate and help students in learning such key concepts and their principles.

5. **Consistency:** A good example should apply the principles of a particular programming paradigm in a consistent manner. In case of object oriented paradigm, every example should take care of the object oriented design heuristics and guidelines. For example, an example program to demonstrate how methods operate on data fields, should implement the operations according to the principles of encapsulation. Malan and Halland [48] quoted an example program where member data is being passed to member methods of a same class as a formal parameter, against the principles of object oriented programming. Therefore both educators and students should be careful with their choice of example programs to make sure that example programs are consistent with respect to all learning objectives and principles of a particular programming paradigm.

6. **Reflectivity:** A good example should reflect the problem domain in question [7]. In case of object oriented programming, a program modules and its member data fields as well as methods should reflect real world object's states and behavior. Identifier names should possibly match to the objects from real world problem domain. It helps to improves quality of an example program by making it easier to read and understand. Students can easily spot and recognize syntactic units to establish a relationship between them and this makes understanding of the concepts easier.

7. **Beauty:** Webster's online dictionary defines beauty as a quality that gives pleasure to the senses. It is simply natural to feel and realize the beauty, but hard to describe it in words. Object's structure, appearance, design and inherent characteristic plays vital role to make it beautiful. Beauty of an example program is that along with satisfying all other quality attributes, it should be attractive and interest provoking. Students should tend to play and learn from example programs without feeling any stress.

Although the above enumerated properties are comprehensive and covers a wide spectrum, this is not likely to be a last and a final word.

3.3 Program Complexity

Program complexity is regarded as an intrinsic attribute of a program and, in general, can not be avoided completely. It affects readability as well as quality of a program making it difficult to read and understand [18]. Complexity is analogous to "beauty", as they say *"beauty lies in the eyes of a beholder"*. The same is true for complexity, because the level of complexity perceived is dependent on the person perceiving a program [14]. An example program that looks simple to some of the students may create problems in understanding for the rest. Basili [4] defines complexity as *a measure of the resources expended by a system while interacting with a piece of software to perform a given task*. According to Kearney et al. [36], If the interacting system is a computer then complexity is characterized in terms of execution time (time complexity) and storage required to perform the computation (space complexity), but if the interacting system is a programmer then complexity can be characterized in terms of difficulty of performing tasks such as coding, debugging, testing, or modifying the software.

3.3.1 Program Complexity Measures

The commonly reported types of program complexity are: time complexity, space complexity, computational complexity, structural complexity, and cognitive complexity. Time complexity is defined as a measure of the time a program takes to execute as a function of the input size. In context of program understanding, it can be quantified as a time required to understand a program. Space complexity is defined as a measure of the storage space in memory consumed by a program at run time. Whereas computational complexity is defined as a measure of the number of steps or arithmetic operations performed by a program [1]. Structural complexity is based on the program's intrinsic attributes: syntactic structure, program size, control flow and decision structures.

Halstead's software science [29] and McCabe's cyclomatic complexity [50] are the mostly used and good choices to measure program complexity based on a program's syntactics and decision structure. Halstead's measures are the functions of the number of operators and

operands in a program. McCabe' cyclomatic number is the maximum number of linearly independent execution paths through the program [36]. While measuring complexity, one should also consider the impact of object orientation, in case of object oriented paradigm. Inheritance, polymorphism, data encapsulation, and abstraction have a dominant affect on program structure, that impacts structural complexity of the program as well. One may use *Depth of Inheritance Tree (DIT), Coupling between Objects (CBO),* and *Lack of Cohesion in Methods (LCOM)* measures defined by Chidamber [16] to evaluate structural complexity.

Cognitive complexity deals with cognitive problems in learning. Amount of Information, familiarity and recognizability of the concepts, functional and variable dependencies, cuing level, ambiguity factor, and expressional complexity are the dominant factors adding towards cognitive complexity of an example. Cant et al. [14] treat cognitive complexity as those software characteristics that interact with programmer characteristics to affect the level of resources used. The authors have described mathematical formulas to measure cognitive complexity of a program. In theory these measures of cognitive complexity are very comprehensive and logical, however bit hard to understand and implement. We are unable to find any system implementing these measures of cognitive complexity. We are interested to implement and evaluate these measures of cognitive complexity of Cant et al, in future.

Complexity of example program is affected by the principles of a particular programming language or paradigm as well. In case of object oriented paradigm, programs are modularized on the basis of objects, where modularization results in less chunking but more tracing efforts. The data encapsulation property of object oriented paradigm helps to reduce cognitive complexity by reducing both forward: to follow ripple effects; and backward tracing efforts:, to resolve variable dependencies [13]. Inheritance and polymorphism, being key concepts of object orientation, have almost balanced effect on example complexity. They increase cognitive complexity by making functional dependencies difficult to resolve, but at the same time they reduce structural complexity by improving semantic consistency and decreasing the number of lines of code. Inheritance results in functional dependencies, where a method implementing some behavior might be contained in the parent class, and so on. Method calls also become difficult to trace as a result of polymorphism.

3.4 Software Metrics : Measures of Example Quality

A software metric is a function or a measurement tool used to measure certain properties of a software. In an article [35], Kaner et al. define software quality metric as "a function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality".

A large number of software metrics and models [53, 13, 41, 11, 36, 32, 29, 50] have been proposed, and more or expected in future. In the former two Sections 3.2 and 3.3.1, we have discussed software quality attributes and complexity issues related to this study. In this section we provide a brief overview about a set software metrics commonly used to measure different attributes of software quality and complexity. This study is a part of on going research in "Computer Science Education" research group at Umeå University about the quality of object oriented example programs used in academia. These metrics have been investigated in an earlier work [5] to evaluate understandability of example programs. Therefore we use the same set of metrics to compare the results and strengthen the findings.

3.4.1 Software Reading Ease Score (SRES)

SRES metric [5] measures readability property of an example. It is a software oriented extension of a well known "Flesch Readability Formula" used to measure readability index of ordinary text. The next chapter describes both the SRES and Flesch reading formula in details.

3.4.2 Halstead's Metrics

Halstead's metrics [28] are one of the primitives, most successful, and widely used metrics of program complexity. Halstead metrics are based on simple static code analysis technique and works by interpreting a source code as a sequence of tokens and then classifying each token either as an operator or an operand. The four basic measures of Halstead's metrics are:

1. n1 = Number of Distinct Operators
2. n2 = Number of Distinct operands
3. N1 = Total Number of Operators
4. N2 = Total Number of Operands

Program Difficulty (PD), Program Volume (PV), and *Program Effort (PE)* metrics are based on the above basic measures of operators and operands. Since "Program Effort" is just a product of PD and PV, and can be computed from these two measures, therefore we only consider Program Difficulty and Program Volume metrics to capture the complexity index of a program.

An important issue with Halstead metrics is that there is no standard definition of operators and operands, i.e., what to count as an operator and what to count as an operand remains an open question. A well defined counting strategy is required to satisfy this question. Therefore we have explicitly described a counting strategy, Section 4.6, for the Halstead's measures. Since off-the-shelf available metrics tools have their own interpretations for operators and operands, so we implement our measurement strategy for Halstead's metrics to be more precise in results.

3.4.3 McCabe's Cyclomatic Complexity

We apply McCabe's cyclomatic complexity metric [50] to measure the structural complexity on a program. It is one of the widely used and well acknowledged complexity metrics. Cyclomatic Complexity index $v(G)$ is a measure of the complexity of a method's decision structure. It is the number of linearly independent paths and therefore, the minimum number of paths that should be tested. A threshold value for $v(G)$ is 10 however its value greater than 7 may indicate an overly complex method.

3.4.4 Lines of Code (LoC)

LoC is a simple static way of measuring program size that can be interpreted to estimate the efforts or resources required to read, understand, edit as well as maintain a program. Apparently LoC looks a simple way to go. However the problem is what to count as a line: Should we include or exclude the empty lines and the lines containing comments? Should we count physical lines of text or should we figure out for the logical lines of code?

Answers to such questions lead to different variants of LoC. For our studies we use an LoC variant that counts effective physical source code lines only, excluding the comments and empty lines. Such variant of LoC is referred as 'SLoC' (Source Lines of Code) or 'eLoC' (Effective Lines of Code) metric in literature as well. We use this variant of LoC because our implementation of SRES does not considers empty lines and comments. Therefore to study the true correlation between SRES and LoC we have opted for LoC count without empty line and comments. However we acknowledge the impact of empty lines, whitespace, and comments on the readability, and in future, we aim to investigate on these static properties as well.

3.4.5 Cyclomatic Complexity per LoC (CC/LoC)

CC/LoC is a static measure of average cyclomatic complexity number per LoC. Here 'CC' is the sum of the cyclomatic complexities for all methods declared in an example program. Threshold for this metric is 0.16 [43].

3.4.6 Lines of Code per Method (LoC/m)

LoC/m metric, as the name implies, measures average code size per method in terms of LoC. Threshold for this metric is 7 [43].

3.4.7 Average Method per Class (m/c)

m/c metric, as the name implies, measures average number of methods per class in a program. Threshold for this metric is 4 [43].

3.4.8 Weighted Method Count (WMC)

WMC is a product of above three metrics (i.e., WMC = CC/LoC * LoC/m * m/c). Threshold for this metric is 5 [43].

Chapter 4

Program Readability

4.1 Introduction

So far we have achieved the first two goals, as enumerated in Section 1.1.2, of this study . Chapter 2 and 3 have discussed issues related to *comprehension, quality, and complexity* of programs. Chapter 2 describes various models of program comprehension and attempts to build a theory of *how do novices understand example programs.* In chapter 3, a comprehensive set of desirable quality attributes is proposed. All the attributes described in Section 3.2 are important to study and investigate their impact on the quality of example programs. However for the sake of simplicity and due to time limitations, we decided to begin with the prime attribute of *readability.* So from this point on ward we will focus only on readability attribute, and will try to accomplish the last two goals of this study by implementing a readability based tool to evaluate example quality.

4.2 Readability in General

Readability is simply the ease of reading and understanding a text. It is what makes a text easier to read and understand consequently [21]. Hargis et al. [31] define readability, the "ease of reading words and sentences", as an attribute of clarity. While George Klare [38] defines readability as "the ease of understanding or comprehension due to the style of writing".

Above definitions give us a clear picture that readability is closely related to comprehension or understanding of the text. It is the contents (words and sentences) as well as writing style or format that improve or worsen a text readability. Research on ordinary text readability started in 1920's. Classical readability studies began with an aim to develop practical methods to match reading materials with the abilities of its readers [21]. A number of readability measure and formulas were defined, but only few succeeded to confirm validation standards. Few of the most popular readability formulas include: Flesch's Reading Ease Score [25], Dale-Chall's Readability Formula [17], SPACHE Readability Formula [65], Fry Graph Readability Formula [26], SMOG Grading [51], Cloze Procedure[67], Lively-Pressey's Formula[47] and Gunning's Fog Index (or FOG) [27].

4.3 Flesch Reading Ease Score - FRES

The Flesch formula is one of the most successful readability formulas designed to measure readability for adult materials [2]. It was proposed by Rudolph Flesch [25], a writing consultant and well known supporter of the Plain English Movement. Simplicity and accuracy are the two main characteristics of the Flesch reading ease score. Unlike other formulas, it is easy to calculate and is regarded as more accurate readability index. Total number of *words, syllables and sentences* are the basic counts of the formula. Then it uses average sentence length and average number of syllables per word to compute a final readability score for a given text. The original Flesch Reading Ease Formula is as below:

$$R.E. = 206.835 - (0.846 * wl) - (1.015 * sl) \tag{4.1}$$

Here:
$R.E.$ = Reading Ease
wl = Word Length (The number of syllables in a 100 word sample).
sl = Average Sentence Length (the number of words divided by the number of sentences, in a 100 word sample).

Below is the modified form of the formula in case of text having more than 100 words:

$$R.E. = 206.835 - (84.6 * ASW) - (1.015 * ASL) \tag{4.2}$$

Here:
ASW = Average Number of Syllables per Word (total number of syllables divided by the total number of words).
ASL = Average Sentence Length (the number of words divided by the number of sentences).
Constants in the formula are selected by Flesch after years of observation and trial [2].

The R.E. value ranges from 0 to 100 and higher value implies easier the text is to read. Abram and Dowling [2] use following interpretations for FRES, originally specified by Klare and Campbell [12].

R.E. Range	**Description**	**Suitable Reader's Grade**
0-29	Very Difficult	College graduate
30-49	Difficult	Completed high School
50-59	Fairly Difficult	Some High School
60-69	Standard	8th Grade
70-79	Fairly Easy	7th Grade
80-89	Easy	7th Grade
90-100	Very Easy	Below 6th Grade

Table 4.1: Flesch Reading Ease Scores - Interpretations

Using FRES, *mean readability* of a book can be accurately determined by analyzing seven random samples drawn from the book [2]. Due to its simplicity and better performance, FRES has been adopted as a standard test of readability by a number of government agencies, including The U.S. Department of Defense [11]. Most of the popular word processing programs, e.g.,MS Word, Google Docs, WordPerfect, and WordPro, have integrated FRES measure as a readability index.

4.4 Program Readability

Program readability is actually a judgment about the understanding of a program [11]. It is one of the prime program quality attributes described in the previous chapter. In an academic environment where students use example programs as a learning tool, readability of the examples must be good and helpful in understanding. The research on readability of ordinary text shows that a text with low readability is difficult to understand as well. The same principle can be applied on computer programs as, *a program that is hard to read, is likely to be hard to understand.*

A computer program or software, in terms of a set of instructions, is a special type of text with its own semantics and syntactic rules. The program semantics and syntactic rules are defined by a particular programming language. A hypothesis behind program readability is that *if we treat a program code as a plain text then we can apply the text readability measures on computer programs as well.* This means the idea of ordinary text readability can be extended to *program readability*, and we can use the well established and validated readability formulas to measure program readability as well. The results obtained from program readability measures then can be used to predict difficulty or complexity index of a program, i.e., how much effort or time will be required to read or understand a program. It will be useful in academic perspective as well. It may help the teachers and students to evaluate example programs quoted in text books or lecture notes to match their level of readability.

Most of the classical readability formulas, including FRES, are based on the count of lexical tokens or entities, e.g., total number of words, unique words, sentences, syllables, paragraphs. In order to apply readability formulas to computer programs, one have to find the equivalents of these lexical entities for a program text. Programming languages at present are not exactly same as natural languages are, however the basic lexical units are similar. They have their own set of characters equivalent to alphabets, keywords and user defined identifiers equivalent to words, statements equivalent to sentences, block structures equivalent to paragraphs or sections, and modules equivalent to chapters. Static code analysis techniques and tools can be used to count these static code elements. As a part of this study, we have developed a stand alone application to measure software readability on the basis of software readability metrics described in the next section.

Program complexity and readability are closely interrelated, however they are not exactly same. Complexity is an intrinsic or essential code property based on the problem domain and it can not be avoided completely in all scenarios. Whereas readability is an accidental property that can be avoided independent of the problem domain or problem complexity [11]. Readability is a static measure based on independent individual elements, such as identifiers, statements, comments, indentation, and program style. Whereas complexity depends on both static components as well dynamic interactions among program components.

4.5 Software Readability Ease Score - SRES

As the name implies, *Software Readability Ease Score (SRES)* is a measure of program readability proposed by Börstler et al. [5]. The basic idea behind SRES is Flesch's reading ease score [25], and we can say SRES is a software oriented extension of FRES. SRES works by interpreting program's identifiers, keywords and other lexical tokens as words, its statements as sentences, and word's length as syllables. A program readability formula can be defined on the basis of number of lexemes, statements, and modules declared in a program. At present we use *Average Sentence Length (ASL)* and *Average Word Length*

(AWL) as program readability index. Lower values for ASL and AWL imply the program is easier to read, because of shorter sentences and words length; and higher values indicate the program is difficult to read and understand. For more details about what we count as words, sentences and syllables, please consult SRES counting strategy, Section 4.5.1.

4.5.1 SRES Counting Strategy

1. Each space separated word, including java keywords and user defined identifiers, is counted as one word. In natural languages a word in its simple form is is interpreted as a contiguous sequence of alphabetic characters. In a recent study [19], Delorey et al. discuss several definitions of a word in programming languages ad different levels of abstraction. The simple form of word counting described by Delorey et al. is the one defined at the lexicographic level of abstraction, according to which "word" is a string of contiguous non-white-space characters. We could not find any specific definition of word to be counted by Flesch's readability formula, and presume it to be simply a continuous sequence of non-white-space characters.

2. In stead of syllables per word, we count word-length, i.e., the number of characters in a word. To simplify the task of counting syllables per word, we use word-length instead of syllables. The analogy is based on a general rationale of words with more syllables have more number of characters. For example, a java keyword 'new' is a monosyllabic and has less number of characters than a disyllabic keyword 'super'.

3. To measure the number of sentences, we use semicolon ';' and block symbol '{ }' to mark a sentence. The idea is based on the theory, Section 2.5, of how do novice learners read and understand a program. Expressions (core components of statements), statements, and blocks (set of statements) are the fundamental units to constitute a program code. As a syntactic rule, every java statement ends in a statement terminator operator, a semicolon ';'. Novice learners have a natural tendency of reading a program code in a same way as they read ordinary text. Therefore they treat ';' as '.' in the perspective of ordinary text and read program statements word by word.

   a. Novices treat a statement terminator operator, a semicolon ';', equivalent to full stop in ordinary text to mark the end of a sentence. Therefore we opt to use semicolons to find and count sentences in a java program, with few exceptions described below.

   b. In case of a block structure, e.g., control statements, class, interface, methods, and static code blocks, curly braces '{ }' are counted as a compound sentence that may contain several other sub sentences. In natural languages there is also a notion of complex or compound sentences that are longer than the average sentence length and may have several sub-units separated by punctuation marks.

      For example:

      ```
      while (boolVal) {
              doSomething();
         }
      ```

      There are two sentences in the above code snippet. The first one marked by ';' and the other by '{ }'.

c. Java allows to code control statements, *if, else, for, while, and do-while*, with or without block '{ }' symbols. It is a common practice to omit block symbols if a control statement affects only a single statement. Similarly in case of selection or iterative control statements if there is only one underlying statement, and no block '{ }' used then the control statement and the underlying statement both are combined to form a single sentence. Figure 4.1 shows a java example where loops are being used without block symbol '{ }'.

```
//Example: without block symbols

class NoBlockControl{
    public static void main(String []args ) {
            int a=0;

            for(; a<5; a++) //for and print statement constitute a single sentence
                    System.out.println("For Loop "+a);

            a=0;

             while (a<5) //while and print statement constitute a single sentence
                    System.out.println("While Loop"+a++);

            a=0;
            do  //do and print statement constitute a single sentence
                    System.out.println("Do Loop "+a++);

            while(a<5); //while is counted as a sentence, because of ; in the end.

      } //

    }
```

Figure 4.1: Example: without block symbols

4. In case of a *for* loop, "initialization, test, and update" parts altogether constitute a single for statement, therefore we count them as a part of single sentence. We do not count semicolons at the end of initialization and test condition as separate operators, since they are the parts of a relative long or complex statement. The reason to do not count them as separate sentences is that originally they are a part of for statement and can be empty, as in below code snippet. And the impact of difficulty added by these elements is counted by resulting a longer sentence. Since they all together are counted as a single long sentence that results in higher *ASL* score. Further if they exist in a complex combination using logical AND, OR operators, their complexity is

taken into account by increased word count, word length, and sentence length. All these basic measures ultimately results in higher *SRES* scores.

```
//Example:  For loop variations
class ForLoop{
public static void main(String []args ) {
int a=0;
for( ; a<5; ){
System.out.println("For Loop without internals "+ a++);
}

for (int i = 0, x=1; i < 5;  i++, x = x * i){
System.out.println("x= "+ x);
}
}//main
}//class
```

5. We do not count statement terminator ';' as well as comma ',' separators as a word or a part of word-length, we consider semicolon ';' just as a sentence marker. Because these are generally treated as punctuation marks to separate words, clauses, or sentences. They are not qualified to add cognitive load or difficulties in reading, therefore counting these operators as independent words (with shorter length of '1') will result in low *SRES* score. Another view about these operators is that these actually make reading of a text easier by clearly marking boundaries and among words, phrases and sentences. So it might be interesting to study the helping impact on readability added by these or any other parts of a program.

6. An empty statement (just a semicolon ';' and nothing else) is counted just as a one word of length 1, and a single sentence. Since ';' itself appears as an independent statement and not as a part of any other statement, therefore we treat as a simple word and sentence. For example:

```
  class Empty{
;  // empty statement - counted as one word of length 1 and a sentence.
}
```

7. We assume that nested block structures like { { } } and nested parenthesized expressions such as: `a*(b+(c-d))`, are more complex in comprehension than { } { } or simple expressions without any parenthesis or block symbols, such as: `a*b+c-d`. This implies nested blocks should get more scores indicating more "difficulty". Therefore in case of nested blocks and parenthesis, complexity impact is taken into account based on the level of nesting involved. The nesting level starts from 0 and goes upto n; and the SRES measures of wordsCount and WordLength are updated according to following equations:
wordsCount += (nestingLevel * 2)
wordsLength += (nestingLevel * 5)

   The constants 2 and 5 are kind of weight values, that can be adjusted later to reflect appropriate level of complexity.

8. Code statements that involve access to class members (fields or methods) are relatively complex than the simple arithmetic expressions involving plus or minus arithmetics. As these statements interrupt control sequence and involve chunking and tracing efforts, therefore we need to define special SRES counting rules for such statements.

   In case of Object access operator (dot), counting rule works as follow:

```
objRef.field = value;
objRef.method();
```

   - The identifier 'objRef' , before dot operator is counted as one word, with its specific length.
   - Dot operator is counted as 2 words with total length 2.
   - The identifier for member field or method, after dot operator is counted as 2 words, and its corresponding length is multiplied by 2. (parenthesis in case of method call are not counted).
   - The assignment operator '=' and assigned value are counted as normal, one word with their corresponding length.

   For more details about this rules, please see the example below:

```
class ScopeTest {
int test = 10;

 void printTest() {
   int test = 20;
   System.out.println("Test:"  + test); // Total words: 12
/*
The 2 dot operators are counted twice in words count and their length is also counted
twice,  same is for followed identifiers "out" and "println", each is counted twice for
both word count and word length. However the reference identifier "System" and arguments
within parenthesis, including the "+" concatenation operator are counted as normal
counting rule of one word with its corresponding length.
*/
 }

 public static void main(String[] arguments) {
   ScopeTest st = new ScopeTest();
   st.test=30; //Total words: 7
/*
The dot operator is counted twice in words count and its length is also counted twice.
While the object reference identifier "st", assignment operator "=" and the value assigned
are counted as normal rule of one word for each word/operator with its specific length.
*/
   st.printTest();
}// main

}// class
```

Note: Above rule is not applied to dot operator in case of dot qualified Identifier name in import statement or package declarations. In Import and package declarations we count dot as one word of length 1.

9. We count a string literal as one word, and its complexity is measured by the string length, that means longer string literals will have correspondingly longer word-length resulting in higher SRES score. For example, in case of `System.out.println("DotThis.f()");`, the string literal counted as one word is "DotThis.f()". Since string literals in case of print statements are just printed as they appear in a code segment, therefore we count them as a single word without applying the special rule to count 'dot', as in the above described counting rule.

10. We count array operator '[ ]' as a word of length 2 in case of array declarations. Whereas in case of array access expressions, we count array index and array operator as a combined word with the resulting length combined the array operator and index.

4.6 Counting Strategy for Halstead's Measures

1. All program entities including header (package declaration, import statements), declarations (class, method, field declarations) are included. Only comments are ignored.

2. All the keywords are counted as operators.

3. All kinds of operators including following symbols are counted as operators:
';' (semicolon)
'.*' (import all, e.g., `import java.io.*`)
'.'(package separator)
'@interface' (in case of annotation declaration)
'@' (annotation)
'?' (wildcard in generics arguments)

4. Following delimiter operators are counted: `[ ],( ), { },< >`.

5. Parenthesis '( )' are counted as operators in case of grouping expressions, e.g (2+4)*8, and type casting, and not in in case of method declarations and calls, around the argument list.

6. Keyword "default" used in annotations' method declaration statement is distinguished from the "default" keyword used under switch statement.

7. Colon ':' used after case and default statement in switch, colon used after label declaration, and a colon used in assert statement, all are distinguished operators, and are counted as unique operators.

8. Primitive type identifiers (boolean, char, byte, short, int, long, float, double) and User defined types' identifiers are counted as operators, when they are used to specify data type of some value, e.g., in case of variable declarations, definitions, method arguments, return value etc. But in their own declaration (class, interface, enum, decalarations) identifiers for user defined types are counted as operands [52].

9. Unary '+' and '-' are counted distinctly from the binary addition '+' and subtraction '-' operators [52].

10. We make a distinction between postfix and prefix increment '++' operators, i.e., both are counted as distinct operators. Same is true for postfix and prefix decrement '–' operators.

11. We agree with Miller [52] to count local variables with same names in different methods (scopes) to be counted as unique operands. It is contrary to the counting strategy [60] defined by Salt. The reason to count them as unique operands is that we have to treat them separately in order to understand the program correctly based on scope.

12. We count method identifier as an operand in its declaration statement and as an operator in a call statement. In case of method overloading, each overloaded method call is counted as a distinct operator [52].

Chapter 5

Implementation - SRES Measurement Tool

5.1 Introduction

In this study we develop an alpha version of the SRES measurement tool, developed using a parser generator tool, ANTLR, and Java platform. The current version implements Halstead's measures of software science [29] and measures of software readability ease scores [5]. Counting strategies implemented are described in the previous chapter. Main objective of this study is to determine and evaluate the quality of commonly used java example programs. SRES measures provides statistics about readability of the program whereas Halstead's measures focus on code complexity, and by combining both the results one can make a conclusion about quality of the given example. Readability is only a one of the several quality attributes described earlier in section 3.2. The problem in measuring other quality attributes is the difficulty in finding their corresponding quantitative or statistical measures.

At present we focus only on the source code of example programs, and do not consider textual description, lecture notes, program comments or any other form of documentation. Textual description or any help about example program is quite important as it impacts program quality and comprehension by facilitating students to clearly understand any complex part, acting as a shield against misinterpretation. However due to lack of time and resources we contended initially to focus only on the source code. SRES measure of readability is based on the number of lexical tokens and it does not consider the use of comments, code style and indentation, coding standards and naming conventions reflecting a problem domain plus any other factor affecting program readability. We assume that the programs are well indented, well formated, and have an appropriate level of comments inside.

5.2 SRES Measurement Tool

SRES measurement tool developed is a standalone java source code analyzer. It reads java programs and displays results for the implemented metrics of SRES and Halstead. At present it supports Java 1.5. It follows static code analysis approach performed with the help of *Java Lexer, Parser, and Tree Parser* components, automatically generated using ANTLR, parser generator tool. For more details about ANTLR, please see the section

5.2.2. To generate a lexer, parser, tree parser, and other auxiliary files ANTLR need a corresponding grammar file. For this purpose we reuse the java grammar files *Java.g and JavaTreeParser.g*, originally written by Dieter Habelitz. There grammar files are available for use at ANTLR home page `http://www.antlr.org/`. "Java.g" is a lexer and parser combined grammar file and ANTLR generates both *JavaLexer* and *JavaParser* against the specified lexer and parser rules.

Figure 5.1 shows there are four key components involve in the development of SRES.

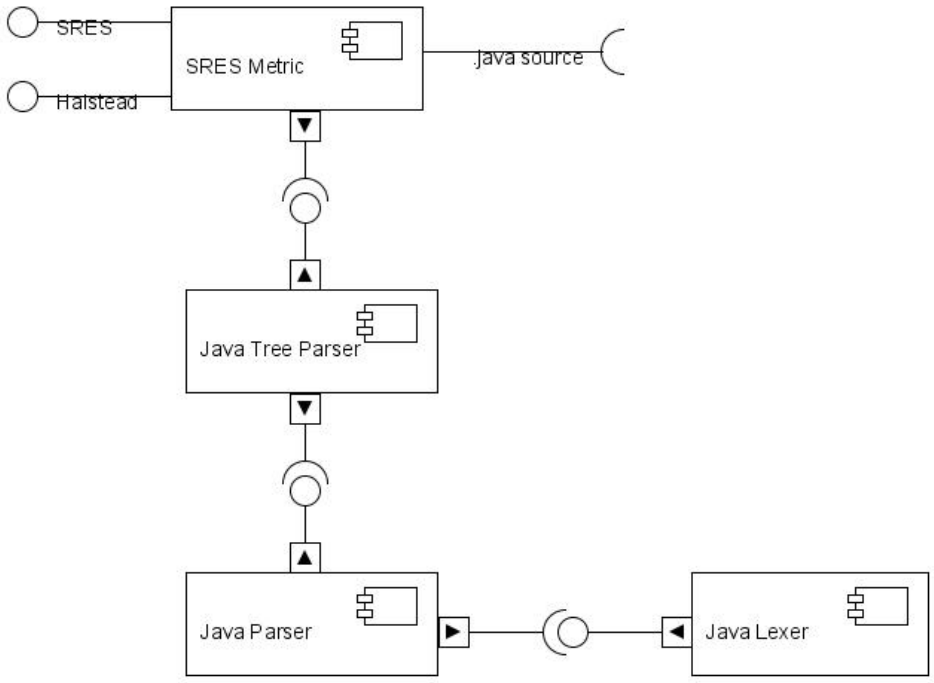

Figure 5.1: SRES - Components Diagram

SRES Metrics: This component is responsible to implement software readability ease score, halstead measure; and any other software metrics added in future. It also provides user interface to the SRES tool. As shown in the figure 5.1, it requires source java file as input and provides Halstead and SRES metrics results as output. The package "edu.cs.umu.sres" in the source code contains all the implementation classes for this component, where *Main.java* provides user interface, while *SRES.java* and *Halstead.java* implements corresponding software metrics.

Java Lexer: Lexer also known as a scanner or lexical analyzer is a program component that recognizes input stream of characters and breaks it into a set of vocabulary symbols or tokens. A token object returned by the lexer may represent more than one characters of same type, for example *INT* token represent integers. Each tokens consist of at least two pieces of information: the token type (lexical structure) and the text matched by the lexer [56]. These tokens are then pulled and used by the

parser component. Its implementation in source code is provided by *JavaLexer.java* class that is automatically generated by ANTLR against the lexical rules defined in the *Java.g* grammar file. Lexer rules match characters on the input stream and return a token object automatically. Below is an excerpt from lexer grammar, that defines lexer rules for hexadecimal, decimal and octal literals.

```
HEX_LITERAL : '0' ('x'|'X') HEX_DIGIT+ INTEGER_TYPE_SUFFIX? ;
DECIMAL_LITERAL : ('0' | '1'..'9' '0'..'9'*) INTEGER_TYPE_SUFFIX? ;
OCTAL_LITERAL : '0' ('0'..'7')+ INTEGER_TYPE_SUFFIX? ;

fragment
HEX_DIGIT : ('0'..'9'|'a'..'f'|'A'..'F') ;

fragment
INTEGER_TYPE_SUFFIX : ('l'|'L') ;
```

Java Parser: Parser is simply a language recognizer that. It applies grammatical structure defined as parser rules to a stream of tokens (vocabulary symbols) received from the lexer component. Both Lexer and Parser perform similar task, the difference is that the parser recognizes grammatical structure in a stream of tokens while the lexer recognizes structure in a stream of characters [56]. Along with language recognition, parser has the ability to act as a translator or interpreter, and can generate appropriate output or an intermediate data structure, usually a parse tree or abstract syntax tree (AST). An abstract syntax tree (AST) is a simplified tree representation of a source code, where each node denotes a particular syntactic construct matched in the source code. Parsing a code by constructing abstract syntax tree simplifies as well as accelerates the task, because required components are added to the AST, and the interpreter or parser has not to parse the extra lines of code.

Java parser component is implemented by *JavaParser.java* class automatically generated by ANTLR using the grammar file *Java.g*. It generates an abstract syntax tree against the matched rules and tree construction rules embedded in the parser grammar. Below is an excerpt from parser grammar, that defines parser rule for block statement, for complete description, please have a look at *Java.g* grammar file.

```
block
    :   LCURLY blockStatement* RCURLY
        ->  ^(BLOCK_SCOPE[$LCURLY, ''BLOCK_SCOPE''] LCURLY blockStatement* RCURLY)
    ;
```

Java Tree Parser: ANTLR has the ability to generate a tree parser automatically from a tree grammar. Tree grammar actually describe the structure of the tree (AST built by the JavaParser component). In order to evaluate program statements and compute the corresponding metrics values, actions are embedded as in-line code segments surrounded by curly braces {...}, as shown in the tree grammar excerpt below. This component is implemented by *JavaTreeParser.java* class automatically generated by ANTLR using a tree grammar file *JavaTreeParser.g*. It traverses a two-dimensional abstract syntax tree and computes the metrics values for each node matched in the AST. Below is an excerpt from tree parser grammar *JavaTreeParser.g* grammar file.

```
block
    :   ^(BLOCK_SCOPE LCURLY {blockList.add("{");} blockStatement* RCURLY
     {blockList.add("}");})
     {updateOPCounter("{}"); sentencesCount++;}
    ;
```

5.2.1 How it Works

Figure 5.2 shows the graphical user interface, the user interacts. The input area on the top allows user to select either a single java source file or an application folder containing more than one java source files. We assume that the source java files are error free: they have no compile time or run time errors. Results area below displays the end results, when a user clicks on either "SRES" or "Halstead" button. Figure 5.3 shows the result calculated. "Save" button in the down right corner lets the user to save the results calculated through a save dialog box, so that result log can be maintained.

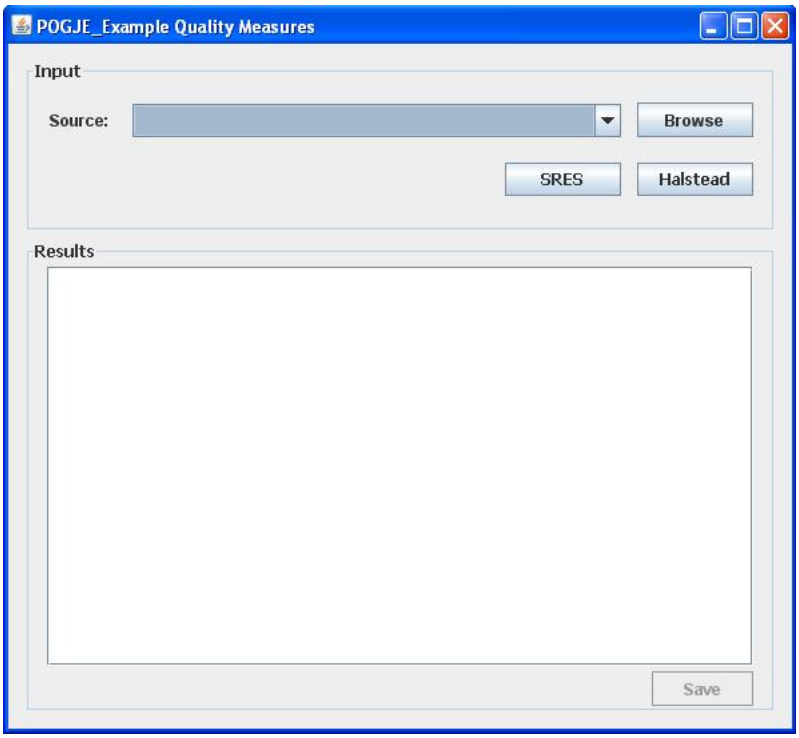

Figure 5.2: SRES - User Interface

5.2.2 ANTLR - Parser generator

ANTLR stands for "ANother Tool for Language Recognition" is used to construct other language processing tools such as translators, compilers or compiler compiler. It takes a grammar file as input and produces desired source code for language recognizers, analyzers and translators. ANTLR grammar file specifies target language to be matched with a set of rules and actions to follow. The rules are defined to match specific language phrases, whereas

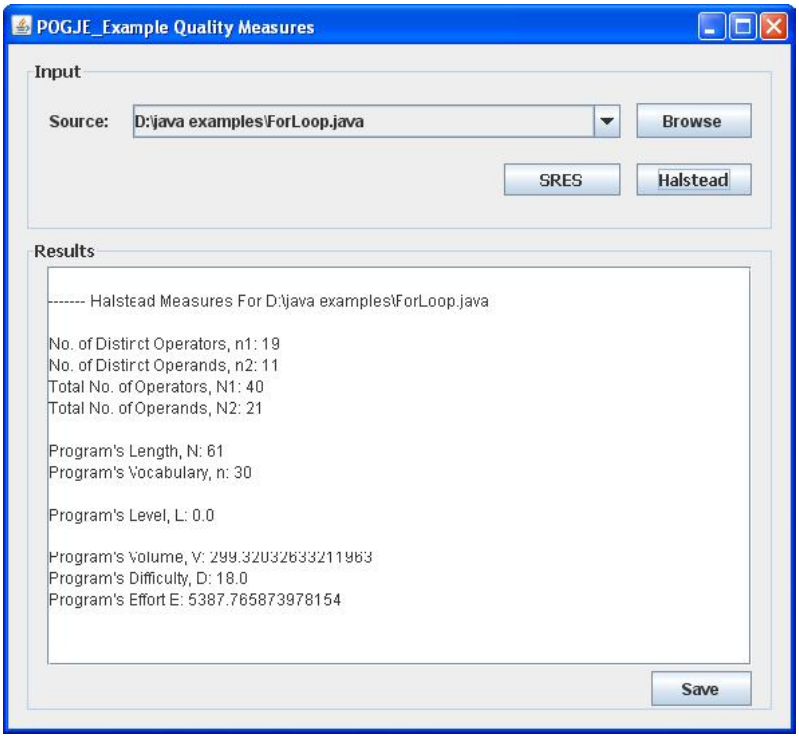

Figure 5.3: SRES - Results

actions are embedded to perform specific tasks based on the matched rule and context. For more information about ANTLR, please visit the home page `http://www.antlr.org/`.

5.2.3 Difficulties

As long as the theory is concerned both Halstead and SRES measures look pretty simple and easy to enumerate. But there arise some unanswered questions or difficulties once you decide to implement these measures. A key issue with Hasltead' measures is that there is no clear or standard definition provided for *what to count as unique operator, unique operand, total operator, and total operand* [60]. "The definitions of unique operators, unique operands, total operators, and total operands are not specifically delineated" [22]. Apart from the basic issue of defining operators and operands, we faced following specific questions during implementation.

- What to do with the pair of tokens that appear in combination, such as *if-else, do-while, try-catch-finally*?
- Should we include or exclude the commented code?
- Should we count the *apostrophe* symbols used to delimit character literals as an operand or an operator?

A counting strategy described in section 4.6 helps to fix above issues.

Similar problems are faced in case of *Software Reading Ease Score*. It looks really simple, appealing and easy to implement SRES measures, as we have to count only the basic entities: words, sentences and word's length. But there is no precise definition available for what to consider as a word and how to limit the sentences of a java program. We faced following specific questions while implementing SRES measures:

- What to count as syllables in case of a words (keywords or user defined) in a program?
- Should we calculate readability of the comments or just ignore them?
- Should we count an empty statement (just a semicolon ';' nothing else) as a word of length 1, and a sentence as well? See Rule 6 in section 4.5.1.
- Should we count initialization, test condition, and update segments of a for loop as 3-separate sentences or a part of a single sentence altogether?
- What to do with parenthesis ( ), as in case of "if, for, while, and method calls"? Should we count it as a separate word or a part of the word preceding it?
- What about an array operator [ ], should we count it as a separate word or a part of the word(identifier) preceding or following it?
- What about <> operator used in generics, should we count it as a separate word or a part of the word(identifier) preceding or following it?

A counting strategy described in section 4.5.1 helps to fix above issues to much extent.

Chapter 6

Experimental Results and Validation

6.1 Experimental Setup

We use a subset of java example programs selected by Börstler et al. [7] to apply and evaluate our metric of software readability, namely the SRES. The subset comprises 18 examples, as listed in in the figure 6.1, where the first sixteen examples, E1 to E16, are categorized as "mandatory". These examples programs are selected from 10 different commonly used java programming textbooks. SRES metric of program readability considers only source code of the example programs. Textual description given in textbooks or any other documentation is not considered.

We apply *Pearson's product-moment correlation coefficient*, denoted by r, to study the relationship between SRES and a set of selected conventional software metrics as described in Section 3.4. These metrics are summarized in the Table 6.1 as well. Threshold for these metrics are selected according to the likelihood of the capabilities and limitations of novice learners. SRES measure of readability is still on evolution and the threshold is based on average sentence length and word length suitable enough for the beginners.

6.2 Experimental Results

Figure 6.2 shows results for all the software metrics against each example program. Except HQS [8] in the last column, these results are captured using 3 different measurement tools: Krakatau Professional (see `http://www.powersoftware.com/kp/`), JHawk (see `http://www.virtualmachinery.com/jhawkprod.htm`), and the SRES measurement tool developed as a part of this study. The reason for using 3 different measurement tools is that we are unable to find any single open source tool to measure all the selected metrics collectively.

Most astonishing result is that none of the selected example programs satisfies selected threshold for all the measures simultaneously. However the examples E2, E3, and E7 satisfy all metrics except the average Cyclomatic Complexity per Lines of Code metric. These three examples are also rated among the high ranked examples by the human reviewers [8]. This implies that the metric results are in support of the quality impression by human subjects. Out of these three examples E2 is rated as the best in terms of all software metrics applied. On the other end E26, E12, and E10 are the example programs in decreasing order, that

ID*	Description	Author	Book Title	Edition	Pages
E1	First user defined class: TicketMachine	D. Barnes & M. Kölling	Objects First with Java	4th	pp 18-22
E2	User defined class: TrafficSignal.	Nino & Hosch	Introduction to Programming and Object Oriented Design Using Java	3rd	pp 85-92
E3	First user defined class: BankAccount	C. Horstman	Big Java	3rd	pp 85-90
E4	First user defined class: Thermometer .	D.S. Malik, R.P. Burton	Java Programming-Guided Learning with Early Objects	1st	pp 185-192
E5	First user defined class: Student.	E.S. Roberts	Java-An Introduction to Computer Science	2nd	pp 190-198
E6	First user defined class: Dice.	R. Bravaco, S. Simonson	Java Programming-From the Ground Up	1st	pp 404-408
E7	Inheritance: SavingsAccount inherited from BankAccount	C. Horstman	Big Java	3rd	pp 438-442
E8	First example of inheritance: EmployeeWithTerritory inherited from Employee	J. Farrell	Java Programming	5th	pp 370-375
E9	Interacting classes: Pile and Player.	Nino & Hosch	Introduction to Programming and Object Oriented Design Using Java	3rd	pp 123-135
E10	Composition and inheritance: Train, TrainCar, and Boxcar	E.S. Roberts	Java-An Introduction to Computer Science	2nd	pp 332-338
E11	First example of polymorphism: Cake, CircularCake, and RectangularCake	D. Riley	The Object of Java	2nd	pp 386-395
E12	First example of polymorphism: Firm, Staff, StaffMember, Volunteer, Employee, Executive, and Hourly.	J. Lewis and W. Loftus	Java-Software Solutions	6th	pp 515-527
E13	First example of 'if': Driver	D. Riley	The Object of Java	2nd	pp 263-265
E14	First example of 'if': Age	J. Lewis and W. Loftus	Java-Software Solutions	6th	pp 241-243
E15	First example of 'while': Ch6SleepStatistics	C.T. Wu	A Comprehensive Introduction to Object-Oriented Programming with Java	1st	pp 309-310
E16	First example of 'for': Investment	C. Horstman	Big Java	3rd	pp 236-241
E21	First user defined class: Die.	J. Lewis and W. Loftus	Java-Software Solutions	6th	pp 190-194
E26	First example of composition/collaboration: ClockDisplay and NumberDisplay	D. Barnes & M. Kölling	Objects First with Java	4th	56-71

* Example ID's are the same as being used by Börstler et al. in their on going research on quality of object-oriented example programs.

Figure 6.1: Example Programs used in Evaluation

Metric	Description	Threshold	Tool Used
LoC	Source Lines of code: number of source code lines, excluding whitespaces and comments.	30 (User Defined)	Krakatau Professional
PD	Halstead's Program Difficulty, A measure of how difficult a method code is to understand.	30	SRES Measurement Tool
PV	Halstead's Program Volume: Halstead Volume for a method. Calculated as: $V = Nlog_2n$.	300	SRES Measurement Tool
SRES -ASL	Average Sentence Length.	5	SRES Measurement Tool
SRES -AWL	Average Word Length.	6	SRES Measurement Tool
TCC	Total Cyclomatic Complexity: Cyclomatic Complexity (v(G)) is a measure of the complexity of a method's decision structure. It is the number of linearly independent paths and therefore, the minimum number of paths that should be tested. A v(G) value greater than 7 may indicate an overly complex method.	10	JHawk
CC/LoC	Average CC per LoC, where CC is the sum of the cyclomatic complexities of all methods.	0.16	Self Computed
LoC/m	Average LoC per method, where m is the number of methods.	7	Self Computed
m/c	Average number of methods per class, where m and c are the number of methods and classes, respectively.	4.	Self Computed
WMC	Weighted Method Count, the product of the three previous measures.	5	Self Computed
HQS	Human Quality Score: In an ongoing study [8], Börstler et al. compute HQS based on the reviews by 24 individuals which review each example using an *Evaluation Instrument* described in the article [8].	Maximum = 30	

Table 6.1: Selected Measures for Correlational Study

have most violation of the selected metrics. Results of SRES measure are also interesting to observe. Four examples, E1, E10, E11, and E14, have SRES-ASL greater than 5 (threshold), but all of these examples have no problem with the SRES-AWL threshold. SRES-AWL measure is above the threshold of 6 for four other example programs, E3, E4, E5, and E7. There are 8 out of 18 (44%) example programs that fail to satisfy either *SRES-ASL* or *SRES-AWL* measures. According to our results, SRES threshold violation set for AWL and

Program	LoC	PD	PV	SRES-ASL	SRES-AWL	TCC	CC/LoC	m	c	LoC/m	m/c	WMC	HQS
E1	35	17	530.00	5.17	4.74	5	0.14	5	1	7.00	5.00	5.00	14.43
E2	16	12	291.43	4.08	4.67	3	0.19	3	1	5.33	3.00	3.00	15.71
E3	20	11	294.00	3.64	6.25	5	0.25	5	1	4.00	5.00	5.00	17.71
E4	18	11	216.00	3.17	7.34	5	0.28	5	1	3.60	5.00	5.00	9.00
E5	32	11	558.00	3.52	6.17	8	0.25	8	1	4.00	8.00	8.00	13.86
E6	30	22	478.00	4.10	3.88	6	0.20	5	1	6.00	5.00	6.00	9.57
E7	10	11	153.73	4.00	6.90	2	0.20	2	1	5.00	2.00	2.00	16.57
E8	25	17	374.00	3.00	6.00	6	0.24	6	2	4.17	3.00	3.00	6.43
E9	33	30	521.55	3.00	5.00	7	0.21	7	2	4.71	3.50	3.50	20.29
E10	56	67	2420.80	6.00	5.00	6	0.11	5	3	11.20	1.67	2.00	12.29
E11	63	94	2883.17	7.00	4.00	11	0.17	10	3	6.30	3.33	3.67	5.00
E12	117	112	4454.35	5.00	5.00	20	0.17	18	7	6.50	2.57	2.86	17.86
E13	27	23	592.60	5.00	4.00	5	0.19	4	1	6.75	4.00	5.00	-2.43
E14	14	16	372.53	7.00	4.00	2	0.14	1	1	14.00	1.00	2.00	1.14
E15	41	42	1004.43	5.00	5.00	6	0.15	4	1	10.25	4.00	6.00	-2.43
E16	31	22	667.43	4.00	4.00	7	0.23	5	1	6.20	5.00	7.00	16.00
E21	21	17	346.12	4.06	4.88	5	0.24	5	1	4.20	5.00	5.00	20.20
E26	77	49	1328.56	4.00	5.00	14	0.18	11	2	7.00	5.50	7.00	23.80

Figure 6.2: Measurements Results

ASL measures are mutually exclusive.

Another interesting result to observe is the difference in SRES-ASL scores in case of examples E7 and E14. E7 looks simple to read and SRES-ASL scores accordingly a low index of 4. But in case of example E14 that apparently looks simple to read, the SRES-ASL score (SRES-ASL =7.0) is comparatively higher than some other examples (e.g., E5, E13, and E15) which apparently look difficult to read. The obvious reason for the relatively low SRES-ASL score (in case of E5, E13, and E15) is do not considering the comments by the SRES metric. On the other hand, SRES-ASL results high score, in case of E14, due to the presence of several relatively longer print sentences (statements). This implies that we should consider the use of less weights in case of simple print like sentences (statements).

6.3 Statistical Significance and Correlations

We apply statistical data analysis techniques to further evaluate our experimental results. Pearson correlation coefficient is one of the best known and most commonly used statistics to measure correlation between two data sets. It shows the degree and direction of relationship between two variables. In our experiment these two variables are: SRES and the other one is a software metric from a set of selected conventional software metrics. To analyze the relationship between SRES and other software metrics, we use a correlation matrix as shown in the Figure 6.3. Since we have a set of 18 examples, so the input data set used for the two variables of correlation, consists of 18 values, i.e., $N = 18$,.

Regarding the interpretation of correlation coefficient, there are no universally accepted rules. Its value ranges from -1 to 1. Positive sign indicates direct relationship (the value of

	SRES-ASL	SRES-AWL	LoC	PD	PV	TCC	CC/LoC	LoC/m	m/c	WMC	HQS
SRES-ASL	1.0										
SRES-AWL	-0.6	1.0									
LoC	0.3	-0.2	1.0								
PD	0.5	-0.3	0.9	1.0							
PV	0.5	-0.2	0.9	1.0	1.0						
TCC	0.1	-0.1	0.9	0.8	0.8	1.0					
CC/LoC	-0.8	0.5	-0.4	-0.5	-0.4	-0.1	1.0				
LoC/m	0.8	-0.5	0.2	0.3	0.2	-0.1	-0.8	1.0			
m/c	-0.5	0.2	-0.1	-0.3	-0.3	0.1	0.6	-0.5	1.0		
WMC	-0.4	0.0	0.0	-0.2	-0.2	0.2	0.4	-0.3	0.9	1.0	
HQS	-0.5	0.2	0.2	0.0	0.1	0.3	0.3	-0.5	0.3	0.1	1.0

Figure 6.3: Correlation matrix for the software metrics as described in Table 6.1, based on the Pearson product-moment correlation coefficient.

Y increases as X increases) whereas negative sign indicates inverse relationship (the value of Y decreases as X increases). A maximum value of 1 means there exist a perfect linear relationship between variables X and Y, whereas a value of 0 is interpreted as there exist no linear relationship between the two variables. Our interpretation of correlation coefficient, r, is shown in Table 6.2.

Correlation Value	**Interpretation**
0 to $\pm$ 0.29	Very Weak
0.3 to $\pm$ 0.49	Weak
0.5 to $\pm$ 0.69	Strong
0.7 to $\pm$ 1.0	Very Strong

Table 6.2: Correlation Coefficient-Interpretation

An important consideration regarding correlation interpretation is to answer the question: "Is the correlation coefficient significant or not"? In statistics, statistically significant means a result is unlikely to have resulted by chance. So a significant correlation implies that the correlation is not obtained by chance only. To analyze the significance of the results obtained, we investigate on *p-value* calculated using a *paired two-tailed distribution t-test*. We apply a paired version of the t-test because the two software metrics are applied on a same set of data (example programs).

To test significance of a correlation, we apply a threshold (i.e., α level) of $r = \pm 0.468$ at $p = 0.05$. This means , for a correlational statistics, if the p-value is less than the α level, 0.05 in this case, and corresponding value of 'r' is greater than 0.468 or less than -0.468 then the correlational statistics are statistically significant (i.e., the probability is small that the results happened by chance).

LoC	PD	PV	SRES-ASL	SRES-AWL	TCC	CC/LoC	LoC/m	m/c	WMC	HQS
37.00	32.44	971.48	4.49	5.10	6.83	0.20	6.46	3.98	4.50	11.94

Figure 6.4: Arithmetic means of the measurement results shown in Figure 6.2

	SRES-ASL	SRES-AWL	LoC	PD	PV	TCC	CC/LoC	LoC/m	m/c	WMC	HQS
SRES-ASL		0.2088	0.0001	0.0010	0.0023	0.0403	0.0000	0.0113	0.3038	0.9761	0.0008
SRES-AWL	0.1116		0.0001	0.0012	0.0023	0.1183	0.0000	0.0662	0.0217	0.2402	0.0017

Figure 6.5: p-values for SRES and all other software measure as described in the Table 6.1

6.3.1 SRES and LoC

Statistics given by the Figures 6.3 and 6.5 show that there exist a non-significant weak positive, $r < \alpha$, correlation between SRES-ASL and LoC. The weak correlation between the two measures is quite intuitive as the number of lines in a code has nothing to do with the length of a program statements (sentences). The weak positive correlation between SRES-ASL and LoC is further supported by the graph shown in Figure 6.6. There are only 6 (one third of the total) examples (E1, E2, E5, E10, E11, and E16) for which SRES-ASL and LoC show linear relationship. However non-significance of the correlation implies that the result is likely to be observed by chance, and therefore may differ in case of a different sample data.

The statistics for SRES-AWL and LoC also predict non-significant very weak and negative correlation of degree 0.2. Very weak correlation between the two measures is again quite expected as the number of lines in code is not supposed to have any affect on the program's word-length. The graph in Figure 6.7 supports the very week negative correlation between the two measures. However the non-significance of the result implies high probability for the relationship to be happened by chance.

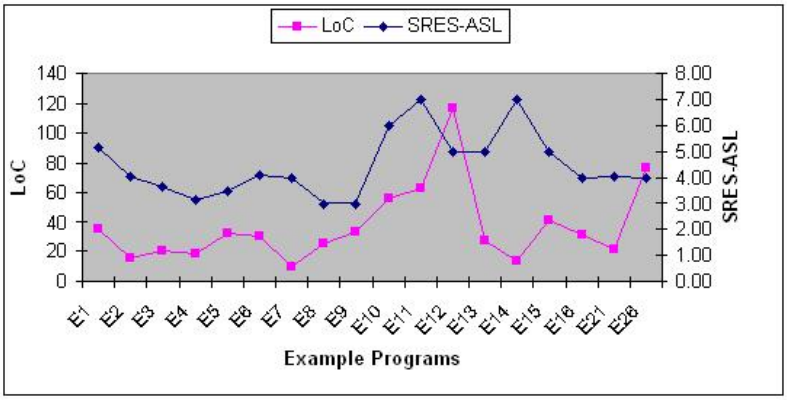

Figure 6.6: SRES-ASL and LoC Correlation

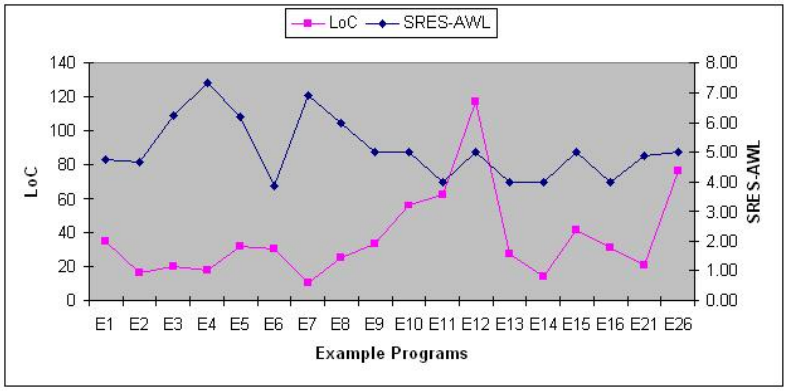

Figure 6.7: SRES-AWL and LoC Correlation

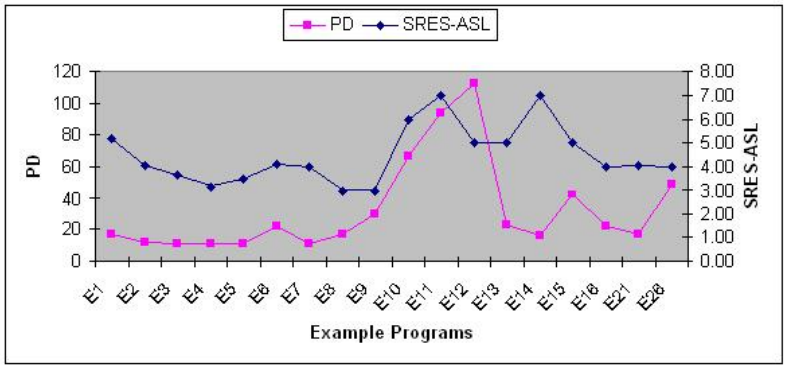

Figure 6.8: SRES-ASL and PD Correlation

6.3.2 SRES and PD Correlation

SRES-ASL and Halstead's measure of difficulty (PD) have a a significantly strong, $r > 0.468$ (critical value) at $p < 0.05$, and positive correlation of degree 0.5. A strong correlation with a low degree of 0.5 between the two metrics seems quite logical because Halstead's difficulty measure is based on the number of operators and operands which may affect the average length of program sentences. measure. Significantly weak positive correlation between SRES-ASL and PD is further supported by the graph shown in Figure 6.8.

The p-value and r-value statistics for SRES-AWL and Program Difficult measure predict non-significant and weak negative correlation with r = -0.3 and p-value less than the α level. The weak inverse relationship between the two variables can be observed by a look at the graph shown in the Figure 6.9. However non-significance of the correlation implies that the result is likely to be observed by chance, and therefore may differ in case of a different sample data.

6.3.3 SRES and PV Correlation

Correlation coefficient 'r' observed for SRES-ASL and Halstead's Program Volume is +0.5 at $p < \alpha$. This implies a strong direct correlation between the two metrics. The p-value

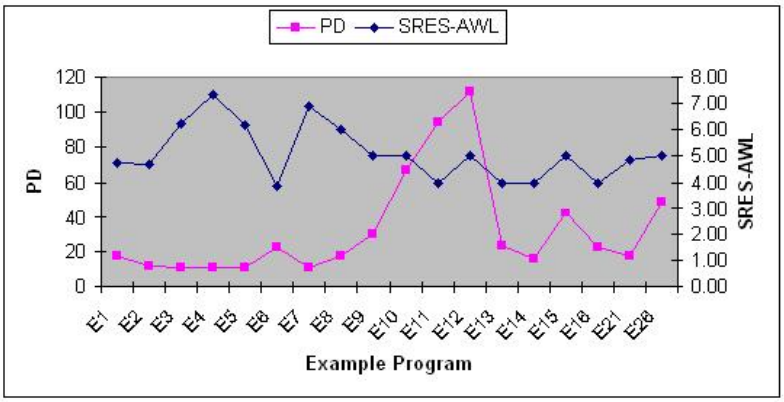

Figure 6.9: SRES-AWL and PD Correlation

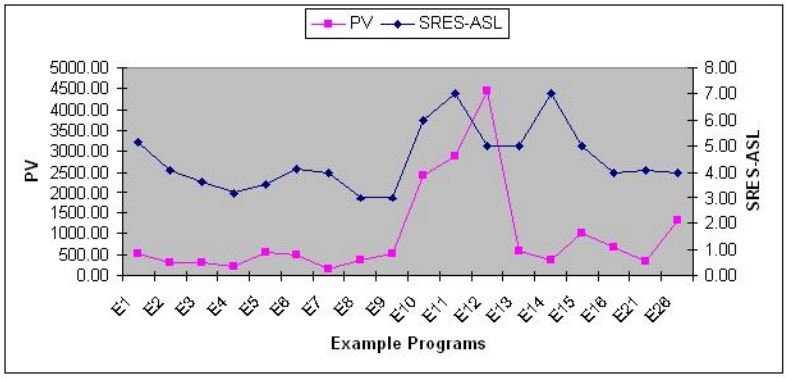

Figure 6.10: SRES-ASL and PV Correlation

for SRES-ASL and PV is below the α level, while 'r' is greater than the critical level, therefore the strong correlation is significant enough as well, with a low probability that it has resulted by chance. Since PV metrics is based on the number of operators and operands in a program, therefore the strong correlation with the program's average sentence length metric seems to be logical in theory as well. The graph, shown in Figure 6.10 is also in agreement to the this prediction.

The correlation coefficient for SRES-AWL and Program Volume measure is -0.2 with p-value less than the α level. The statistics given in Figures 6.3 and 6.5 implies that the SRES-AWL and Program Volume measures have a non-significant and week negative correlation. The non-significance of the result implies high probability of these results to be happened by chance, and the statistics might be different in case of a different data set. The weak relationship between the two variables can also be observed by a look at the graph shown in Figure 6.11.

6.3.4 SRES and TCC Correlation

Correlation coefficient 'r' for Average Sentence Length and Total Cyclomatic Complexity measures is 0.1 with corresponding p-value of 0.04. These statistics given in Figures 6.3 and

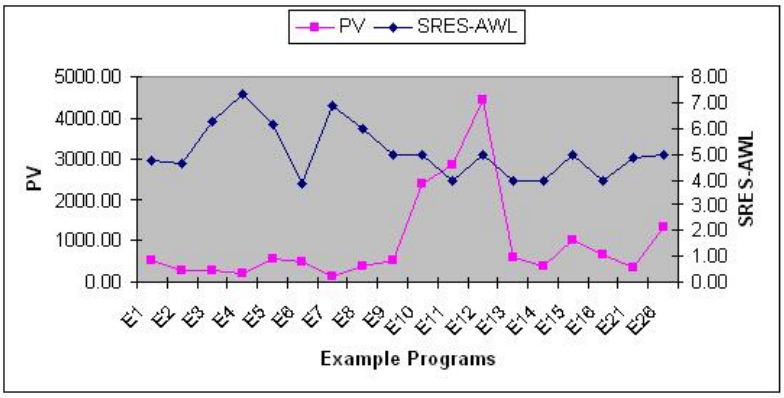

Figure 6.11: SRES-AWL and PV Correlation

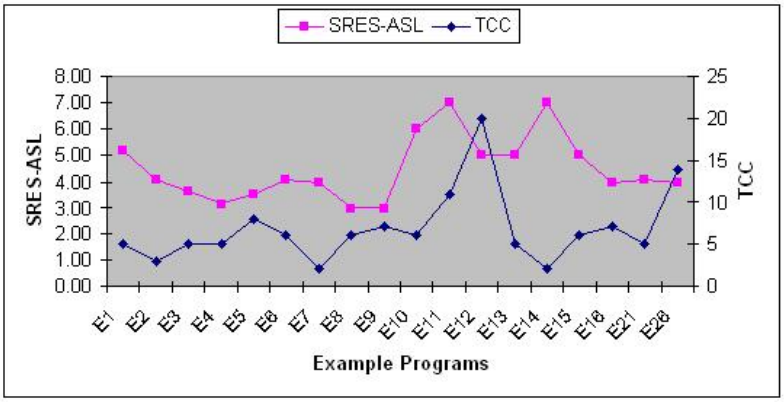

Figure 6.12: SRES-ASL and TCC Correlation

6.5 implies a non-significant and very weak positive correlation. TCC metric is based on the number of linearly independent paths through the source code, and these control paths have nothing to do with the the length of program sentences (statements). Therefore the prediction of very weak correlation between the two metrics is intuitive to much extent.

Statistics for TCC and Average Word Length measure of SRES predict a non-significant weak negative correlation of degree 0.1, as shown in Figure 6.3. The correlation is non-significant because the p-value is greater than the α level as well as the value of 'r' is also below the critical value of 0.468 . Non-significance of the correlation implies that the result is likely to be observed by chance, that means it is not a reliable prediction. The weak relationship between the two variables can also be observed by a look at the graph shown in the Figure 6.11.

6.3.5 SRES and CC/LoC Correlation

Correlation coefficient for SRES-ASL and "Average Cyclomatic Complexity per LoC" is -0.8 with the p-value less than the α level. These statistics given in Figures 6.3 and 6.5 predict a significantly very strong correlation between the tow metrics in opposite direction,

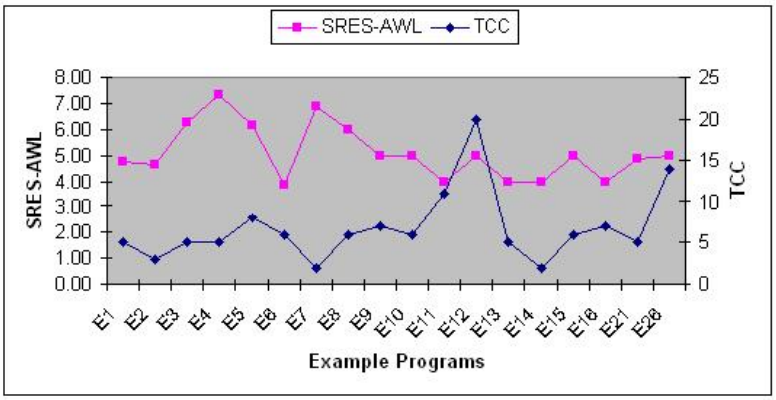

Figure 6.13: SRES-AWL and TCC Correlation

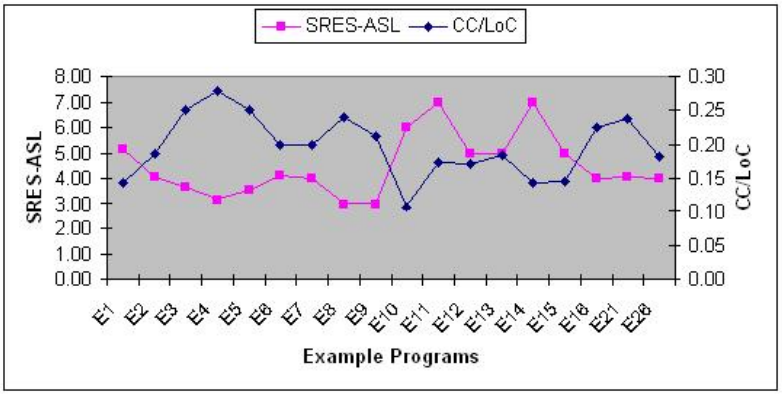

Figure 6.14: SRES-ASL and CC/LoC Correlation

i.e., increase in the value of one measure decreases the value of the other. This very strong inverse linear relationship between the two variables can be observed by having a look at the graph shown in the Figure 6.14. Significance of the statistics implies the results are likely to be reliable with small probability of the relationship to be observed by chance.

Whereas the correlation coefficient for the SRES-AWL and "Average Cyclomatic Complexity per LoC" measure is 0.5 with a corresponding p-value less than the α level. These statistics given in Figures 6.3 and 6.5 imply a significantly strong positive correlation, as supported by the graph of the Figure 6.15.

6.3.6 SRES and LoC/m Correlation

SRES-ASL and LoC/m metrics have a a significantly very strong, $r > 0.468(criticalvalue)$ at $p < 0.05$, positive correlation of degree 0.8. Though the very strong correlation is not very intuitive in theory, however statistically it has been resulted significant to show its reliability. This perfect very strong linear relationship between the two measures is clearly visible in the graph, shown in Figure 6.16.

The correlation coefficient observed for SRES-AWL and "LoC per Method" (LoC/m) is

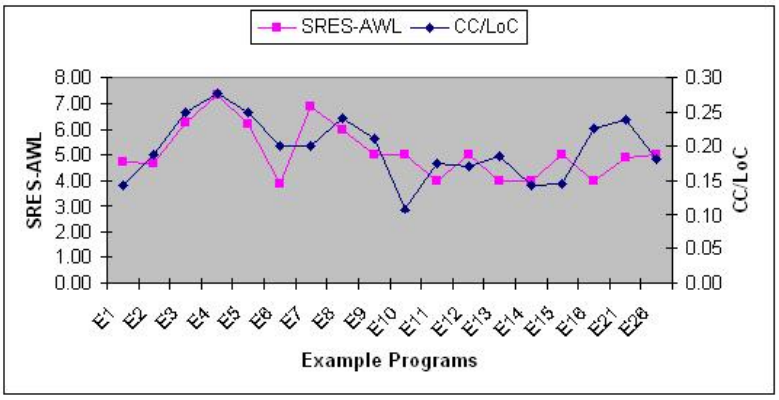

Figure 6.15: SRES-AWL and CC/LoC Correlation

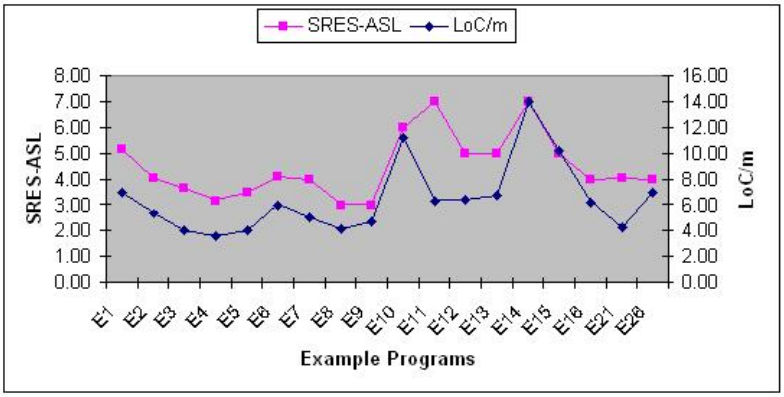

Figure 6.16: SRES-ASL and LoC/m Correlation

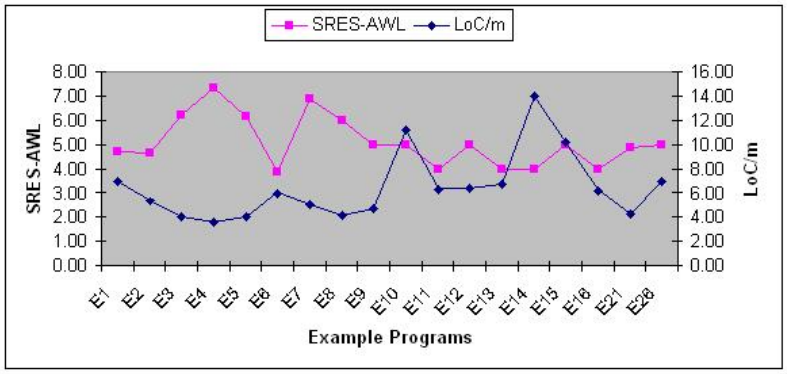

Figure 6.17: SRES-AWL and LoC/m Correlation

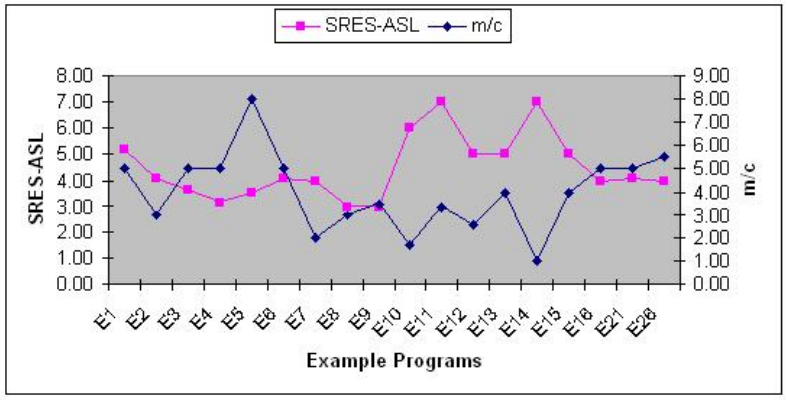

Figure 6.18: SRES-ASL and m/c Correlation

-0.5 as shown in the correlation matrix of Figure 6.3. However The p-value for two metrics is greater than the α level, resulting in non-significance of the correlation. These statistics imply that there exist a non-significantly strong negative correlation between the measures, as shown in the graph of Figure 6.17.

6.3.7 SRES and m/c Correlation

The correlation coefficient for SRES-ASL and 'm/c' metrics found is -0.5, with a corresponding p-value of 0.3. These statistics leads to the interpretation of non-significantly strong inverse correlation, since $p > \alpha$. The graph plotted for SRES-ASL and m/c metrics, Figure 6.18, also supports a fairly strong correlation between m/c and SRES measure of Average Sentence Length (SRES-ASL), in opposite direction. However statistically the correlation is not significant and may lead to different interpretation of correlation when tested with different data set.

Whereas the correlation coefficient 'r' for SRES-AWL and "Methods per Class" (m/c) measure is 0.2, with a corresponding p-value of 0.02. These statistics predict a non-significantly very weak correlation between the two measures, since the value of 'r' is less than the critical value of 0.468. The graph in Figure 6.19 also supports a very weak direct

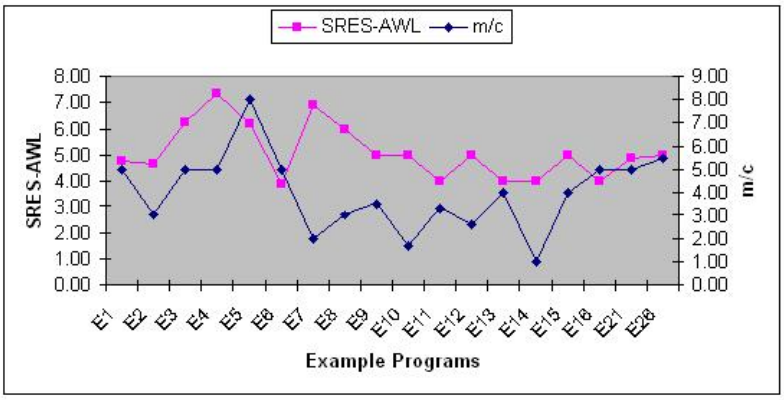

Figure 6.19: SRES-AWL and m/c Correlation

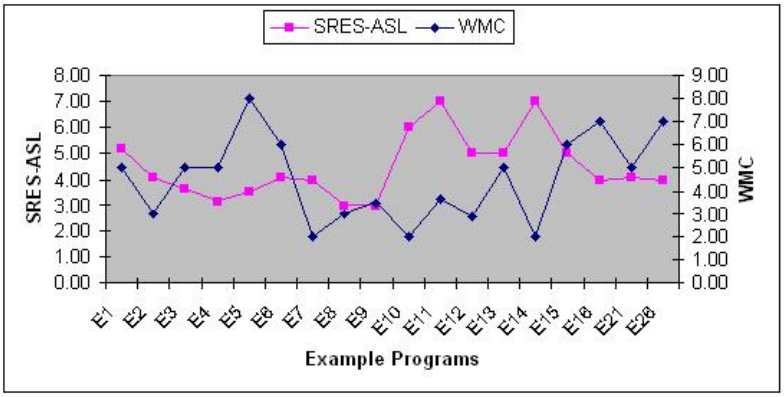

Figure 6.20: SRES-ASL and WMC Correlation

correlation between the two measures. The non-significance of the measures imply high probability of statistics to be resulted by chance.

6.3.8 SRES and WMC Correlation

Correlation coefficient for SRES-ASL and WMC measure is -0.4, with a p-value of 0.98, much higher than the α level. These statistics lead to interpretation of non-significant and weak negative correlation between the two metrics. The graph of ASL and WMC measure for selected example programs is shown in the Figure 6.20.

The correlation coefficient 'r' for SRES-AWL and WMC metrics is 0.0 (much lower than the critical value of 0.468), with a corresponding p-value greater than the α level of 0.05. This predicts that the correlation between the two metrics is non-significant and very weak as well. Correlation coefficient of 0 does not mean that the two variables are unrelated or independent of each other, rather it implies that there exist no linear relationship at all between them. Further the non-significance of the correlation implies that the results might have occurred by chance, and the two metrics may have different correlation coefficient for some other data set. This very weak or almost no linear relationship between the two

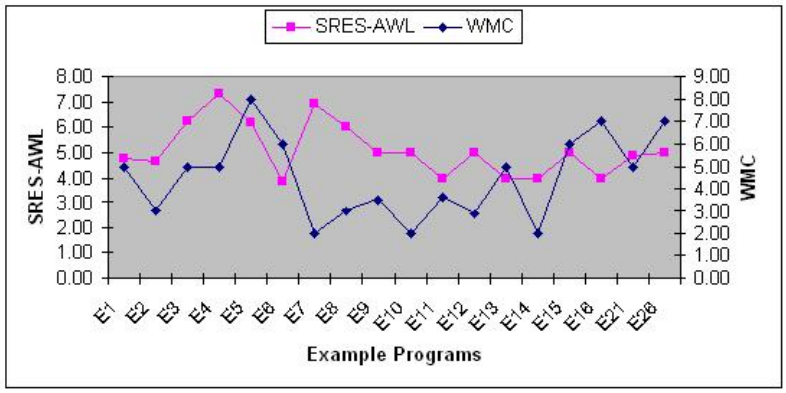

Figure 6.21: SRES-AWL and WMC Correlation

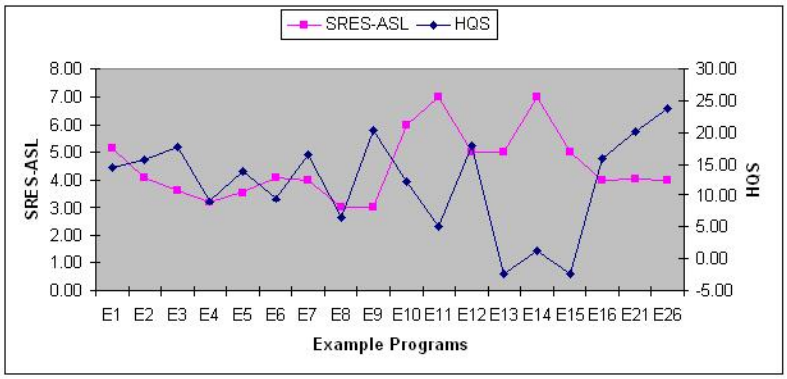

Figure 6.22: SRES-ASL and HQS Correlation

variables can be noted by a look at the graph shown in the Figure 6.21

6.3.9 SRES and HQS Correlation

As shown in Figure 6.3, the correlation coefficient for SRES-ASL and HQS measures is -0.5, with a corresponding p-value less than the α level of 0.05. Both 'r' and 'p' values satisfy the threshold for the significance of the correlation. However the negative sign with 'r' indicates inverse correlation. Inverse direction of the correlation implies that the SRES-ASL measure of example's readability is not in agreement to the quality score perceived by a group of human reviewers. This opposite direction of linear relationship between two measures is quite obvious in the graph shown in Figure 6.22

Whereas the correlation coefficient for SRES-AWL and HQS measure is 0.2, with a corresponding p-value less than the α level. These statistics predict a non-significant very weak correlation between the two measures. The weak correlation between the two metrics is supported by the Figure 6.23. The weak correlation between two measure is not as expected to be, and it shows that the SRES based readability index is not in a linear relationship to the quality score assigned to example programs by the human reviewers in case of HQS.

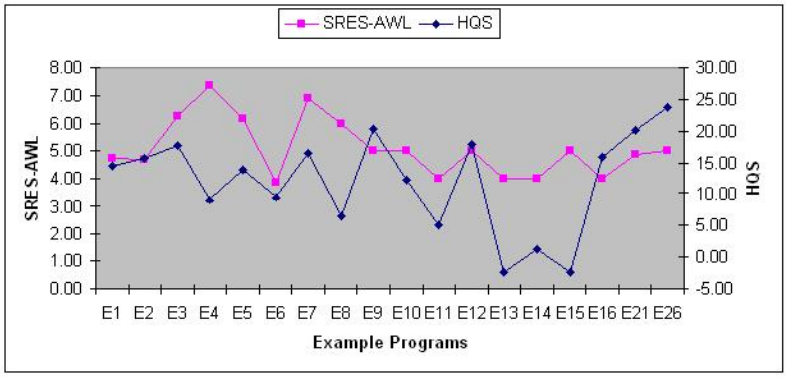

Figure 6.23: SRES-AWL and HQS Correlation

6.3.10 SRES and Buse's Metric for Software Readability

We have described Buse and Weimer's Metric for Software Readability [11] in Section 1.3. It would be interesting to see how these two metrics of software readability correlates. To measure readability of a code, both metrics count syntactic elements of the code which are easy to measure using static code analysis techniques. Buse's metric is based on a fairly large code readability survey (12000 human judgments) to determine which factors are the prime determinant of readability. Whereas SRES metric is based on the principles of "Flesch's Readability Formula" described in the Section 4.3. The code features that these two metrics use to score readability index are partially overlapping. SRES has two parts, "SRES-Average Sentence Length" and "SRES-Average Word Length". In Buse's metric, SRES-ASL appears with the name of "Average Line Length" and is second most contributing code features to predict the readability index. "SRES-Average Word Length" measure somewhat corresponds to the "Average Identifier Length" feature, which is observed by Buse et al. to have almost no contribution in the measure of readability. However Buse and Weimer's findings are based on human annotation subjected to personal preferences, and should not be followed as a comprehensive model of readability. But it would be interesting to further investigate the true role of identifier-length in the measures of readability.

In order to statistically analyze the correlation between SRES ans Buse's metrics, we follow the same approach as in case of other software metrics described in the earlier sections. To get a score for Buse's metric we use an on-line tool available at `http://www.arrestedcomputing.com/readability/readapplet.html`. In support of Buse's approach of using code snippets instead of large code segments or multiple java classes, this tool expects java methods as input. Therefore to meet the tool requirements, we shortlisted our input examples data set of Figure 6.1, to a new subset that does not include examples with more than 1 java classes. The new subset contains 9 examples: E1-E3, E5-E7, and E13-E15. Figure 6.24 shows metrics scores and Figure 6.25 shows correlation coefficient and p-values for SRES and Buse's metrics using new examples data set. Since our new data set now have size N=9, therefore we change the threshold for p-value to 0.01. Accordingly the threshold for 'r' is now 0.798, to test if the correlation statistics are significant or not at p ¡ 0.01.

Statistics given by Figure 6.25 show that there exist non-significant (value of 'r' is much less than the critical value) very weak direct correlation between SRES-ASL and Buse's metrics. The weak linear relationship between the two metrics is supported by a graph,

Example	SRES-ASL	SRES-AWL	Buse
E1	5.17	4.74	0.65
E2	4.08	4.67	1.89
E3	3.64	6.25	0.02
E5	3.52	6.17	0.00
E6	4.10	3.88	0.18
E7	4.00	6.90	0.82
E13	5.00	4.00	0.00
E14	7.00	4.00	0.99
E15	5.00	5.00	0.06

Figure 6.24: Results of SRES and Buse's Readability Metrics

	r	p-value
SRES-ASL and Buse Metric	0.20	0.00
SRES-AWL and Buse Metric	-0.12	0.00

Figure 6.25: Correlation Coefficient and p-value for SRES and Buse's Readability Metrics

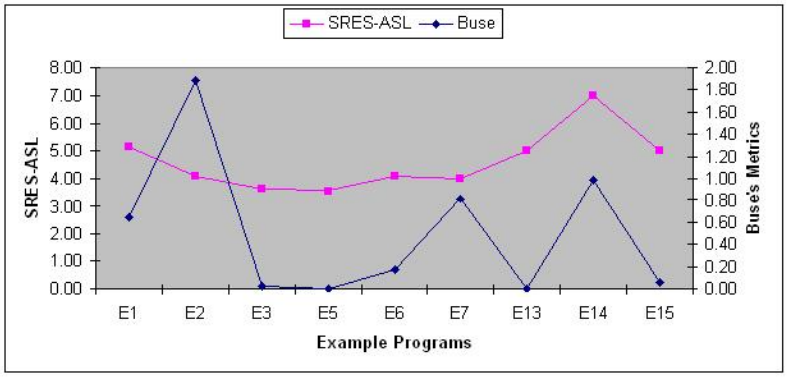

Figure 6.26: SRES-ASL and Buse's Readability Metrics Correlation

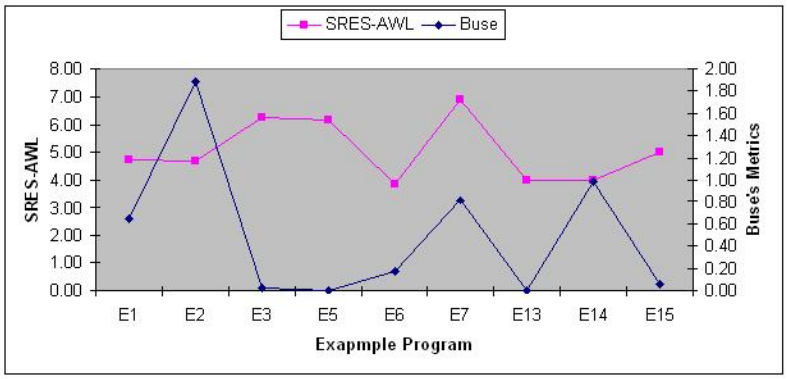

Figure 6.27: SRES-AWL and Buse's Readability Metrics Correlation

shown in the Figure 6.26. The statistics observed for SRES-AWL and Buse's metrics predict non-significantly very weak ($r = -0.12$) negative correlation, as supported by a graph of the Figure 6.27. Though we have not empirically investigated the reasons for such weak correlations but we speculate that it is likely due to the different treatments for "comments" and "average identifier length" code features, by the two metrics. Further since both the correlations statistics are non significant so there is a high probability that it might have resulted by chance.

6.4 Validation of Halstead's Metrics

We have defined our own counting strategy for Halstead's metrics and developed a measurement tool based on this counting strategy. Therefore its important to validate our results of Halstead's metrics in comparison to the Halstead's results captured using some other tool. For this purpose use the same example data set, as we used in the Section 6.3.10, along with the same thresholds for the critical values of 'r'. We use the measurement JHawk (`http://www.virtualmachinery.com/jhawkprod.htm`) to measure Halstead's *Program Effort (PE)* metric. Figure 6.28 shows the results for PE metric captured by our custom built SRES measurement tool and JHawk.

Examples	PE by SRES	PE byJHawk
E1	9010	588.42
E2	3497.124	102.75
E3	3234	226.63
E5	6138	93.6
E6	10516	1254
E7	1691.03	182.36
E13	13629.8	1286.6
E14	5960.48	1410.7
E15	42186.06	6709.55

Figure 6.28: Halstead Measure of Program Effort by JHawk and SRES Measurement Tool

Using the data set given in the Figure 6.28 for the two measurement tools, Pearson's correlation coefficient 'r' value computed is 0.98, and the p-value is less than the α level of 0.01. This predict a significantly very strong positive correlation between the two measurement tools. Such very strong and significant correlation of our Halstead's metrics implementation to another commercially used measurement tool supports the validity of our results in case of Halstead's metric.

Chapter 7

Discussion and Conclusions

7.1 Discussion

Learning "How to Program" is a two-fold process where a student learns language syntax as well as underlying programming methodology such as object-oriented, procedural and non-procedural. The most difficult part in learning object oriented programming is learning object-oriented design skills that require students to truly understand problems and design a good solution. This can be simplified by teaching programming with the use of carefully designed example programs. Apart from "bad" examples, there are several other reasons of student problems in learning object oriented programming [39, 45, 40, 33, 61, 42, 37, 68]. These problems include conceptual myths and misconception about object orientation and ineffective pedagogy. However, this study focus only on the role of example programs and their understanding by novices.

Learning "how to program" is a cognitive process, commonly referred as program comprehension. Being a cognitive process, program comprehension varies person to person, based on the limits of individual's experience, interest and cognitive capabilities. Under such circumstances, it is very difficult to find a standard measure of comprehension or ease of understanding. However, we may have well calculated and justified measures based on a wide scale empirical studies and well known cognitive theories of comprehension.

Understanding the problems and complexities confronted during program development has remained a topic of great interest for researchers during the last few decades. Earlier attempts, such as those made by McCabe [50] and Halstead [29], primarily focused on the static physical structure of the program rather than cognitive issues. Complexity is an intrinsic feature associated with the problem domain and can not be avoided completely. Whereas readability is a more controlled attribute that can be improved independent of the problem complexity or problem domain [11].

In this study, we focus only on the source code for example programs, any other auxiliary text, description or tools are not explored. All the example programs analyzed and evaluated have been selected from popular java text books, see Figure 6.1, and the text books also include explanatory notes or description about the example programs used as learning tools. Such description or explanatory text helps reader and learners to understand the quoted examples, with special emphasis on code excerpts rated as complex or difficult to understand. We qualify such notes or textual description as a supplementary part of an example program. Readability and quality of these supplementary text or programming theory provided in text books or as lecture notes, greatly impacts students' understanding. Therefore, we

acknowledge that one should measure and evaluate the readability and quality of lecture notes or textual description about associated examples' source code.

We acknowledge that *comments* in a program have a fair impact on readability and quality. Although We regarded comments as an important factor of program comprehension, Section 2.3.2, however, the *Software Reading Ease Score*, a measure of readability, implemented and evaluated in this study does not count on the impact of comments. Comments provide additional support to understand complex program elements, but at the same time, too many comments in the code may obscure actual lines of code, making the program bulky or clumsy to read. In itself it is an interesting research to define a suitable criterium to measure the effect of comments on program quality. In future we aim to experiment about how good or bad the comments might be, can we have some threshold for right use of comments per line of code or any other unit of code.

SRES metrics of program readability does not care about program indentation and format style. A badly formated and poorly indented program creates problems in chunking and tracing, making the code difficult to read and understand. SRES is inspired from FRES [25], a readability metric for ordinary text. However, the indentation factor is more contributive in a program text than in an ordinary text. Indentation remains mostly standard in general text with just few variations in paragraph style. However, in case of a computer program, level of indentation varies greatly depending on author or organizational preferences. Though the example programs, quoted in text books or used academia, are well structured, having proper indentation and formating, still it remains an important factor to determine the readability of example programs.

We evaluated 18 example programs selected from 10 different well known object oriented java programming textbooks. An important finding is that none of those example programs satisfies all the 10 selected software metrics, Table 6.1. There are 8 out of 18 (44%) example programs that fail to satisfy either *ASL* or *AWL* measure of Software Readability Ease Score. This implies that not all the example programs quoted in commonly used java text books are perfect in their readability.

Apart from readability, there are several other quality attributes, described in section 3.2. However we need to identify some measurable criteria to have a quantitative measure for each of those quality attributes. For example, simplicity, communication, consistency, and reflectivity are few of the proposed example quality attributes, for which we do not have any empirically validated or well known metrics. It requires more extensive and in detail empirical study to establish sound measures for these quality attributes, that we aim as a future work.

SRES is yet on initial stages. There is no sound, empirically proved and tested threshold to categorize example programs as good or bad in readability. In order to have a standard threshold, we need a large scale study involving a more extensive number of example programs reviewed by a good number of researchers with readability perspective. Further SRES works on the pattern of Flesch' readability formula designed for ordinary text, while format and organization of program code varies from ordinary text. The SRES readability formula does not consider the impact of factors like indentation, code style, naming convention and comments support.

7.2 Conclusions

Example programs act as a valuable learning tool and play a significant role in programming education. Examples are quoted in text books, lecture notes and tutorial sessions to provide students with a practical demonstration. Unfortunately, all example programs

are not equally good or useful in learning. Poorly designed example program may badly impact student's understanding. We need to carefully evaluate and review the quality of the example programs based on the principles and standards of learning. To distinguish good and bad examples, a set of desired quality attributes is proposed as a key objective of this study. Among the proposed attributes, *readability* is rated as a prime attribute assuming the hypothesis: *you can't understand if you can't read*. *Software Reading Ease Score*, a readability metrics implemented as a part of this study is inspired from Flesch's readability formula. SRES is based on average sentence length and number of character per word. It can be extended by adding some weights based on the support from comments, beacons, indentation, coding standards, and other pro readability factors. Empirical results show that in comparison to other software metrics, SRES scores differently on a set of example programs evaluated. SRES is yet in evolution phase and at present does not consider comments, white-spaces, and indentation style. By considering these static code features, there are bright chances to improve the SRES based predictions about program readability.

7.3 Future work

We aim to further investigate usability and applications of *SRES*, the program readability metric implemented in this study. We plan a large scale study involving educators, students, and researcher in the domain, to evaluate the results captured by current SRES metrics and actual readability perceived by human subjects. Since readability depends on a variety of factors both internal and external to the program, therefore, we plan to find out appropriate weights for all those factors that impact program readability. At the same time, our preferences are to maintain the ease and simplicity of the readability formula.

At present, the SRES measurement tool developed as a part of this study implements SRES and Halstead's metrics. We plan to extend it by adding more software metrics and options to customize the measures. In addition to readability, we also plan to research on measurable criteria of other program quality attributes discussed in Section 3.2.

Chapter 8

Acknowledgments

At first, I would like to record my gratitude to Jürgen Börstler for his supervision, counseling, and guidance throughout this study. I am much indebted to him for showing confidence, encouraging and helping me achieve the goals and objectives of this study. He always supported me to understand the things by sharing his worthy knowledge and experience. Many thanks go to Per Lindström, who is always kind and up with valuable advice throughout my studies here.

On the personal level, I am thankful to my loving parents for all their sincere prayers, valuable support and all the good things they blessed me. Particularly I would like to dedicate this work to my beloved mother. I lost her while this study was in progress, but she will always remain alive in my heart.

References

[1] The free on-line dictionary of computing. Sep 2009.

[2] M.J. Abram and W.D. Dowling. How Readable are Parenting Books? *Family Coordinator*, pages 365–368, 1979.

[3] C. Aschwanden and M. Crosby. Code Scanning Patterns in Program Comprehension. In *Symposium on Skilled Human-Intelligent Agent Performance. Mesurement, Application and Symbiosis. Hawaii International Conference on Systems Science*, 2006.

[4] VR Basili. Qualitative Software Complexity Models: A Summary. *Tutorial on Models and Methods for Software Management and Engineering. IEEE Computer Society Press, Los Alamitos, Calif*, 1980.

[5] J. Börstler, M.E. Caspersen, and M. Nordström. Beauty and the Beast–Toward a Measurement Framework for Example Program Quality. (UMINF-07.23), 2007.

[6] J. Börstler, M. Nordström, L.K. Westin, J.E. Moström, H.B. Christensen, and J. Bennedsen. An Evaluation Instrument for Object-Oriented Example Programs for Novices. (UMINF-08.09), 2008.

[7] Jürgen Börstler, Henrik B. Christensen, Jens Bennedsen, Marie Nordström, Lena Kallin Westin, Jan Erik Moström, and Michael E. Caspersen. Evaluating oo example programs for cs1. In *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*, pages 47–52. ACM, 2008.

[8] Jürgen Börstler, Mark S Hall, Marie Nordström, James H Paterson, Kate Sanders, Carsten Schulte, and Lynda Thomas. An evaluation of object oriented example programs in introductory programming textbooks. *inroads*, 41, 2009.

[9] R. Brooks. Towards a Theory of the Comprehension of Computer Programs. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983.

[10] J.M. Burkhardt, F. Détienne, and S. Wiedenbeck. Mental Representations Constructed by Experts and Novices in Object-Oriented Program Comprehension. *Arxiv Preprint CS/0612018*, 2006.

[11] Raymond P.L. Buse and Westley R. Weimer. A Metric for Software Readability. In *ISSTA '08: Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 121–130, New York, NY, USA, 2008. ACM.

[12] L.R. Campbell and G.R. Klare. Measuring the Readability of High School Newspapers. 1967.

[13] SN Cant, B. Henderson-Sellers, and D.R. Jeffery. Application of Cognitive Complexity Metrics to Object-Oriented Programs. *Journal of Object Oriented Programming*, 7:52–52, 1994.

[14] SN Cant, DR Jeffery, and B. Henderson-Sellers. A Conceptual Model of Cognitive Complexity of Elements of the Programming Process. *Information and Software Technology*, 37(7):351–362, 1995.

[15] M.T.H. Chi and M. Bassok. Learning from examples via self-explanations. *Knowing, learning, and instruction: Essays in honor of Robert Glaser*, pages 251–282, 1989.

[16] SR Chidamber, CF Kemerer, and C. MIT. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[17] E. Dale and J.S. Chall. A Formula for Predicting Readability. *Educational Research Bulletin*, pages 11–28, 1948.

[18] L.E. Deimel, J.F. Naveda, and Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst. Reading Computer Programs: Instructor's Guide to Exercises., 1990.

[19] D.P. Delorey, C.D. Knutson, and M. Davies. Mining Programming Language Vocabularies from Source Code. In *21st Annual Psychology of Programming Interest Group Conference, 2009. PPIG 2009*, 2009.

[20] F. Détienne and F. Bott. *Software Design–Cognitive Aspects*. Springer Verlag, 2001.

[21] W.H. DuBay. The Principles of Readability. *Impact Information*, pages 1–76, 2004.

[22] J.L. Elshoff. An Investigation into the Effects of the Counting Method Used on Software Science Measurements. *ACM Sigplan Notices*, 13(2):30–45, 1978.

[23] J.L. Elshoff and M. Marcotty. Improving Computer Program Readability to Aid Modification. *Communications of the ACM*, 25(8):512–521, 1982.

[24] V. Fix, S. Wiedenbeck, and J. Scholtz. Mental Representations of Programs by Novices and Experts. In *Proceedings of the INTERACT'93 and CHI'93 Conference on Human Factors in Computing Systems*, pages 74–79. ACM New York, NY, USA, 1993.

[25] R. Flesch. A New Readability Yardstick. *The Journal of Applied Psychology*, 32(3):221, 1948.

[26] E. Fry. A readability formula that saves time. *Journal of reading*, pages 513–578, 1968.

[27] R. Gunning. *The Technique of Clear Writing*. McGraw-Hill, 1968.

[28] MH Halstead. Toward a Theoretical Basis for Estimating Programming Effort. In *Proceedings of the 1975 Annual Conference*, pages 222–224. ACM New York, NY, USA, 1975.

[29] M.H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc. New York, NY, USA, 1977.

[30] N.J. Haneef. Software Documentation and Readability: A Proposed Process Improvement. *ACM SIGSOFT Software Engineering Notes*, 23(3):75–77, 1998.

[31] G. Hargis, M. Carey, A.K. Hernandez, P. Hughes, D. Longo, S. Rouiller, and E. Wilde. *Developing Quality Technical Information: A Handbook for Writers and Editors*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2004.

[32] S. Henry and D. Kafura. Software Structure Metrics Based on Information Flow. *Transactions on Software Engineering*, pages 510–518, 1981.

[33] Simon Holland, Robert Griffiths, and Mark Woodman. Avoiding object misconceptions. In *SIGCSE '97: Proceedings of the Twenty-Eighth SIGCSE Technical Symposium on Computer Science Education*, pages 131–134, New York, NY, USA, 1997. ACM.

[34] R. Jeffries. A Comparison of the Debugging Behaviour of Expert and Novice Programmers. In *Proceedings of AERA Annual Meeting*, 1982.

[35] C. Kaner and W.P. Bond. Software Engineering Metrics: What Do They Measure and How Do We Know? *methodology*, 8:6.

[36] Joseph P. Kearney, Robert L. Sedlmeyer, William B. Thompson, Michael A. Gray, and Michael A. Adler. Software Complexity Measurement. *Commun. ACM*, 29(11):1044–1050, 1986.

[37] Päivi Kinnunen and Lauri Malmi. Why students drop out cs1 course? In *ICER '06: Proceedings of the Second International Workshop on Computing Education Research*, pages 97–108, New York, NY, USA, 2006. ACM.

[38] G.R. Klare. The measurement of readability: useful information for communicators. *ACM Journal of Computer Documentation (JCD)*, 24(3):107–121, 2000.

[39] M. Koelling and J. Rosenberg. Guidelines for Teaching Object Orientation with Java. *ACM SIGCSE Bulletin*, 33(3):33–36, 2001.

[40] M. Kölling. The Problem of Teaching Object-Oriented Programming. *Journal of Object Oriented Programming*, 11(8), 1999.

[41] D.S. Kushwaha and AK Misra. Improved Cognitive Information Complexity Measure: A Metric that Establishes Program Comprehension Effort. *ACM SIGSOFT Software Engineering Notes*, 31(5):1–7, 2006.

[42] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. A study of the difficulties of novice programmers. In *ITiCSE '05: Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, pages 14–18, New York, NY, USA, 2005. ACM.

[43] M. Lanza, R. Marinescu, and S. Ducasse. *Object-Oriented Metrics in Practice*. Springer-Verlag New York, Inc. Secaucus, NJ, USA, 2005.

[44] S. Letovsky. Cognitive Processes in Program Comprehension. In *Empirical Studies of Programmers*, pages 58–79. Intellect Books, 1986.

[45] John Lewis. Myths about object-orientation and its pedagogy. In *SIGCSE '00: Proceedings of the Thirty-First SIGCSE Technical Symposium on Computer Science Education*, pages 245–249, New York, NY, USA, 2000. ACM.

[46] D.C. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental Models and Software Maintenance. In *Empirical Studies of Programmers: First Workshop*, pages 80–98. Ablex Publishing Corp. Norwood, NJ, USA, 1986.

[47] B.A. Lively and S.L. Pressey. A Method for Measuring the'Vocabulary Burden'of Textbooks. *Educational Administration and Supervision*, 9(389-398):73, 1923.

[48] Katherine Malan and Ken Halland. Examples that Can Do Harm in Learning Programming. In *OOPSLA '04: Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 83–87, New York, NY, USA, 2004. ACM.

[49] R.E. Mayer. The Psychology of How Novices Learn Computer Programming. *ACM Computing Surveys (CSUR)*, 13(1):121–141, 1981.

[50] T.J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[51] G.H. McLaughlin. SMOG Grading: A New Readability Formula. *Journal of Reading*, 12(8):639–646, 1969.

[52] DM Miller, RS Maness, JW Howatt, and WH Shaw. A Software Science Counting Strategy for the Full Ada Language. *ACM SIGPLAN Notices*, 22(5):32–41, 1987.

[53] S. Misra. An Object Oriented Complexity Metric Based on Cognitive Weights. In *6th IEEE International Conference on Cognitive Informatics*, pages 134–139, 2007.

[54] R. Mosemann and S. Wiedenbeck. Navigation and Comprehension of Programs by Novice Programmers. In *9th International Workshop on Program Comprehension, 2001. IWPC 2001. Proceedings.*, pages 79–88, 2001.

[55] M. Nordström. He[d]uristics - Heuristics for Designing Object Oriented Examples for Novices, March 2009.

[56] T. Parr. The Definitive ANTLR Reference. *The Pragmatic Programmers (May 2007)*.

[57] N. Pennington. Comprehension Strategies in Programming. In *Empirical Studies of Programmers: Second Workshop*, pages 100–113, 1987.

[58] D.R. Raymond. Reading Source Code. In *Proceedings of the 1991 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 3–16. IBM Press, 1991.

[59] S. Rugaber. Program Comprehension. *Encyclopedia of Computer Science and Technology*, 35(20):341–368, 1995.

[60] N.F. Salt. Defining Software Science Counting Strategies. *ACM Sigplan Notices*, 17(3):58–67, 1982.

[61] Kate Sanders and Lynda Thomas. Checklists for grading object-oriented cs1 programs: Concepts and misconceptions. *SIGCSE Bull.*, 39(3):166–170, 2007.

[62] B. Shneiderman and R. Mayer. Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *International Journal of Parallel Programming*, 8(3):219–238, 1979.

[63] E. Soloway, B. Adelson, and K. Ehrlich. Knowledge and Processes in the Comprehension of Computer Programs. *The nature of expertise*, pages 129–152, 1988.

[64] E. Soloway and K. Ehrlich. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, 10(5):595–609, 1984.

[65] G. Spache. A New Readability Formula for Primary-Grade Reading Materials. *The Elementary School Journal*, pages 410–413, 1953.

[66] CS2008 Interim Review TaskForce. Computer Science Curriculum 2008: An Interim Revision of CS 2001, December 2008.

[67] W.L. Taylor. Cloze Procedure: A New Tool for Measuring Readability. *Journalism quarterly*, 30(4):415–433, 1953.

[68] Phil Ventura and Bina Ramamurthy. Wanted: Cs1 students. no experience required. In *SIGCSE '04: Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, pages 240–244, New York, NY, USA, 2004. ACM.

[69] L. Verhoeven and C. Perfetti. Advances in Text Comprehension: Model, Process and Development. *Applied Cognitive Psychology*, 22(3), 2008.

[70] A. Von Mayrhauser and A.M. Vans. *Program Understanding : A Survey.* Colorado State Univ., 1994.

Appendix A

User's Guide

SRES measurement tool is a stand alone GUI based java application. Current version of the tool is delivered as a *jar* file "Pogje.jar" that can be executed using *java -jar* command. Figure A.1 shows the graphical user interface of the application.

After executing the application, please follow the below steps to proceed:

1. Click on "Browse" button to select either a single java source file or a directory collection of java source files.
2. Click on "SRES" buttion if you want to compute SRES measures for the selected source files.
3. Click on "Halstead" buttion if you want to compute Halstead's measures.
4. Measurement results will be displayed in the "Results" text area, and "Save" button will be enabled.
5. Click on "Save" button and specify a target file name and destination, if you want to record the computed results in a file. If you do not save, results will be overwritten or just discarded if you quit the application.
6. Application can be closed using either windows standard close button in the top right corner or using the key combination "Alt+F4".

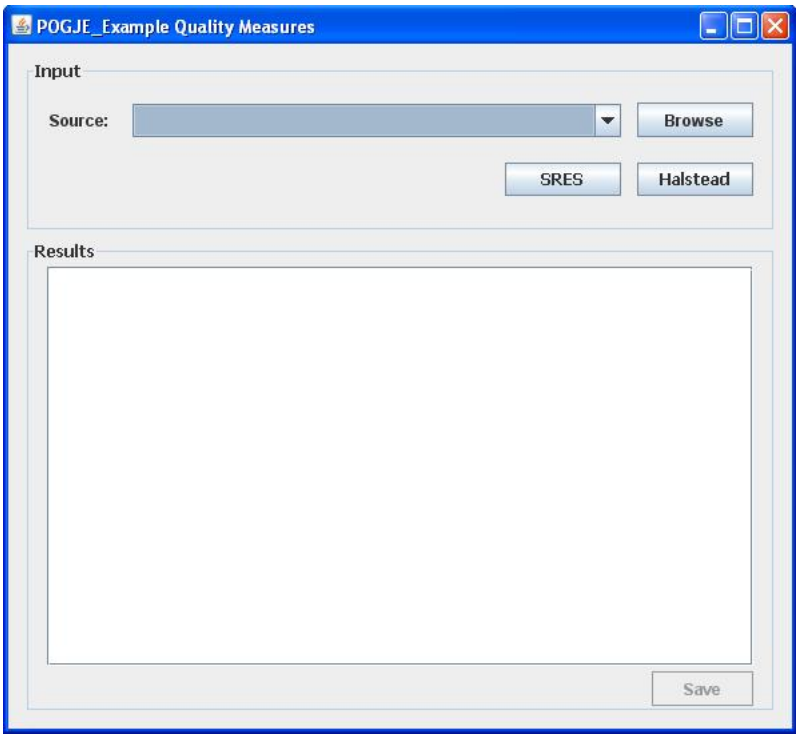

Figure A.1: SRES - User Interface