Factors Affecting the Design and Use of Reusable Components

Reghu Anguswamy

Dissertation submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
In
Computer Science and Applications

June 13th, 2013
Falls Church, VA

Keywords: Software Reuse, Reuse Design Principles, Reusable Components, Empirical Study

Factors Affecting the Design and Use of Reusable Components

Reghu Anguswamy

ABSTRACT

Designing software components for future reuse has been an important area in software engineering. A software system developed with reusable components follows a '*with*' reuse process while a component designed to be reused in other systems follows a '*for*' reuse process. This dissertation explores the factors affecting design *for* reuse and design *with* reusable components through empirical studies. The studies involve Java components implementing a particular algorithm, a stemming algorithm that is widely used in the conflation domain. The method and empirical approach are general and independent of the programming language. Such studies may be extended to other types of components, for example, components implementing data structures such as stacks, queues etc.

Design *for* reuse: In this thesis, the first study was conducted analyzing one-use and equivalent reusable components for the overhead in terms of component size, effort required, number of parameters, and productivity. Reusable components were significantly larger than their equivalent one-use components and had significantly more parameters. The effort required for the reusable components was higher than for one-use components. The productivity of the developers was significantly lower for the reusable components compared to the one-use components. Also, during the development of reusable components, the subjects spent more time on writing code than designing the components, but not significantly so. A ranking of the design principles by frequency of use is also reported. A content analysis performed on the feedback is also reported and the reasons for using and not using the reuse design principles are identified. A correlation analysis showed that the reuse design principles were, in general, used independently of each other.

Design *with* reuse: Through another empirical study, the effect of the size of a component and the reuse design principles used in building the component on the ease of reuse were analyzed. It was observed that the higher the complexity the lower the ease of reuse, but the correlation is not significant. When considered independently, four of the reuse design principles: *well-defined interface, clarity and understandability, generality,* and *separate* *concepts from content* significantly increased the ease of reuse while *commonality and variability analysis* significantly decreased the ease of reuse, and *documentation* did not have a significant impact on the ease of reuse. Experience in the programming language had no significant relationship with the reusability of components. Experience in software engineering and software reuse showed a relationship with reusability but the effect size was small. Testing components before integrating them into a system was found to have no relationship with the reusability of components. A content analysis of the feedback is presented identifying the challenges of components that were not easy to reuse. Features that make a component easily reusable were also identified. The Mahalanobis-Taguchi Strategy (MTS) was employed to develop a model based on Mahalanobis Distance to identify the factors that can detect if a component is easy to reuse or not. The identified factors within the model are: size of a component, a set of reuse design principles (*well-defined interface, clarity and understandability*, *commonality and variability analysis,* and *generality*), and component testing.

Dedicated to my parents

Annakkodi and *Anguswamy*

ACKNOWLEDGEMENT

First and foremost, I would like thank my academic advisor, Dr. William B. Frakes, who has constantly given me enormous support during both good and difficult times. I owe a great deal to his understanding of me and my strengths/weaknesses. He has always encouraged me to be independent and aggressive in my research, at the same time he has been guiding me with his invaluable knowledge and experience. I greatly appreciate him for introducing and exploring deep into the field of software reuse. His enthusiasm and commitment to the field has been a constant motivation factor.

I would like to thank my committee members for their continuous feedback. Dr. Gabriella Belli has especially guided me enormously in the statistical analysis and survey research parts of my dissertation. I always enjoyed every meeting with her and was very useful in laying a foundation for making my dissertation robust. Dr. Gregory Kulczyzki has been constantly guiding me in fields related to software engineering and formal methods. He always gave a new dimension of thoughts that helped me explore deep for my dissertation. I also thank Dr. Ing-Ray Chen and Dr. Okan Yilmaz for their continuous feedback and spending time in going over my work which has helped to enrich my dissertation in many ways.

The graduate school under the Dean Karen DePauw has been funding me throughout my PhD here at Virginia Tech. I sincerely thank for funding and supporting me through the Research Methods Consortium (RMC). I am also greatly touched by the understanding and support from the Dean DePauw during the times when I had to take a few months off due to a medical emergency. It was great fun working with Dr. Frakes and Dr. Belli as part of the RMC. I learnt a lot from them outside my PhD work. I will always cherish our meetings.

Last but not least, I would like to thank my family and friends for all their love and unconditional support: my mom – Annakkodi, my dad – Anguswamy, my sisters – Kavitha and Vanitha, my brothers-in-law – Sivaraman and Karunakaran, my nieces – Ammu (Aiswarya) and Paapu (Sandhya), my nephews – Appu (Aswin) and Pappu (Aakash). I am not going to list my friends as there are too many close to my heart, I am sure they would understand as they always have been.

Table of Contents

Chapter 1: Introduction 1

1.1 Problem Formulation and Motivation 2

1.1.1 Design *for* Reuse 2

1.1.2 Design *with* Reuse 3

1.1.3 Expert Opinion 4

1.1.3.1 Reuse Design Principles 5

1.1.3.2 Designing and Building for Reuse 5

1.1.3.3 Designing and Building with Reusable Components 6

1.2 Research Hypotheses 7

1.2.1 Designing and building *for* reuse 7

1.2.2 Designing and building *with* reusable components 8

1.3 Contributions 9

1.4 Dissertation Outline 11

Chapter 2: Background and Related Work 13

2.1 Software Reuse and Success Stories 13

2.2 Component-Based Software Engineering 17

2.3 One-use Component vs. Reusable Component 21

2.3.1 Example 21

2.4 Software Reuse Design Process and Evaluation 23

Chapter 3: Reuse Design Principles 26

3.1 Abstraction 27

3.2 Clarity and Understandability 29

3.3 Commonality and Variability Analysis 30

3.4 Composition 31

3.5 Documentation 31

3.6 Encapsulation and Information Hiding 33

3.7 Generality 35

3.8 Genericity 36

3.9 Isolate Context and Policy 37

3.10 Linking of Test to Code 38

3.11 Modularization 38
3.12 One Component Use Many Helper Components 39
3.13 Optimization 40
3.14 Parameterization 41
3.15 Restrictiveness 42
3.16 Self-documenting Code 42
3.17 Separation of Concepts from Content 45
3.18 Variability Mechanisms 46
3.19 Well-defined Interface 47
Chapter 4: Building and Designing *for* Reuse 53
4.1 Hypotheses 53
4.2 Method 55
4.2.1 A S-Stemmer Component 55
4.2.2 Data Collection 56
4.2.3 Evaluation Metrics 56
4.3 Results and Analysis 57
4.3.1 Demographics 57
4.3.2 Reuse Design Principles 60
4.3.3 Content Analysis$^+$ 62
4.3.3.1 Why the reuse design principles were used 64
4.3.3.2 Why the reuse design principles were *NOT* used 69
4.3.4 Correlation between reuse design principles 71
4.3.5 SLOC, Effort, Productivity and Parameters 74
4.3.5.1 Design, coding, and testing efforts for reusable components 78
4.3.5.2 Matched Pair t-tests 80
4.3.6 Size vs. Reuse Design Principles 81
4.4 Code Examples – Illustrating Reuse Design Principles 84
4.4.1 Well-defined interface 84
4.4.2 Documentation 86
4.4.3 Generality 88
4.4.4 Clarity and Understandability 89

4.4.5 Separate Concept from Content 89
4.5 Threats to Validity 90
4.5.1 Threats to External Validity 90
4.5.2 Threats to Internal Validity 91
Chapter 5: Designing and Building *with* Reusable Components 92
5.1 Hypotheses 93
5.2 Method 94
5.2.1 Subject Demographics 94
5.2.1.1 Roles of the subjects 95
5.2.1.2 Experience in software engineering and programming 95
5.2.1.3 Experience in software reuse 96
5.2.1.4 Experience Levels in Java Programming 97
5.3 Data Collection 98
5.3.1 Component allocation 98
5.3.2 Description of selected components 101
5.4 Results and Analysis 102
5.4.1 Complexity of components vs. reusability of the components 105
5.4.2 Reuse design principles vs. average reusability of the components 106
5.4.2.1 Well-defined interface 107
5.4.2.2 Documentation 109
5.4.2.3 Clarity and Understandability 109
5.4.2.4 Generality 112
5.4.2.5 Separate concept from content 112
5.4.2.6 Commonality and variability analysis 115
5.4.3 Subject experience levels vs. reusability 116
5.4.3.1 Experience levels in software engineering vs. Reusability 117
5.4.3.2 Experience in Reuse vs. Reusability 118
5.4.3.3 Experience in Java vs. Reusability 119
5.4.4 Component Testing vs. Reusability 119
5.4.5 Content Analysis$^+$ 120
5.4.5.1 Why components were NOT reused 121

5.4.5.2 Why components were NOT EASILY reused 122
5.4.5.3 Why components were EASILY reused 124
5.4.6 Mahalanobis-Taguchi Strategy 126
5.4.6.1 Construction of the Mahalanobis Space (MS) 128
5.4.6.2 Taguchi Strategy 130
5.4.7 Stepwise Regression 132
5.4.8 Threats to validity 133
5.4.8.1 Threats to Construct Validity 133
5.4.8.2 Threats to Internal Validity 134
5.4.8.3 Threats to External Validity 134
Chapter 6: Summary and Conclusions 135
6.1 Design *for* Reuse 135
6.2 Design *with* Reuse 137
6.3 Recommendation for Research 138
6.4 Recommendations for Practice 139
6.5 Publications 139
References........143
Appendix A: Software Reuse – Expert Opinion Survey 154
Appendix B: Demographics Survey (for Chapter 4) 159
Appendix C: Demographics Survey (for Chapter 5) 161
Appendix D: Component Reuse Survey – Chapter 5 164
Appendix E: Code Example 1 167
Appendix F: Code Example 2 173
Appendix G: Code Example 3 182
Appendix H: Code Example 4 191
Appendix I: Code Example 5 207
Appendix J: IRB Approval Letters 212

List of Figures

Figure 1. Mindmap of the reuse design process 27
Figure 2. Experience of the subjects in software engineering and programming......................... 58
Figure 3. Distribution of subject professional roles... 59
Figure 4. Experience of the subjects in Java... 59
Figure 5. Experience of the subjects in the field of software reuse 60
Figure 6. Pareto ranking of the reuse design principles... 61
Figure 7. Distribution of the number of reuse design principles used... 61
Figure 8. Comparison of actual size (SLOC).. 75
Figure 9. Comparison of effort (hours)... 75
Figure 10. Comparison of #parameters... 76
Figure 11. Comparison of productivity (SLOC/hr) 76
Figure 12. Density distributions of SLOC, effort, productivity and number of parameters......... 79
Figure 13. Distributions of the times for design, coding, and testing for reusable components... 80
Figure 14. Size comparison of components when well-defined interface was used vs. when not used 81
Figure 15. Size comparison of components when documentation was used vs. when not used .. 82
Figure 16. Size comparison of components when clarity and understandability was used vs. when not used 82
Figure 17. Size comparison of components when generality was used vs. when not used.......... 83
Figure 18. Size comparison of components when separate concepts from content was used vs. when not used 83
Figure 19. Size comparison of components when commonality and variability was used vs. when not used 84
Figure 20. Distribution of the roles the subjects has in their organizations.................................. 95
Figure 21. Distribution of the subjects' experience in software engineering and software programming... 96
Figure 22. Distribution of the subjects' experience in the field of software reuse........................ 97
Figure 23. Distribution of the subjects' experience levels in Java programming 98
Figure 24. Distribution of the 25 components' SLOC (source lines of code)............................. 103

Figure 25. Distribution of the ease of reusability scores 104
Figure 26. Distribution of the average scores of reusability for the 25 components 105
Figure 27. Bivariate fit of SLOC vs. the average reusability scores of the components 106
Figure 28. Distribution of the reusability scores of the components which had well-defined interfaces 108
Figure 29. Box-plot comparison of the reusability scores of components with and without a well-defined interface 108
Figure 30. Distribution of the reusability scores of the components which had documentation 110
Figure 31. Box-plot comparison of the reusability scores of components with and without documentation 110
Figure 32. Distribution of the reusability scores of the components which had the reuse design principle "clarity and understandability" 111
Figure 33. Box-plot comparison of the reusability scores of components with and without clarity and understandability 111
Figure 34. Distribution of the reusability scores of the components built with generality 113
Figure 35. Box-plot comparison of the reusability scores of components with and without generality 113
Figure 36. Distribution of the reusability scores of the components which separated concept from content 114
Figure 37. Box-plot comparison of the reusability scores of components with and without separated concept from content 114
Figure 38. Distribution of the reusability scores of the components built by analyzing commonalities and variabilities 115
Figure 39. Box-plot comparison of the reusability scores of components with and without analysis of commonality and variabilities 116
Figure 40. Comparison of reusability scores for components with only Javadocs, only internal/external documentation (IE-Docs), and those with both 126
Figure 41. Comparison of Mahalanobis Distance (MD) values 130

List of Tables

Table 1. Organization of the dissertation Chapters 11
Table 2. Organization of the dissertation Appendices 12
Table 3. Benefits of software reuse in some reported studies from the software industry 14
Table 4. hello.c (one-use component) VS. anymessage.c (reusable component) [19] 22
Table 5. Cross-reference between the reuse design principles and the literature 52
Table 6. Ranking of reuse design principles used 62
Table 7. Summary of content analysis - why the reuse design principles were used 64
Table 8. Summary of content analysis - why the reuse design principles were NOT used 70
Table 9. Correlation between the reuse design principles used (pearson's coefficient) 73
Table 10. Descriptive statistics of the times spent for designing, coding, and testing the reusable components 79
Table 11. Matched pair t-test statistics 81
Table 12. Component allocation matrix 99
Table 13. Distribution of the reuse design principles in the components selected for this study 101
Table 14. Distribution of the OS, programming language, and IDE used by the subjects to develop their applications 103
Table 15. Ease of reusability for components built with and without reuse design principles... 107
Table 16. Contingency table: Experience in Software Engineering vs. Reusability score (N=170) 118
Table 17. Contingency table: Experience in Software Reuse vs. Reusability score (N=170) 118
Table 18. Contingency table: Experience in Java vs. Reusability score (N=170) 119
Table 19. Contingency table: Component testing vs. Reusability score (N=170) 120
Table 20. Summary of content analysis for feedback of components not reused (score of 1) ... 121
Table 21. Summary of content analysis for feedback of components not easily reused (reusability scores of 2 or 3) 123
Table 22. Summary of content analysis for feedback of components that were easily reused (reusability scores of 4 or 5) 125
Table 23. Summary statistics of the Mahalanobis Distance (MD) values 129
Table 24. S/N ratio and gain of the variables based on the Mahalanobis Space (MS) 131

Table 25. Stepwise regression results 132
Table 26. Stepwise regression final model 133

Chapter 1: Introduction

Software reuse has been widely studied over the past four decades. “Software reuse, the use of existing software artifacts or knowledge to build new systems, is pursued to realize benefits such as improved software quality, productivity, or reliability [1]”. Approaches to measuring reuse and reusability can be found in [2].

Software reuse in industry has been studied and its benefits analyzed [3-12]. These papers document an improvement in software quality and productivity from software reuse. There are many types of software reuse [2].

Component-based software engineering (CBSE) has been a direct result of advances in software reuse. Designing software components for future reuse has been an important area in software engineering. Various characteristics, desired properties, and design principles for CBSE have been studied and analyzed. A software system developed with reusable components follows a ‘*with*’ reuse process while a component designed to be reused in other systems follows a ‘*for*’ reuse process.

In the *for* reuse process, the overarching question is to study how components are built for reuse and how the process affects the quality of the components. There has been no empirical study to identify the most commonly used reuse design principles. In this dissertation, through an empirical study these principles are identified. In the *with* reuse process, successful reuse of the components depends on how easily a user can integrate them into a system. It is important to understand the factors that affect the ease of reuse.

The empirical studies presented in this dissertation involve components implementing a stemming algorithm which is one of the simplest and widely used in the conflation domain [13]. In the first of the two studies in this dissertation, the subjects built components implementing the stemming algorithm in Java. In the second study, the subjects reused the Java components. Though the studies involve only components implementing a particular algorithm in only one language, the method and empirical approach are general. Such studies may be extended to other types of components, for example, components implementing data structures such as stacks, queues etc. The method and the empirical approach are also independent of the programming language.

1.1 Problem Formulation and Motivation

1.1.1 Design *for* Reuse

Many reuse design principles have been proposed [14-17], but there has been little empirical analysis of their use. Ramachandran [18] categorized reuse design guidelines into six different classes: language-specific, design-specific, domain-specific, product-specific, architecture-specific, and organizational/managerial-specific. However, there is no generally accepted list of reuse design principles that are independent of the language and the domain. As an initial attempt to begin creating such a list, the literature of software reuse and reuse design over the past few decades have been analyzed. The results of this analysis are in Chapter 3, which provides a discussion of language and domain independent reuse design principles.

Practitioners and researchers also need to address the problem of how to build reusable components. Sametinger [15] identified that non-reusability of found components is a major obstacle to the success of software reuse. According to Sametinger, software is seldom written effectively and it may be more efficient to build it from scratch. Hence, a guideline of design principles used in building reusable components is necessary. Based on the literature review, the most frequently reported reuse design principles were presented to a group of programmers who had been instructed to develop a reusable component. They were asked to indicate which design principles they had used and why. The purpose for this exploratory study was to identify the most commonly chosen reuse design principles when developing a reusable component. The results of this study, which can be used as a guideline for building reusable components, are presented in Chapter 4.

The software reuse literature often refers to a one-use component and its reusable equivalent, but there has been little study of this concept. Even though the relationship between software quality and reuse has been established, no empirical study has been found comparing one-use and equivalent reusable components. In Chapter 4, the differences between one-use and reusable components are quantified in terms of their sizes, number of parameters used, and effort required based on the model in [19].

One study that is similar is presented by Seepold and Kunzmann [20] for components written in VHDL (Very-high-speed integrated circuits Hardware Description Language). However, the major limitation in that study was that it involved a very small sample size (only four components - two one-use and two equivalent reusable components). According to that study the complexity, effort and productivity were all higher for reusable components. The reasons identified were due to overhead in domain analysis, component verification and documentation.

1.1.2 Design *with* Reuse

A common belief is that the larger the component the harder to reuse. Even in popular cost estimation models such as COCOMO II (COnstructive Cost Model II) [21] which consider software reuse, the cost is estimated higher for larger reusable components. Vitharana [22] discussed the challenges and risks for three stakeholders involved in component-based software engineering (CBSE): the component developers (programmers or engineers involved in developing reusable components), application assemblers (personnel involved in using and integrating the reusable components into the system), and customers. One of the challenges discussed for the component developers is that the size of the components and their dependencies play a vital role in their successful reuse by the application assemblers. In Chapter 5, the effect of the size of components and the reuse design principles on the ease of reuse is analyzed.

Lucredio et al. [12] conducted a study on the status of software reuse in the Brazilian software industry. They identified some of the key factors in adopting an organization-wide software reuse program. They surveyed 57 Brazilian organizations - 25 small (less than 50 employees), 11 medium (50-200 employees) and 21 large (more than 200 employees) organizations. The success rate of adopting software reuse was 64% for small companies, 27% for medium companies, and 52% for large companies. The overall success rate was 53%. An organizational factor that affected the success of reuse in small and medium companies was development experience. Companies with professionals having more than 5 years of experience had significantly higher success than companies with professional having less than 5 years of experience. However, there have been few empirical studies of the relationship between programmer demographics and the ease of reuse. In one study [23], the correlation between programming and UNIX experience, and the effectiveness of searching components was studied. The relationship was found to be not significant. In Chapter 5, through an empirical study the effect of a programmer's demographics such as experiences in programming, software reuse, and programming languages on the ease of reuse is analyzed.

1.1.3 Expert Opinion

Based on these observations, it was thought that it would be good to get some industry participation and to test opinions related to reusable components in the industrial environment. A personal opinion survey (refer Appendix A for the survey) was conducted among the members of a software reuse group called the ESDS-SRWG: Earth Science Data Systems Software Reuse Working Group (http://earthdata.nasa.gov/our-community/esdswg/software-reuse-srwg). The group has members from different organizations including the NASA Jet Propulsion Laboratory, the University of Southern California, and the Center for International Earth Science Information Network (CIESIN) in Columbia University. The members are active in the software reuse industry and possessed considerable expertise in the field. Seven members took part in the survey. Four members were in the software engineering industry for more than 16 years, one had 8-16 years of experience and the rest two had 4-8 years. In the field of software reuse and software programming, 3 of them had more than 16 years of experience and another 3 had 8-16 years of experience. Five of the 7 participants had received training on designing and building software components for reuse. Two of them were trained through college courses in software engineering. The others were through workshops, conferences, or self-training using books.

1.1.3.1 Reuse Design Principles

A list of reuse design principles were given to the survey participants. The participants were asked to comment if there are any reuse design principles not included in the list. Three of them said yes. One of them mentioned that maintainability and portability were not included in the list. However, they are *desired characteristics* of reusable components and cannot be considered as reuse design principles. *Desired characteristics* are those properties of a component that makes it reusable and the reuse design principles are applied in designing and building the component to achieve those characteristics. Another principle pointed out was the use of clear use case examples. This however is either an example of documentation or a link to test code, both of which are already included in the list.

1.1.3.2 Designing and Building for Reuse

The members of the reuse group were asked to give their personal opinions comparing one-use and reusable components in terms of 4 characteristics: size, effort required, number of parameters, and productivity.

- "*One-use components will be smaller than their equivalent reusable components*" – 2 members agreed that this statement is true while two said the statement is false. Two others said they didn't know. One of them mentioned that though the reusable components are generally larger in size, it is not always so and hence they cannot say whether the statement is true or false.
- "*Reusable components require lesser effort to be built compared to its equivalent one-use components*" – Five of them said this is false indicating reusable components require more effort than their equivalent one-use components.
- "*Reusable components will have more parameters than its equivalent one-use components*" – Three members said the statement is true, two said it's false and one member said don't know.

- "*Productivity i.e., number of lines of code written per hour will be lower when building reusable components*" - Three members said the statement is true, two said it's false and one member said don't know.

One member did not answer for any of the above four statements. The member commented that these statements do not have a clear true or false answer as there are many conditions which affect the statements. Based on these responses, we can see that there is no consensus among this sample of experts in comparing reusable components with their equivalent one-use components. Hence, there is a need to explore the comparison between one-use and reusable components through an empirical study.

1.1.3.3 Designing and Building with Reusable Components

Based on their experiences and knowledge, the members were asked to answer true/false/don't know on statements related to designing and building with reusable components.

- "*The larger the size of the component, the easier it is to reuse.*" – Five members said the statement is false indicating that components are easier to reuse when they are smaller. One member said the statement is not necessarily true because size is not an indicator of reuse complexity. The member also said the interface and the behavior of the components need to be well documented irrespective of the size of the components.
- "*The higher the experience of the user in software programming, the easier it is for the user to reuse a code component.*" – Five of the members agreed with this statement, while one disagreed.
- "*The lower the experience of the user in the programming language (in which the code component is written), the easier it is for the user to reuse the code component.*" - Five of the members agreed to this statement, while one disagreed.

One member did not answer any of the above three statements. The member commented that these statements do not have a clear true or false answer as there are many conditions which affect the statements. We can see that though majority has the same opinion there is no consensus among the experts. Hence, there is a need to empirically address the relation between the demographics of programmers and the ease of reuse.

1.2 Research Hypotheses

1.2.1 Designing and building *for* reuse

Four hypotheses related to reusable components were studied. Due to the higher complexity and functionality of the reusable components, their size (in SLOC - source lines of code), effort (in hours), the productivity (in source lines of code per hour), and number of parameters should be significantly higher than their equivalent one-use components. These hypotheses are summarized in equations (1), (2), (3), and (4). $SLOC_{Reuse}$ is the actual source lines of code in the reusable component while $SLOC_{ReuseDiff/hour}$ is the difference in the source lines of code between the reusable and one-use components. The difference is considered for the productivity of reusable components because the reusable components studied in this paper were not built from scratch; instead, they were reengineered by modifying the one-use components.

Hypothesis I-a: A reusable component is larger than its equivalent one-use component.

$$SLOC_{Reuse} > SLOC_{one\text{-}use} \quad (1)$$

Hypothesis I-b: A reusable component requires more development effort than its equivalent one-use component.

$$Effort_{Reuse} > Effort_{one\text{-}use} \quad (2)$$

Hypothesis I-c: When designing and building a reusable component, the developer is more productive (i.e. number of SLOC written per unit time) than when the developer designs and builds an equivalent one-use component.

$$SLOC_{ReuseDiff/hour} > SLOC_{one\text{-}use/hour} \tag{3}$$

Hypothesis I-d: A reusable component has more number of parameters than its equivalent one-use component.

$$Parameters_{Reuse} > Parameters_{one\text{-}use} \tag{4}$$

1.2.2 Designing and building *with* reusable components

When reusable components are used in other applications, four hypotheses were studied and tested. In general components are considered less complex when smaller in size measured by source lines of code (SLOC). Hence, smaller components should be easier to reuse. In section 4.3.3 a discussion on the direct correlation between complexity and size is provided. When a component is built *for* reuse, the reuse design principles used should aid improvement in the ease of reuse. Generally, experience is an indicator of expertise. Hence, a programmer with higher experience should be reusing components with greater ease. Also, when a programmer tests a component before using it, the programmer gets a better understanding of the component. This should improve the ease of reusing the component.

Hypothesis II-a: The smaller the component the easier it is to reuse. The size is measured in SLOC (source lines of code).

Hypothesis II-b: A component designed and built with a given reuse design principle will be easier to reuse than a component which is not built using that reuse design principle. In this study, the effect of the six most used reuse design principles as identified in the study in Chapter 4 are considered: *well-defined interface, documentation, clarity and understandability, generality, separate concepts from contents,* and *commonality and variability.*

Hypothesis II-c: The more the experience a programmer has the easier it is for the programmer to reuse a component. For Hypothesis II-c, three types of experiences in a programmer are considered – programming experience, software reuse experience, and programming language experience.

Hypothesis II-d: A component, when tested by the user before reuse, is easier to reuse than a component which is not tested by the user before reuse.

The studies in the dissertation involve graduate level students at Virginia Tech, U.S., as subjects. The issue of using students as subjects in software engineering experiments has been discussed in the past [24-28] and there has been mixed results on whether students could provide the same results as using professionals. However, the students considered in these studies are full-time undergraduate students. The subjects in this study are mostly part-time graduate students and are working professionals with varying experience levels in the software industry. Carver et al. [27] have mentioned that the gap between students and novice professionals are decreasing especially in the context of the US educational climate. The data were collected as part of assignments in a coursework environment. Based on the faceted classification on types of software reuse by Frakes and Terry [2], the environment of the studies involve:

- *Development scope*: internal (reusable components are from within the project)
- *Modification*: white box (internal modification is allowed i.e. re-engineering)
- *Domain scope*: vertical (within a domain)
- *Management*: ad hoc (reuse is not systematic)
- *Reused entity*: code

1.3 Contributions

The major contributions of this dissertation are:

- A list of reuse design principles have been used in this dissertation based on reviewing the literature over the past few decades. This list may be used and updated in future work.

- Through an empirical study, the most commonly used reuse design principles for re-engineering components to be reusable have been identified. The reasons for using the principles are also identified. This can be a guideline for developers to build reusable components.
- One-use and their equivalent reusable components have been compared based on complexity (in SLOC), effort required, parameters, and productivity. These results may be used or referred to for cost-estimation models.
- Factors, including user demographics and component characteristics, affecting the ease of reusing components are also identified through an empirical study. They can be used as a guideline for managers selecting personnel and components for use in their software projects.
- The method followed is in itself an important contribution. Such studies may be replicated in industry as well because the method is generic and can be easily carried out.
- Empirical evaluation and validation followed in the dissertation is also an important contribution because such an approach can be used as a model in industry to study various phenomena related to reusable components and how they affect the productivity of developers.
- Implications for research and practice based on the results of the studies in this dissertation have also been provided in sections 6.3 and 6.4.
- Publications based on the work presented in this dissertation are given in Chapter 6 (section 6.5).
- A direct result of the work based on this dissertation is the DReMeR '13: International Workshop in Designing Reusable Components and Measuring Reusability (http://www.nvc.cs.vt.edu/ICSRworkshop-DreMeR-13/index.html) held in conjunction with the 13th International Conference on Software Reuse, ICSR '13: http://softeng.polito.it/ICSR13/index.html at Pisa, Italy on 18 June, 2013.

1.4 Dissertation Outline

The dissertation Chapters and Appendices are organized as described below in Tables 1 and 2.

Table 1. Organization of the dissertation Chapters

Chapter	Title	Description
2	Background and Related Work	• discusses the background and reviews the related work on software reuse, component-based software engineering, reusable components, and reuse design process.
3	Reuse Design Principles	• discusses and analyzes the set of reuse design principles identified in the literature over the past few decades
4	Building and Designing *for* Reuse*	• presents an empirical evaluation of a study related to designing and building *for* reuse • studies and tests the hypotheses I-a, I-b, I-c, and I-d • also explores and identifies the most commonly used reuse design principles • correlation between reuse design principles • content analysis on the feedbacks for *why* and *why not* the reuse design principles were used
5	Designing and Building *with* Reusable Components*	• presents an empirical study related to designing and building *with* reusable components • studies and tests the hypotheses II-a, II-b, II-c and II-d • content analysis on the feedbacks reasoning the reusability of components • Mahalanobis-Taguchi Strategy (MTS) to identify factors affecting reusability of components • Stepwise regression to identify factors affecting reusability of components and comparing it to the results from MTS
6	Summary, Conclusions, and Future Work	• Summary, Conclusions, and Future Work

*The code examples and snippets written by external authors have been modified (such as removing headers, modifying variable names etc.) so that the author is not identifiable; they are provided to enhance the understanding of related concepts and results.

Table 2. Organization of the dissertation Appendices

Appendix	Title	Description
APPENDIX A	Software Reuse – Expert Opinion Survey	• Survey Questionnaire for the Expert Opinion on reusable components
APPENDIX B	Demographics Survey (for Chapter 4)	• Survey Questionnaire for the demographics of the subjects in the study in Chapter 4
APPENDIX C	Demographics Survey (for Chapter 5)	• Survey Questionnaire for the demographics of the subjects in the study in Chapter 5
APPENDIX D	Component Reuse Survey – Chapter 5	• Survey Questionnaire for the reusability of components by the subjects in the study in Chapter 5
APPENDIX E-I	Code Example*	• Code example for components used in the studies for Chapters 4 and 5
APPENDIX J	VT-IRB Approval Letters	• VT-IRB approval letters for the survey studies in the dissertation

*The code examples and snippets written by external authors have been modified (such as removing headers, modifying variable names etc.) so that the author is not identifiable; they are provided to enhance the understanding of related concepts and results.

Chapter 2: Background and Related Work

This chapter presents the basic background and related work for software reuse, component-based software engineering, one-use vs. reusable components, and reuse design process.

2.1 Software Reuse and Success Stories

Software reuse has been successfully implemented in industry. Some major companies that published their success are (the benefits are summarized in Table 3):

- **1980s:** Boeing [29], Hartford Insurance Group [30], Intermetrics, Inc. [31], NASA/Goddard Space Flight Center [32], Raytheon [33]

- **1990s:** IBM [34-36], Hewlett-Packard [37, 38], Motorola [39], Sodalia [40, 41], Thomson-CSF[41], Nippon Electric Company (NEC) [42, 43], GTE [44]

- **2000s:** Orbotech [9], ISWRIC (Israel SoftWare Reuse Industrial Consortium – a consortium of seven software companies in Israel) [45], Ericsson [4], NASA/Earth Science Data Systems (ESDS) [46]

Lucredio et al. [12] conducted a study on identifying the scenario of software reuse in the Brazilian software industry. They surveyed 57 Brazilian small (less than 50 employees), medium (50-200 employees), and large (more than 200 employees). The success rate of adopting software reuse was 64% for small companies, 27% for medium companies, and 52% for large companies. The overall success rate was 53%. An organizational factor which affected the success of reuse in small and medium companies was development experience. Companies with professionals having more than 5 years of experience had a significantly higher success than the companies with professionals having less than 5 years of experience.

Morisio et al [6] identified success and failure factors in the reuse industry by conducting an empirical study based on a survey of 24 companies in Europe. One of the failure factors identified was human factors, i.e. the lack of training and education of the developers in the companies. It was also identified that addressing the human factors achieved success in software reuse. Kotov [47] also conducted a survey, based on the same questionnaire as used by Lucredio et al. [12], to investigate the field of software reuse in software development organizations in Latvia. They had data from 18 companies in Latvia. 72% of the respondents claimed to succeed in projects by the means of software reuse in their organization. The influence of the approach for success was similar: 80% for component-based approach and 79% for object –oriented approach. Influence of programming language on the success of reuse was found to be very less. Java (86%) and Ruby (100%) had the highest reuse success percentages; the rest were all between 50% and 80%.

Table 3. Benefits of software reuse in some reported studies from the software industry

	Company/Industry	Higher Quality	Lower defect density	Improved time to market	Reduced Cost	Higher Productivity	Lower Maintenance Cost
1980s	Hartford Insurance [30]	X				X	
	Intermetrics Inc. [31]					X	
	Raytheon[33]					X	
1990s	Fujitsu [48]			X			
	IBM [34-36]	X	X				
	Hewlett Packard [37, 38]	X	X	X		X	
	NEC [42, 43]	X				X	
	Motorola [39]	X		X		X	
	Toshiba [48]		X				
2000s	Orbotech [9]	X		X	X		
	ISWRIC [45]				X		
	Ericsson [4]	X	X				X
	NASA/ESDS [46]	X			X		

Chen et al. [49] conducted a questionnaire-based survey of software development with Open-Source Software (OSS) components used in the software industry in China. They had data from 47 development projects across 43 companies. They found that nearly 84% of the components needed bug-fixing or modification. They also found that learning cost is a major expense in reusing the OSS components.

Ezran et al. [17] reported some examples of estimated benefits in the software industry due to systematic software reuse covering use of various programming languages including Ada, Cobol, and C++:

- DEC
  - Cycle time: 67%-80% lower (reuse levels 50-80%)
- First National Bank of Chicago
  - Cycle time: 67%-80% lower (reuse levels 50-80%)
- Fujitsu
  - Proportion of projects on schedule: increased from 20% to 70%
  - Effort to customize package: reduced from 30 person-months to 4 person-days
- GTE [44]
  - Cost: saved $1.5 million during its first year (reuse level 14%; only 38% of the assets in the repository were being reused).
- Hewlett-Packard
  - Defects: 24% and 76% lower (two projects)
  - Productivity: 40% and 57% higher (same two projects)
  - Time-to-market: 42% lower (one of the above two projects)
- NEC – Nippon Electric Company [42, 43]
  - Productivity: 6.7 times higher
  - Quality: 2.8 times better
- Raytheon [33]

    - Productivity: 50% higher (reuse level 60%)

- Toshiba [48]
    - Defects: 20%-30% lower occurrence of bugs at the time of software system integration testing (reuse level 60%)

In spite of the benefits many challenges have been reported in systematic software reuse. Sametinger [15] identified some major technical obstacles in code reuse. Three of them which addressed are:

- *Non-reusability of found software:* Accessing already existing software easily does not necessarily imply that it increases software reuse. Reusable assets should be carefully specified, designed, implemented, and documented, thus, sometimes, modifying and adapting software can be more expensive than programming the needed functionality from scratch.

- *Modification:* It is very difficult to find a component that works exactly in the same way that the developer wants. In this way, modifications are necessary and there should exist ways to determine their effects on the component and its previous verification results.

- *Integration:* Sometimes it is not possible to integrate components into the system, they are of no use. Software components must be constructed in a way that subsequent reuse can be efficient and straightforward.

In Chapter 5, the challenges and factors that result in these obstacles are explored.

Code reuse may be *black box reuse, white box reuse, grey box reuse*, or *glass box reuse* [17]. If a component is reused without any modification and adaptation, it is known as *black box reuse*. If the component is reengineered i.e. if the internal body of the component is modified so that it can be adapted to the system, it known as *white box reuse*. The intermediate situation, where adaptation is achieved by setting parameters, is known as *grey box reuse. Glass box reuse* is the situation where the internal body of the component is on a 'read-only' basis for understanding its properties but cannot be modified. This is useful when the description of the component is inadequate and a developer can look inside the component to understand its properties better. In this thesis the scope of the work is only for white box reuse, i.e. the reusable components can be modified at the code level.

2.2 Component-Based Software Engineering

Component based software reuse was proposed as early as 1968 by McIlroy [50] suggesting that interchangeable pieces called software components should form the basis for software systems. Component-based software engineering (CBSE) has been a direct result of advances in software reuse. Designing software components for future reuse has been an important question in the field of software engineering. Various characteristics, desired properties and design principles for CBSE have been studied and analyzed in the past.

In CBSE, the most important and fundamental principle is to *reuse* software components. In 1998, Kozaczynski et al. [51] suggested that a definitive definition of a software component is hard to come by. However, a software component has been defined in many different ways in the software reuse literature. All the definitions however agree with the intuitive definition that "*Components are things that can be plugged into a system"*[52]. McIlroy[50] invented the concepts of pipes and filters in the Unix operating system to plug the components into the system.

In 2000, Hopkins [53] gave a modern definition as "*A software component is a physical packaging of executable software with a well-defined and published interface.*" Hopkins said there were two engineering drivers for component-based systems[53]:

- **"Reuse**. The ability to reuse existing components to create a more complex system.
- **Evolution**. By creating a system that is highly componentized, the system is easier to maintain. In a well-designed system, the changes will be localized, and the changes can be made to the system with little or no effect on the remaining components."

In 1998, Szyperski et al. [54] defined a software component as "*…a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third party".* Based on this definition, Hasselbring [55] compared objects and components. An object is a unit of instantiation with a unique identity while a component is a unit of deployment in a system. An object has a state but need not be in a persistent state as components. The ‘state’ of an object encompasses all of the (usually static) properties of the object plus the current (usually dynamic) values of each of these properties [56]. An object encapsulates its state and behavior while a component is a unit of third-party composition. Hopkins [53] was of the opinion that practitioners often find building systems from a component perspective is more naturally modeled in object-oriented programming. However, a well-formed component can also be written in a procedural language such as C, providing a well-defined interface and packaged implementation. Coulange [57] strongly supported the opinion that object-oriented programming is the way to achieve reusability. Griss [58] and Sametinger [15], however, disagreed that object-oriented programming alone is sufficient to achieve successful reuse. Morisio et al. [6] in their survey study of the Brazilian industry scenario identified this as a failure factor - the belief of using object-oriented approach and setting up repositories is all that is necessary to achieve success in reuse. Crnkovic et al. [59] suggest that components must be in a ready to use state; there should be no recompiling or relinking by the third party to reuse the component.

Based on the definition by Szyperski, Hasselbring summarizes the technical features of a component as [55]:

- **Coupling** (inter-relatedness among components): In component-based software engineering, coupling for a component is defined as the extent to which the component is coupled with other components. Low coupling is desired.
- **Cohesion** (strength of association among elements within a component): Cohesion refers to the strength of association of elements within a system. In component-based software development, cohesion of a component is the extent to which its contained elements are inter-related. High cohesion is desired.
- **Granularity** (number of components in a system, complexity): In component-based software engineering, the number of components used to realize a particular system is an important design parameter. The trade-off between many small components and a few large components must be considered in component and system design.

Sametinger [15] provides two approaches to define components. One is that components can be seen as some identifiable and reusable parts of a software system. Functions and classes would be examples of such components. Components can also be considered as the next level of abstraction after functions and classes. Based on this, Sametinger provides a more precise definition as: "*Reusable software components are self-contained, clearly identifiable artifacts that describe and/or perform specific functions and have clear interfaces, appropriate documentation and a defined reuse status.*" The elements of the definition are elaborated below:

- **Self-contained**: components should be reusable without using any other components. Functions are components if they do not need any other functions. If they do, then the whole set of functions is considered as a reusable component. Modules and packages are such components which have many functions in them. Function libraries can also be considered as a reusable component, they have many different interfaces and functionalities.
- **Identification**: components must be clearly identifiable; the artifacts could be source code, documentation, or executable code.
- **Functionality**: components must have a clearly specified functionality which they describe or perform. Code components must be implementing a specific functionality. Software life cycle documents such as specifications, requirements, and design documents which describe specific functionalities can also be considered as components.
- **Interfaces**: components must have clear and well-defined interfaces; they must hide details that are not needed for reuse.
- **Reuse status**: maintenance of the components is needed to support systematic software reuse. The reuse status contains information such the owner of the component, who is maintaining it, and the quality status of the component.

Based on this definition, Sametinger argues that design patterns [60], though they can be reused, are not components. Design patterns are realized by taking existing components and arranging them as described. Design patterns cannot be just taken and integrated into a system. Similarly, Sametinger argues that algorithms are also not components. Algorithms may be reused but they are only ideas and need to be implemented in a programming language to be reused in a software system. This is relates to the 3Cs model (discussed later in the introduction of Chapter 3) by Latour et al. [61] – algorithms are concepts while design patterns are content.

The major goal in component-based software engineering is to build systems from reusable components and maintenance is performed by customizing or replacing the components [62]. In a 2007 survey paper, Mahmood et al. summarized the benefits of component-based software development as [63]:

- *Reduced development time (less time required to buy a component than to design, code, test, debug and document it)*
- *Increased flexibility (component-based systems are immune to the implementation of the components, thus there is more choice of components from which to choose so that they meet the requirements)*
- *Reduced process risk (if a component exists, there is less uncertainty in cost associated with its reuse as compared with new development)*
- *Enhanced quality (components are reused and tested in many different applications. Design and implementation faults are discovered and eliminated in the initial use, thus enhancing the quality of the component)*
- *Low maintenance (easy replacement of obsolete components with new enhanced ones)*
- *Standardization (some standards can be implemented as a set of standard component development. The use of these standards will improve the overall quality of the components and systems)*

Challenges and risks of component-based software engineering have also been discussed in the software reuse literature [22, 62, 64-67]. Vitharana [22] discussed the challenges and risks for three stakeholders involved in CBSE: the component developers (programmers or engineers involved in developing reusable components), application assemblers (personnel involved in using and integrating the reusable components into the system), and customers. One of the challenges discussed for the component developers is that the size of the components and their dependencies play a vital role in their successful reuse by the application assemblers. In Chapter 5, the effect of the size of a component on the ease of reuse by application assemblers have been analyzed. In Chapter 4, changes in the sizes, effort required, parameters used, and productivity of the reusable components compared to one-use components have been analyzed. Well-defined interfaces specifying how components work, along with their inputs, outputs, and exception-handling procedures, also pose a considerable challenge for the component developers [22]. Hence, there is a need to understand how the developers design components for reuse. In Chapter 4, the most commonly used reuse design principles are identified and this can be a guideline for the component developers.

2.3 One-use Component vs. Reusable Component

The distinction between a reusable and its equivalent one-use component is an intuitive concept that is not precisely defined. One-use components are generally written for specific applications and are not meant to be used again. Reusable components on the other hand are developed to be used more than once within a domain or across domains for various applications in various environments. Reusable components are often developed by taking a one-use component and modifying it either to add more functionality or changing it to work in other environments following a re-engineering approach. They can also be developed from scratch. This study involves reusable components built by re-engineering one-use components. It follows that compared to equivalent one-use components; reusable components tend to be larger, more complex and slower. They also should have more potential input/output types and more parameters.

2.3.1 Example

As a simple example of a one-use versus a reusable component, consider the "**`hello world`**" program, **`hello.c`**, as a one-use component and the **`anymessage.c`** program as its reusable equivalent. Here is the code for each [19]:

```
//hello.c (one-use component)
main(){printf("hello world\n");}
```

```
//anymessage.c (reusable component)
main(argc, argv) /* print any message to output */
int argc; char *argv[];
{
       int i;
       for (i=1; i< argc; i++)
              printf("%s ",argv[i]);
       printf("\n");
}
```

Table 4 [19] shows the relationships between hello.c and anymessage.c in terms of attributes such as size, complexity, etc. The `wc` program [68] was run in the Unix environment on `hello.c` and `anymessage.c`. As predicted, the reusable program is larger and has more parameters. As hypothesized `anymessage.c` will be more complex, require more testing, require more design knowledge, have higher execution speed, require more time to develop and will be usable in more environments. Thus, Table 3 summarizes the theoretical model of the relationship between a one-use and a reusable component.

Table 4. hello.c (one-use component) VS. anymessage.c (reusable component) [19]

Attribute	hello.c	anymessage.c
Size (lines, chars, executable)	(1, 8, 9878)	(10, 25, 9931)
Complexity	<	>
Parameters	0	2
Domain Knowledge	Low	Medium
Testing	<	>
Design	<	>
Execution Speed	<	>
Effort	<	>
Environments	<	>

One study that is similar in comparing one-use and reusable components is presented by Seepold and Kunzmann [20] for components written in VHDL (Very-high-speed integrated circuits Hardware Description Language). However, the major limitation in that study was that it involved a very small sample size (only four components - two one-use and two equivalent reusable components.) According to that study the complexity, effort and productivity were all higher for reusable components. The reasons identified were due to overhead in domain analysis, component verification, and documentation.

2.4 Software Reuse Design Process and Evaluation

McClure [69] proposed a 5 step process to create a component for reuse:

1. Generalize - Process of abstracting the commonalities and stripping away the differences; most common technique for generalization – parameterization; other techniques are:
   - separation of concept from content (separating internal/implementation details) [70, 71].
   - abstraction – reuse of design rather than code [72].
   - encapsulation [72]
   - analysis of commonalities and variabilities [71]

The component must have two parts – fixed part (for the benefits of black box reuse – use as is form) and variable part (benefit of white box reuse – use by modification). For example, in object-oriented programming the fixed part is the class hierarchy composed of abstract and concrete classes, while the variable part is the empty methods which users can override for application – specific implementation. The variable part also can have restricted customization through defining a range for parameter substitution, enumerated choices, specifying performance constraints etc.

2. Standardize – through documentation, standardized interface design, and standardized testing reusability increased because of higher quality and general usefulness.

3. Automate – for example, use CASE and reengineering tools to generate code, check design consist and completeness, identify redundancy and opportunities for reuse, and perform management functions.
4. Certify – for compliance to standards, links to requirements, complexity metrics, testing and inspection, etc.
5. Reuse specific documentation - 6 types of documentation

Ezran et al. [17] have classified the desired characteristics for reusable components into three criteria:

- General criteria (quality and reusability): compliance to standard, compliance to the software engineering process, completeness of the artifacts and information provided, simplicity and understandability, and modularity.
- Functional criteria: the component's function is to automate or simulate, fully or partially, a business process; must always remain available to the clients etc.
- Technical criteria: interoperability, portability, self-descriptiveness, security etc.

Stroustrup [73] gave 4 criteria that make a component a likely candidate for reuse:

- Generality: A component must represent a general concept. It doesn't matter how elegantly a component is represented or how thoroughly it is documented if it represents the solution to a single particular problem only.
- A clean interface to users: The more complicated an interface is, the more work it is to use the component and the more attractive it becomes to building something specialized instead.
- Well-defined dependencies on "the rest of the system: " The more dependencies a component has on other components, and the messier and unobvious those dependencies are, the harder it is to port the component into a new system.
- Acceptable Efficiency: The importance of efficiency varies enormously. However, run-time or space overheads can be critical, and even where they are not, obvious overhead tempts the designer and programmer to do better with a special purpose solution.

Similarly Coulange [57] has also provided a set of criteria for evaluating reusability of software components:

- Productivity – reuse must increase productivity, reusing a component must be less costly than to develop it.
- Maintainability – error correction costs in systems using the reusable components must be minimal.
- Reliability – functionality of the component reused should not be disturbed
- Extensibility – extensions to the reusable components must be with minimal effort.
- Usability – measure of the independence of the component with respect to each other; i.e. the components must be easily assembled and integrated with other components.
- Adaptability – component must be capable of adapting to different contexts (environments).

Ramachandran [18] categorized reuse design guidelines into six different classes: language-specific, design-specific, domain-specific, product-specific, architecture-specific, and organizational/managerial-specific. He also suggested that reuse design guidelines must be objective and realizable. Such guidelines are important because they:

- help assess the reusability of software components against objective reuse guidelines.
- provide reuse advice and analysis.
- improve the components for reuse, which is the process of modifying and adding reusability attributes.

Ramachandran also had presented a prototype automation tool, for designing components for reuse, known as the Reuse Assessor and Improver System (RAIS) [74], which can interactively identify, analyze, assess, and modify abstractions, attributes, and architectures that support reuse. Practical and objective reuse guidelines are used to represent reuse knowledge and to perform domain analysis. It takes existing components, provides systematic reuse assessment, which is based on reuse advice and analysis, and produces components that are improved for reuse. Their work on guidelines has been extended to a large-scale industrial application [11].

Chapter 3: Reuse Design Principles

"*Reuse is a result of good design: it is not something you get from simple-minded use of special language features*." – Bjarne Stroustrup [73]

Component based development in software reuse was presented as early as 1968 by McIlroy [50] suggesting that interchangeable pieces called software components should form the basis for software systems. A software system developed with reusable components follows a 'with' reuse process while a component designed to be reused in other systems follows a 'for' reuse process. Ramachandran [18] categorized reuse design guidelines into six different classes: language-specific, design-specific, domain-specific, product-specific, architecture-specific, and organizational/managerial-specific. He also suggested that reuse design guidelines must be objective and realizable. Such guidelines are important because they:

- help assess the reusability of software components against objective reuse guidelines.
- provide reuse advice and analysis.
- improve the components for reuse, which is the process of modifying and adding reusability attributes.

Many reuse design principles have been proposed. These are summarized in the mindmap in Figure 1 (presented in Virginia Tech class CS 6704 by Prof. William B. Frakes in Summer 2009) based on the work by Frakes and Lea [14]. The principles are at various levels of abstraction. The 3Cs model of reuse design by Latour at al. [61], for example, was developed to explore the reuse design process in a general framework. It specifies three levels of design abstraction.

- Concept – representation of the abstract semantics.
- Content – implementation details of the code or software.
- Context –environment required to use the component.

How to make a software component reusable has been one of the key questions for software reuse research. Reusable components may be built from scratch or re-engineered from existing artifacts. As can be seen from Figure 1, the quality of the reusable components may be measured in terms of safety (when implemented and/or merged with another component), execution speed (generally the reusable components are slower than the one-use components), cost, and size. Each reuse design principle, as shown in Figure 1, is presented and discussed in detail.

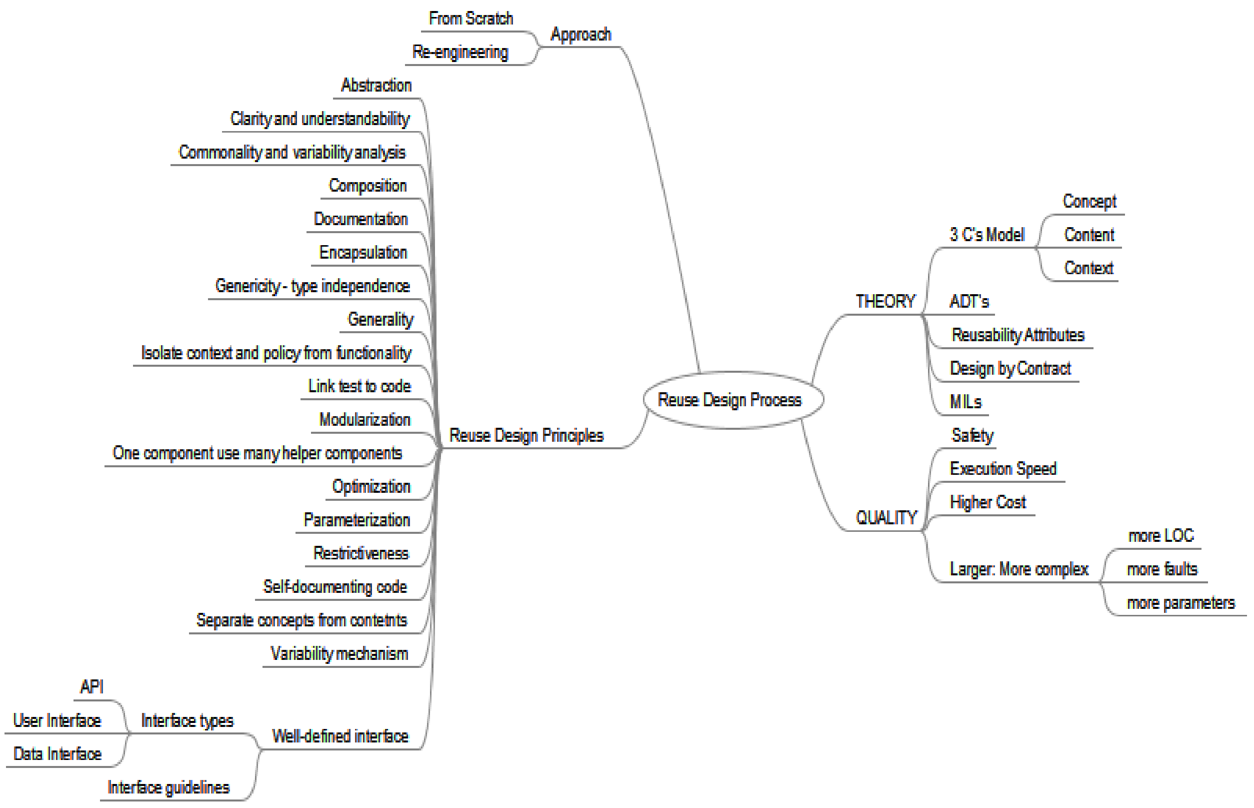


Figure 1. Mindmap of the reuse design process

3.1 Abstraction

Liskov et al. [75] defined abstractions as: "A very-high-level language attempts to present the user with the abstractions (operations, data structures, and control structures) useful to his application area. The user can use these abstractions without being concerned with how they are implemented; he is only concerned with what they do. He is thus able to ignore details not relevant to his application area, and to concentrate on solving his problem."

In general, abstraction means concentrating on important essentials while temporarily ignoring the unimportant details [76]. Sodhi et al. [76] propose that purely object-oriented design and that only object-oriented design can enhance reusability because information hiding in object-oriented programming enforces abstraction and supports reusability; and also helps achieve modularity, increase quality, reliability and maintainability. An abstraction has a hidden part, a variable part, and a fixed part [77]. In the specification of the abstraction, the variable and the fixed part will be visible while the hidden part is not. The fixed part is the non-changing features of the abstraction while the variable part includes the features that could be changed for specific implementations. Jacobson et al. [72] say that abstraction should be *general*, so that it is useful in several applications without having to under changes; and also the component is standardized with respect to name, fault handling, structure and so on.

Leach [78] argues that the level of abstraction should be thin because higher levels of abstraction means higher genericity. Too many levels of abstraction will mean additional testing and integration problems. The relationship between reuse and abstraction has also been well documented [79-82]. Standish [82] suggests that reuse is achieved at the level of abstraction. According to Krueger [77], abstraction plays a central role in software reuse. Concise and expressive abstractions are essential if software artifacts are to be effectively reused. It is argued that if there is no abstraction, developers would be forced to go through the entire collection of components to figure out what each component did, when it could be reused and how it could be reused.

Wegner [83] describes abstraction as an importantly desired property for reusability and defines three types of abstraction:

- Function abstraction – implementation is hidden while only the input/output relationship is visible through interface specification.
- Data abstraction – the data as well as the function within the component is hidden
- Process abstraction – it is like data abstraction with the additional feature of permitting an independently executing thread of control; useful for concurrent and distributed process based programming.

Liskov and Guttag [84] defined three kinds of abstraction as: procedural abstraction (abstraction of a single event or task/procedure which is ‘operation-like’), data abstraction (set of objects and operations that characterize the behavior of the objects), and iteration abstraction (which allows iteration of all the elements without any constraints on the order of the elements).

According to Sodhi et al. [76], abstraction is the opposite of encapsulation. Abstraction focuses on the observable behavior of an object, and encapsulation focuses on the implementation of the behavior. Encapsulation hides the implementation details, whereas the user of the abstraction knows only the essence of the behavior. Liskov et al. [75] had introduced the concept of Abstract Data Types (ADTs): “An abstract data type defines a class of abstract objects which is completely characterized by the operations available on those objects. This means that an abstract data type can be defined by defining the characterizing operation for that type.” The IEEE Standard Glossary of Software Engineering Terminology defines ADT as [85]: “A data type for which only the properties of the data and the operations to be performed on the data are specified, without concern for how the data will be represented or how the operations will be implemented.”

3.2 Clarity and Understandability

Clarity and understandability of software components is an important property for reusability [16, 17, 84, 86-88]. Matsumoto [88] discusses definiteness as a major characteristic to make components reusable; definiteness represents the degree of clarity to which the module's purpose, capability, constraints, interfaces, and required resources are defined. Karlsson defines understandability as [87]: “the attribute of the software that provides explanation on its content and its possible use.” The understandability is measured based on its *self-descriptiveness*: “the criterion that measures how well a component explains its functions. It is provided by standard formats, prologue comments on each modules, etc. [87]”

Braun ’93 [86] provided general guidelines in implementing control structures in ADA for achieving clarity and understandability of algorithms such as:

- labels and goto statements must be avoided
- a loop’s exit condition should be explicit and apparent
- avoid deeply nested loop statements
- do not use case statements when if statements are more appropriate, and vice versa
- use control structures instead of highly complex expressions

Braun ’93 [86] also argues that coding standards and proper indentation can improve readability of the code and understandability of the algorithms. With proper indentation and coding standards the flow of the program with respect to the algorithm implemented is better understood. Expression statements must be structured to avoid ambiguity and provide clarity. Features like overloading should be used to improve understandability by using it for naming similar functions. Using overloading for naming dissimilar functions can reduce understandability.

Karlsson defines understandability as [87]: “the attribute of the software that provides explanation on its content and its possible use.” The understandability is measured based on its *self-descriptiveness*: “the criterion that measures how well a component explains its functions. It is provided by standard formats, prologue comments on each modules, etc. [87]” Karlsson [87] has decomposed understandability into four factors: self-descriptiveness (explanation of how the functionality is implemented), documentation level (accessibility, level of detail and quality of reuse documentation), structure complexity (how easy it is to understand the relationships between component’s parts) and inheritance complexity (how easy it is to understand the relationships between a class and its superclass).

3.3 Commonality and Variability Analysis

Software systems may contain similar sub-systems with common design but with some variations. Commonalities and variabilities are identified within the domain for the component to aid a design for reuse. Sodhi et al. [76] define classification, grouping of objects with behavior (methods and operations) and characteristics (data) as a way of achieving commonality and variabilities. Various techniques have been proposed for the analysis of commonalities and variabilities; Ramachandran et al. [89] have discussed various approaches followed in the industry such as SCV (Scope, Commonality and Variability [90]), FODA (Feature-oriented domain analysis [91]), and also proposed a new approach FARE (Family oriented Analysis and Requirements Engineering). Domain analysis is widely recommended for the analysis of commonalities and variabilities [17, 69, 76].

3.4 Composition

Composition refers to the process of how to connect different software components. Some guidelines include: identify and minimize import requirements (for helpers), identify and minimize interference among helpers, use layering to define complex components using simple ones, implement policy on top of mechanism. Technologies within the Microsoft family such as the .NET framework, DCOM/COM+ (Component Object Model), MTS (Microsoft Transaction Server) and ActiveX, and the Java family such as J2EE (Java 2 Platform, Enterprise Edition), and EJB (Enterprise JavaBeans) support composition to promote reuse [17]. In COM technology introduced by Microsoft, interprocess communication and dynamic object creation in various programming languages is enabled.

Module Interconnection Languages (MILs) help in the composition for building software systems. They provide formal grammar constructs for describing the global structure of a software system and for deciding the various module interconnection specifications required for its complete assembly [92]. The first MIL language was proposed by DeRemer and Kron [93]. According to Shaw and Garlan [94], the key issue in designing a MIL is the nature of the glue code. In the composition model based on *definition/use* bindings [92], each module *defines* a set of facilities available to other modules, and *uses* facilities provided by other modules. The purpose of the glue code is to resolve the *definition/use* relationships by indicating for each use of a facility where its corresponding definition is provided.

3.5 Documentation

The IEEE Standard Glossary of Software Engineering Terminology gives a definition of documentation as [85]: “Any written or pictorial information describing, defining, specifying, reporting, or certifying activities, requirements, procedures, or results.” It is well documented that documentation is an important characteristic for making software components reusable [16, 17, 78, 84, 86-88]. Documentation for software is essential for any future use or modification and critical for maintainability. Programmers are unlikely to reuse software that is not well-documented or commented since it makes it harder to understand and maintain. Documentation should be self-contained, adaptable and extensible [15]. Specific documentation for reuse of the component enhances the chances for usage of the component in future. Braun [86] encouraged documentation embedded in the source code files such that they described the code in a general manner – "the documentation must use generally understood terminology, explain hidden significant implications, if any, and make the declarations in the code more understandable; a rationale for selecting the algorithm must also be provided within the documentation; a consistently formatted prologue for each program unit is also recommended; the prologue must also specify any restrictions on the usage of the program unit, if any."

Leach [78] has encouraged including a "reuser's guide" which includes some of the design rationale of the component that will aid in easy reuse. He has discouraged using self-documentation style in programming like using long variable and function names as they may not be reliable and might be misleading.

McClure [69] suggested that documentation can be in narrative and/or graphic form. Documentation helps easy location, management, retrieval, and maintenance of the reusable components. Six types of documentation are presented for increasing reusability:

- Specific information (high level description of functionality, and if possible a formal semantic description)
- Classification (information in the form of faceted index and keywords that will help classify the component for storage and retrieval)
- Declarative information (information on pre-conditions and post-conditions, assertions, and events/conditions)
- Quality/Certification information (complexity metrics, reliability information such as defect density)
- Reuse information (guidelines for reuse such as range for parameter substation, efficiency performance options etc.; reuse history; systems where the component is used)
- Detailed documentation (additional documentation on reuse guidelines such as input/output, performance documentation, interface requirements, algorithms used, design decisions/trade-offs, limitations, test plans, maintenance support information etc.)

Frakes and Nejmeh [95] argued that every module (a file consisting of one or more functions) and function must begin with a prologue so that the time required to integrate them is reduced and also assure that they perform the necessary operations without harmful side-effects. A prologue for a module should have the following fields [95]: name of the module, abstract, description, references to supporting documents, size (number of functions in the module, number of lines of code in the module, and object size for each machine the module runs on), contents (list of functions in the order in which they appear in the module), global data with brief descriptions, environmental requirements (required hardware and software), documentation quality (comment-to-code ratio and documentation standards used), portability (machines the module will run on), programming standards used, time in use (how long the module has been in use), reuse statistics (projects that have used the module, how and when it was used, and the person who used the module), reuse reviews (reviews from previous users). A prologue for a function should have the following fields [95]: name of the function, author details, date the function was written, abstract, description, keywords, size (number of lines of code in the function, and object size for each machine the function runs on), complexity metrics, performance (execution times), inspection information (reviewed or not), testing details, usage (additional files necessary to call the function), parameters passed to the function with a brief description of each parameter, externals (details of global variables and how they are modified inside the function), macros used, returns (details of the value returned), calls (the functions called by this function along with the modules in which the called functions appear), called by (functions and the corresponding files which call this function), and modification history.

3.6 Encapsulation and Information Hiding

Encapsulation is a key concept in object-oriented programming and according to Snyder [96], "Encapsulation is a technique for minimizing interdependencies among separately written modules by defining strict external interfaces. The external interface of a module serves as a contract between the module and its clients, and thus between the designer of the module and other designers. A module is *encapsulated* if clients are restricted by the definition of the programming language to access the module only via its defined external interface."

Parnas et al. [81, 97] had promoted information hiding as an important design principle to aid reusability of components. According to Jacobson et al. [98], all information in an object-oriented system is stored within its objects and can be manipulated only when the objects are ordered to perform operations. The behavior and information are encapsulated in the object. The only way to affect the object is to perform operations on it. Objects, thus support the concept of information hiding, that is, they hide their internal structure from their surroundings.

Jacobson et al. [98] says that abstract data types and objects are closely related. Both of them are abstractions and are defined in terms of what they perform, not how they perform it. They are both generalizations of something specific, where encapsulation is the central concept. “An abstract data type defines a class of abstract objects which is completely characterized by the operations available on those objects. This means that an abstract data type can be defined by defining the characterizing operations for that type [75].” One advantage with abstract data types is that they can be used independently of their implementation (information hiding), i.e. how the abstract data type is used need not be modified even if the implementation is modified. . In the 3Cs model by Latour et al. [61], encapsulation enables the reuse of the concept without knowing the content. For example, in `C` there is function for obtaining the square root of a number: `double sqrt (double x)`. The function is defined and encapsulated in the header file named `Math.h`. A user just has to include the header file to use the `sqrt` function and need not know the implementation details (content) which is encapsulated in the header file, thus reusing the concept of finding the square root of a number without knowing the content.

Coleman et al. [71] suggested that a good encapsulation principle to improve reusability is to keep the representation hidden and reduce the interface’s dependence on the representation. This permits easy replacement of representations to implement the same interface. According to Sodhi et al. [76], encapsulation is the opposite of abstraction. Abstraction focuses on the observable behavior of an object, and encapsulation focuses on the implementation of the behavior. The user of the abstraction knows only the essence of the behavior. Though procedural languages such as C do not inherently support encapsulation, it can be achieved. For example .h files would contain only the function signatures while the .c files would have the implementation details. An encapsulated abstract data type can also be achieved in C as shown by Blustein [99].

3.7 Generality

In the software reuse literature, generality is one of the most preferred design principles and properties for reusable components. Coleman et al. [71] described generalization as the process of abstracting the commonalities and stripping away the differences (i.e. ignoring the details of how, when, where, and the constraints). Generality has been endorsed as an important principle for designing reusable components in the reuse literature [16, 17, 69, 70, 72, 73, 84, 86-88].

Weide et al [16] defined generality as a property of the specifications of a component that are sufficiently abstract and not too restrictive to allow a variety of implementations. Matsumoto [88] described generality as the extent to which a person who does not know how a software module was developed can understand the module's objects, and the relationships between its objects and algorithms. Ezran et al. [17] specified generality as a functional property of software components; the generality of the components must be implemented with a trade-off between being too specific (less reusable – being specific means the component could be used only in specific scenarios) and being too generic (less valuable – requires more effort in reusing if the component is too generic because the component must be modified to be used in different environments).

Karlsson [87] described generality of the functionality of components as the first and most important factor for developing reusable components. Karlsson presented generality as the "criteria that assess a component's ability to expand the usefulness of a given function beyond the existing module or program and its present scope. [87]" However, he also said that in the process of making components general, factors such as understandability, integration problems and performance issues must be taken into consideration. Karlsson [87] presented parameterization as a technique for achieving generality in software components.

Leinfuss [70] and Coleman et al. [71] have presented separation of concept from content as a means of achieving generality. Jacobson et al. [72] encouraged two techniques to achieve generality – 1. abstraction through reuse of design rather than code, and 2. encapsulation. Coleman et al. [71] also discussed analyzing the commonalities and variabilities in a domain to help identify the design features for improving generality in a reusable component. Braun [86] proposed handling of exceptions, in ADA programming, as a means for providing clarity of the component and as the most flexible way of handling unusual situations to achieve generality. For reusable components, exception handling may vary among different users and hence must be flexible. This means that alternative ways are provided for avoiding exceptions and providing a way for users to correct the problems. "An *exception* describes a situation that, if encountered, requires something exceptional to be done in order to resolve it. During program execution, an *exception occurrence* is a situation in which the standard computation cannot pursue. [100]"

Stroustrup also endorsed generality as a criterion to make a component reusable. According to Stroustrup, "*…a component must represent a general concept. It doesn't matter how elegantly a component is represented or how thoroughly it is documented if it represents the solution to a single particular problem only.*" [73]

3.8 Genericity

Languages such as `Ada`, `Clu`, and `Eiffel` allow modules to have generic parameters that represent types or in other words allow type independence. With genericity, the implementer may write a single module for all instances of the same implementation of a data abstraction to various types of objects [101]. According to Coulange [57],"Genericity is the capacity for creating a package or an object class whose types are not completely defined." They maybe static (if defined before compile and run time) or dynamic (if defined during compile or run time).

According to Meyer [102], genericity allows a module to be defined with generic parameters that represent types. This is a definite aid to reusability because just one generic module is defined, instead of a group of modules that differ only in the types of objects they manipulate. Genericity is supported in `C++` in as parameterized class (class template) as introduced by Stroustrup [73, 103]. `Eiffel` [104] also offers parameterized classes for achieving type independence. The concept of generics in `C#` and `Java` also help achieve genericity. For example, according to Ghosh [105],"In Java Generics, type requirements can be defined on arguments as a set of formal abstractions – this feature is called constrained genericity. The generic types of the classes have to honor these requirements in order to participate in the valid instantiation. Java Generics use interfaces to represent a concept and employ the mechanism of subtyping to model a concept."

3.9 Isolate Context and Policy

The 3Cs model of reuse design [61], for example, was developed to explore the reuse design process in a general framework. It specifies three levels of design abstraction: Concept – representation of the abstract semantics, Content – implementation details of the code or software, Context –environment required to use the component. According to Sloman [106], "In an object oriented approach, the external behavior of an object defines how it interacts with other objects in its environment. We refine the concept of policy to be the information which influences the interactions between a subject and a target and so the policy specifies a relationship between the subject and target. Multiple policies may apply to any object as it may be the subject or target of many policies." Policies are similar to concerns defined for Aspect-Oriented Programming (AOP). According to Elrad et al. [107], "AOP is based on the idea that computer systems are better programmed by separately specifying the various *concerns* (properties or areas of interest) of a system and some description of their relationships, and then relying on mechanisms in the underlying AOP environment to weave or compose them together into a coherent program. *Concerns* can range from high level notions like security and quality of service to low-level notions such as caching and buffering. They can be functional, like features or business rules, or nonfunctional (systemic), such as synchronization and transaction management."

The policies followed for the component such as security and safety must be isolated and separated from the context (operating environment) to encourage reuse in various domains. Ezran et al. [17] had provided some technical criteria for improving reusability of components. One such criterion is the security of the components. It was suggested that, especially for black box reuse, the reuser must be able to control the origin of the asset (digital signatures), and the asset's access to the private resources, and that they should be independent of the environment. This will promote portability as well. Components could be used as plug and play components across various domains. According to Aoyama [108], plug and play is an approach in component based software engineering (CBSE) defined as: "Component should be able to plug and play with other components and/or frameworks so that component can be composed at run-time without compilation." Ezran et al. [17] said component-based technologies such as `ActiveX` [109] or `Java Beans` [110] help achieve this.

3.10 Linking of Test to Code

Code may also be written to implement the test cases for the component part. Programmers generally would like to test code before reusing and such a design of linking test to code may encourage reuse. Kent Beck, the creator of Extreme Programming (XP), was a pioneer in promoting code-driven testing frameworks. He had originally implemented such frameworks for `SmallTalk` known as the `SUnit` [111]. This was later ported to `Java` as the `JUnit`. `JUnit` [http://www.junit.org/] is a unit testing framework in the `Java` programming language that can link test to code. Many modern languages and frameworks also support such unit testing techniques like `CppUnit` for `C++` [http://cppunit.sourceforge.net/], `NUnit` for .`Net` framework [http://www.nunit.org/], `HUnit` for `Haskell` [http://hunit.sourceforge.net/] etc.

Brenner et al. [112] have also recently proposed an approach (MORABIT – Mobile Resource Adaptive Built-In Test) to build test cases into components to automatically check the environment and thus reduce verification effort. Their work is based on the notion of "Built-In Test (BIT) —tests that are packaged and distributed with prefabricated, off-the-shelf components—the approach partially automates the testing process, thereby reducing the level of effort needed to establish the acceptability of the system. [112]." An earlier work based on BIT explicitly focusing on run-time verification was `Component+` [113, 114].

Test-driven development (TDD) is a major feature of Extreme Programming where test-first is an important programming concept i.e. the programmer needs to write automated test cases to test the components before they are built. According to Kent Beck, TDD encourages simple designs and inspires confidence [115].

3.11 Modularization

Sametinger [15] provides modularity as a desired characteristic for software components. Modularity is, "A component should be logically partitioned into subcomponents that perform specific functions. [15]" Leach [78] strongly suggests using object-oriented design and object-oriented programming features of a language such as `Ada` and `C++` to improve modularity which in turn increases the reusability of the component. Coulange [57] proposed that the object-oriented programming approach and use of objects as the best way to achieve modularity and improve reusability. Ezran et al. [17] and Meyer [102] also endorse modularity as a general criterion for improving quality and reusability.

Sodhi et al. [76] also endorsed modularization as an important principle for the object-oriented approach to promote software reuse. They suggested that the solution space definition for the program must be broken into smaller modules. The modules must be grouped around a data type and objects of that data type. They said that in a well-modularized software system, modules at the upper-level must be more abstract while the lower-level modules must be more detailed. Good modularity is achieved by loose coupling between modules i.e. the dependence between the modules is as minimal as possible. Compared to composition, where more than one component with individual functionalities are developed and composed into a single component, modularization divides the functionalities of a component into smaller modules. For example, in `C` more than one function can be written within a single `.c` file.

3.12 One Component Use Many Helper Components

A component created for reuse may be built using many reusable components say from a library. When a component is built using other components, then the whole family of components should be considered as a single component [15]. For example, if a component written in `C` uses component from a standard `C` library, then the written component combined with the library should be treated as one component. Compared to composition, where more than one component is developed and combined into a single component, one component use many helper components is achieved when already existing third-party helper components are used. For example, Microsoft provides data access components [116]: “These components abstract the logic required to access the underlying data stores. Most data access tasks require common logic that can be extracted and implemented in separate reusable helper components or a suitable support framework. This can reduce the complexity of the data access components and centralize the logic, which simplifies maintenance. Other tasks that are common across data layer components, and not specific to any set of components, may be implemented as separate utility components. Helper and utility components are often encapsulated in a library or framework so that they can easily be reused in other applications."

3.13 Optimization

According to McConnell [117], the pareto principle certainly applies for program optimization i.e. 80 percent of the can be achieved with 20 percent of effort. Hence, he suggests that optimization is simply more appealing than anything else. As early as 1971, Knuth [118] had reported that for a set of 24 programs half of the running time was due to only 4% of the programs. In a study of the execution times of a program's routines, Boehm [119] reported that 80 percent of the execution time was due to 20 percent of the routines. Bentley [120] presented a study of a 1000-line program which spent 80 percent of its time in a five-line square-root routine. When the speed of the square-root routine was tripled, the execution time of the whole program was halved.

In general, components built for reuse are usually slower than their equivalent reusable components. Organizations are more likely to use code that meets the quality standards of the organization. As a rule of thumb, if the reusable component is slower by more than 25%, it will not be used [121]. So, optimization techniques such as profiling using profilers (profilers are language-dependent) would encourage reuse of the components. A study [122] was also conducted to understand how a software profiler (`gprof` [123, 124]) can be used to help design, evaluate, and index reusable components. Some programming techniques mentioned by Bentley [125], which are language-independent, would also encourage reuse of the components. For example, some techniques discussed by Bentley [125] include storing pre-computed results to reduce the cost of re-computing an expensive function, cache frequently needed data to reduce search and access time, do lazy evaluation (the policy of not computing a result until it is needed) whenever possible, etc. However, optimizations must be done carefully, since increased optimization often decreases code readability and maintainability [121]. Optimization techniques may also be employed for efficiently using the space (memory) available. For example, dynamic generics help us to define data types as required during run-time and save space when types with larger space requirements are not needed.

One of the four criteria, given by Stroustrup [73], is *acceptable efficiency,* which can be achieved through proper optimization of the code. According to Stroustrup, the importance of efficiency varies enormously. However, run-time or space overheads can be critical, and even where they are not, obvious overhead tempts the designer and programmer to do better with a special purpose solution.

According to McConnell [117], the pareto principle certainly applies for program optimization i.e. 80 percent of the can be achieved with 20 percent of effort. Hence, he suggests that optimization is simply more appealing than anything else. As early as 1971, Knuth [118] had reported that for a set of 24 programs half of the running time was due to only 4% of the programs. In a study of the execution times of a program's routines, Boehm [119] reported that 80 percent of the execution time was due to 20 percent of the routines. Bentley [120] presented a study of a 1000-line program which spent 80 percent of its time in a five-line square-root routine. When the speed of the square-root routine was tripled, the execution time of the whole program was halved.

3.14 Parameterization

According to Lamping [126], "A system is parameterized when it has one or more external inputs which partially determine a result." "Parameterization provides a controlled way of customizing a generalized component when it is reused by substituting in an allowed range of values for the parameters which are embedded "place holders" for the differences in the component" [69]. Karlsson [87] defines isolation of components through parameterization as a desirable characteristic for reusability. Different requirements can be isolated to a small part of the system, and the rest of the system constructed relatively independently of whatever specification is chosen. Parameterization is treated as a special case of isolation where some requirements could be realized through parameters. McClure [69] presented parameterization as the most common technique to abstract the commonalities and strip away differences of the functionalities of the component to promote reuse.

3.15 Restrictiveness

Restrictiveness is one of the important general properties of good reusable component designs [16, 84]. Weide et al. provided a general guideline for reusable components as, “State everything about the behavior that is expected of a correct implementation—and nothing more (“restrictiveness”) [16].” For example, consider a component that has a functionality of performing certain operations on only the string data type. The component could be restricted to accept only the string data type, not other types such as integers or floating point numbers. This is contrary to genericity but the component requires only one data type and a trade-off is made with genericity (type independence).

3.16 Self-documenting Code

Frakes et al [127] presented internal program documentation in two forms: self-documenting code and program comments. They argued that self-documenting code is better than code that relies on program comments. This is because self-documenting code requires less reading. Also, the comments may not be updated when the code is updated, but this cannot occur with self-documenting code. They provide some guidelines for self-documenting code to improve program readability for `C`, but they can be easily followed in other languages as well:

- Use of good names: use meaningful and good names for variables, functions, types, macros, constants etc. The names should express pertinent information about the named object revealing things like the type of the object, the origin of the object, and the role of the object within the function, program or system.
- Use of right types: this is important in type-rich languages (languages with many built-in types and mechanisms to create new types) such as `C`.
- Use of right control structure: empirical studies have shown that good control structure improves program readability [128]
- Display of program structure: the structure of the program should enable users to figure out the execution path easily and this improves program readability. Techniques like formatting and proper indentation may be used.

McConnell [117] strongly suggests the use of self-documenting code to improve readability and understandability which in turn improves the reusability of the components. According to McConnell, source code is its own best documentation. The source code should not be so bad that it requires extensive documentation; the source code must always be improved such a way that the external documentation required is minimal. Commenting should be only for that code which cannot say about itself. Commenting must be minimal, because if done poorly, it is a waste of time and potentially harmful.

Raskin [129] was also of the opinion that in-line comments must be avoided and self-documenting code is preferred instead. Self-documenting code is generally referred to be as clear and understandable as possible. Instead of using `n` or `count`, `numberOfApplesPicked` is used. However, Raskin also argues that self-documenting code is not sufficient because it cannot always explain why the program is written and the rationale for choosing the particular method used in the program.

McConnell [117] also endorsed using techniques to mark different kinds of comments differently. For example in `C++` provides `@keyword` indicating key words, `@param` indicating a parameter to a routine, `@version` indicating file-version information, `@throws` indicating the exceptions thrown by the routine and so on. This way a user can just search for all the `@throws` to retrieve the documentation on the exceptions in a program. McConnell provided the following check list for self-documenting code [117]:

CHECKLIST: Self-Documenting Code

Classes

❑ Does the class's interface present a consistent abstraction?

❑ Is the class well named, and does its name describe its central purpose?

❑ Does the class's interface make obvious how you should use the class?

❑ Is the class's interface abstract enough that you don't have to think about how its services are implemented? Can you treat the class as a black box?

Routines

❑ Does each routine's name describe exactly what the routine does?

❑ Does each routine perform one well-defined task?

❑ Have all parts of each routine that would benefit from being put into their own routines been put into their own routines?

❑ Is each routine's interface obvious and clear?

Data Names

❑ Are type names descriptive enough to help document data declarations?

❑ Are variables named well?

❑ Are variables used only for the purpose for which they're named?

❑ Are loop counters given more informative names than i, j, and k?

❑ Are well-named enumerated types used instead of makeshift flags or boolean variables?

❑ Are named constants used instead of magic numbers or magic strings?

❑ Do naming conventions distinguish among type names, enumerated types, named constants, local variables, class variables, and global variables?

Data Organization

❑ Are extra variables used for clarity when needed?

❑ Are references to variables close together?

❑ Are data types simple so that they minimize complexity?

❑ Is complicated data accessed through abstract access routines (abstract data types)?

Control

❑ Is the nominal path through the code clear?

❑ Are related statements grouped together?

❑ Have relatively independent groups of statements been packaged into their own routines?

❑ Does the normal case follow the if rather than the else?

❑ Are control structures simple so that they minimize complexity?

❑ Does each loop perform one and only one function, as a well-defined routine would?

❑ Is nesting minimized?

❑ Have boolean expressions been simplified by using additional boolean variables, boolean functions, and decision tables?

Layout

❑ Does the program's layout show its logical structure?

Design

❑ Is the code straightforward, and does it avoid cleverness?

❑ Are implementation details hidden as much as possible?

❑ Is the program written in terms of the problem domain as much as possible rather than in terms of computer-science or programming-language structures?

3.17 Separation of Concepts from Content

In the 3Cs model by Latour et. al [61], concepts refer to the representation of the abstract semantics while content represent the implementation details of the component. For example in `C`, the header files can have the declarations of the functions representing the concepts while the actual code that implement these functions can be in a separate `.c` file.

In the object-oriented paradigm, inheritance can be a way of achieving the separation of concepts and content. The methods in the parent class will represent the concepts while the child classes inherited from the parent class will have the code that implements the methods in the parent class. McClure [69] recommends using encapsulation to separate the application logic and implementation logic; the internal representation of the class is hidden from the users of the class to restrict the users of the class to its interface only. Jacobson et al [98] recommends exposing only the interface of an object to an user to aid reuse of the class. According to Coleman et al. [71], one of the original motivations of the object-oriented approach is to promote reuse by separating the interface of an object from its implementation. This can be achieved, for example in C++, by using abstract classes to provide the interface and subclasses of the abstract class to provide the implementation. An abstract class is a reusable object-oriented design for a component. It specifies the interface of a class and the tree of subclasses that can be derived from it. Abstract classes fully specify behavior, not implementation. They cannot be instantiated, only subclassed from [71]. According to Stroustrup [130], a class is an abstract class if it has one or more virtual functions, and no objects can be instantiated from that class. Abstract classes can only be used as interfaces and to create other classes.

According to Coleman et al. [71] encapsulation aids reuse by encouraging clients to depend on the interface an object provides while being shielded from modifications to its implementation and from its interactions with other parts of a system. Encapsulation thus minimizes the exposures of clients to changes in implementation and frees them from being locked into a specific behavior.

3.18 Variability Mechanisms

A variability mechanism is a technique by which an existing content in a component can be customized or modified to be reused. “For the optimal reuse of software development artifacts so called variability mechanisms play a crucial role. Variability mechanisms allow for the derivation of artifact variants from generic artifacts. [131]” Such mechanisms and techniques are popular in product line and domain engineering where variation points (points are identified in a product line where variable implementations are possible) and variants (the variable implementations) are identified to implement the variability mechanism. Puhlmann et al. [131] provided a survey report on the general variability mechanisms that included information hiding, inheritance, parameterization, templates, null-classes, design patterns etc. They also discussed variability mechanisms specialized for UML activity diagrams, UML state machines, and for Business Process Modeling Notation (BPMN).

Martinez-Ruiz et al. [132] introduced such variability mechanisms into SPEM v2.0 (Software process Engineering Metamodel) by defining variability within the `MethodPlugin` package. It includes the abstract class: `VariabilityElement` and the enumeration type: `VariabilityType`. The `VaribilityType` enumeration defines the type of variability between both instances of the `VariabilityElement` class. It includes the `contributes`, `replaces`, `extends`, `extends-replaces`, and `na` (default) values:

- `Contributes` is a variability relationship that allows the addition of a `VariabilityElement` to another base, without altering its original contents. This relationship has transitive properties. A base element must have more than one contributor.
- `Replaces` is a variability mechanism which permits the `VariabilityElement` to be replaced by another one, without modifying its properties. A base element can only define a replaces relationship. Like contribution, the replaces relationship is transitive.
- `Extends` relationship is an inheritance mechanism between the `VariabilityElement.` This relationship is also transitive and both contribute and replace relationships take priority over extends.
- `Extends-replaces` relationship combines the effects of both previous relationships. So while the `replace` relationship replaces all the properties of the base element, this one only replaces those values which have been redefined in the substitute element.

3.19 Well-defined Interface

According to Karlsson [87], "the *interface* describes the boundary of the component i.e. what operations it offers, what parameters it takes, and what it demands from the environment…The distinction between the *interface* (the specification) and the body (the implementation) of a component plays an important part in the modularization of software, not only in object-oriented development, but also in more traditional paradigms.” A well-defined interface aids the reusability of software components. An interface determines how a component can be reused and interconnected with other components. If the component’s interface is simpler, it should be easier to reuse. There are three types of interfaces: application programming interface (API), user interface, and data interface [15]. According to Sametinger [15], APIs may be the most important type of interface for reuse. In reuse, a well-defined API can be used to integrate the application’s functionality into the new software system. APIs may be language dependent or independent. An example of language dependent APIs are the built-in APIs provided by the `.NET` framework for `C#.` An example of language independent APIs are the COM-component APIs for various languages in the `.NET` framework [http://www.microsoft.com/com/default.mspx]. A user interface may be command line or graphical (GUI). Data interfaces are used to facilitate data handling. They can be used to read input data, transform the data until it has reached its final form and write the output data. Leach [78] has strongly recommended use of standard interfaces as absolutely essential for software reuse. He argues that without standard interfaces, information hiding between modules cannot be enforced.

McClure [69] has also recommended developing standards in a project to specify interfaces that will increase project quality and general usefulness for improving reusability. Matsumoto [88] mentions that abstract data type packages, subroutines and functions with well-arranged parameters are good examples of clearly defined software modules. Hooper et al. [101] promote reuse by means of interface abstraction i.e. use of the interface does not require knowledge of the implementation; `SmallTalk-80` is one language which supports this. McClure [69] defines self-descriptiveness as a technical criterion for good reusability of components; self-descriptiveness means a well-defined interface where the interface is well described with a usage protocol and help the user easier to understand and use the component as a black-box reuse. Meyers [133] provided ‘*the most important*’ general interface guideline:

‘Make interfaces easy to use correctly and hard to use incorrectly.’

Meyers [133] provided two aspects to designing interfaces that obey the guideline: First, interface designers must train themselves to try to imagine all (reasonable) ways in which their interfaces could be used incorrectly. Second, they must find ways to prevent such errors from occurring. Consider a (C++) class for representing dates in time and how its constructor might be declared:

```
class Date {
     public:
          explicit Date(int month,
                         int day,
                         int year);
};
```

This is a classic example of an interface that's easy to use incorrectly. This is because all three parameters are the same type. Users of this function can easily mix up the order - an error that is especially likely given that people from different cultures and countries use different ordering conventions for a date's month, day, and year. Furthermore, the interface will also allow nonsense data to be passed in. For example, negative numbers could be passed. Creating separate types for days, months, and years can eliminate the ordering errors, and creating a fixed set of immutable Month objects can essentially eliminate the possibility of specifying invalid months. An example of this approach is given below:

```
struct Day { int d; };        // thin wrappers for Day and
Year
struct Year { int y; };

class Month {
public:
     static const Month Jan;       // a fixed set of immutable
     static const Month Feb;       // Month objects
     ...
     static const Month Dec;
     private:
     explicit Month(int);
};

class Date {                  // revised (safer, more
flexible)
public:
     explicit Date(Day d, Month m, Year y); // interface
     explicit Date(Month m, Day d, Year y);
     explicit Date(Year y, Month m, Day d);
     ...
};
```

Perhaps the most widely applicable approach to preventing errors is to define new types for use in the interface, in this case, **`Day, Month,`** and **`Year.`** It's best if such types exhibit the usual characteristics of good type design, including proper encapsulation and well-designed interfaces, but this example demonstrates that even introducing thin wrappers such as **`Day`** and **`Year`** can prevent some kinds of errors in date specification. A second commonly useful approach to preventing errors is to eliminate the possibility of clients creating invalid values. This approach applies when we know the universe of possible values in advance.

Forcing users of an interface to choose from a set of guaranteed-valid choices is also a good design. Most websites now offer a user interface with a drop-down box or calendar to choose a date. This way a user is forced to choose only a valid date and cannot choose an incorrect date. Meyers, thus, suggested that '*responsibility for interface usage errors belongs to the interface designer, not the interface user*.'

Stroustrup had also put forth that a clean interface to users is a criterion to make a component reusable. According to Stroustrup, *"…the more complicated an interface is, the more work it is to use the component and the more attractive it becomes to building something specialized instead.*[73]*"*.

A summary of the cross-reference between the reuse design principles and the reference literature is given in Table 5.

Table 5. Cross-reference between the reuse design principles and the literature

Reuse Design Principle	References																									
	Frakes and Lea [14]	Sametinger [15]	Weide et al. [16]	Ezran et al. [17]	McClure [69]	Coleman et al. [71]	Jacobson et al. [72]	Stroustrup [73]	Sodhi et al. [76]	Krueger [77]	Leach [78]	Parnas [81]	Braun [86]	Karlsson [87]	Liskov et al. [84]	Matsumoto [88]	Jacobson et al. [98]	Hooper et al. [101]	Meyer [102]	Stroustrup [103]	Beck [111, 115]	Bentley [125]	McConnell [117]	Raskin [129]	Martinez-Ruiz [132]	Meyers [133]
Abstraction							X		X	X	X	X														
Clarity and Understandability			X	X									X	X	X	X										
Commonality and Variability	X								X									X								
Composition	X			X																						
Documentation	X	X	X	X	X						X		X	X	X	X										
Encapsulation						X			X			X					X									
Generality			X	X	X	X	X	X					X	X	X	X										
Genericity								X										X	X	X						
Isolate context from policy	X			X																						
Linking test to code	X																				X					
Modularization				X					X		X								X							
One component use many		X																								
Optimization	X																					X				
Parameterization	X				X									X												
Restrictiveness			X												X											
Self-documenting code																							X	X		
Separate concept from content	X				X	X											X									
Variability Mechanism																									X	
Well-defined interface	X	X			X			X			X					X		X		X						X

Chapter 4: Building and Designing *for* Reuse

For practitioners and researchers, there are two motivations in the study described in this Chapter. One is that even though the relation between software quality and reuse has been established, very few studies have been found comparing one-use and equivalent reusable components. One such preliminary study was conducted by Frakes and Tortorella [19]. The other motivation is that practitioners and researchers need to address the problem of how to build reusable components. This exploratory study used a comprehensive list of reuse design principles presented in the past two decades for software reuse and identified the most used reuse design principles. This can be a guideline for building reusable components. One major limitation of the study in his Chapter is that the components studied are small in size which may affect generalizability. This study is an exploratory study with a good sample size of 107 subjects, nearly all of whom have some experience in software engineering. The sample size is adequate for comparing one-use and reusable components. Also, this exploratory study is a baseline for future study on designing, building, and measuring reusable components.

As discussed earlier in Chapter 1, one study that is similar to this study is presented by Seepold and Kunzmann [20] for components written in VHDL (Very-high-speed integrated circuits Hardware Description Language). However, the major limitation in that study was that it involved only four components - two one-use and two equivalent reusable components. According to that study the complexity, effort and productivity were all higher for reusable components. The reasons identified were due to overhead in domain analysis, component verification, and documentation.

4.1 Hypotheses

We revisit, from Chapter 1, the four hypotheses related to reusable components (Hypotheses Ia-d). Due to the higher complexity and functionality of the reusable components, their size (in SLOC - source lines of code), effort (in hours), the productivity (in source lines of code per hour), and number of parameters should be significantly higher than their equivalent one-use components. These hypotheses are summarized in equations (1), (2), (3), and (4). $SLOC_{Reuse}$ is the actual source lines of code in the reusable component while $SLOC_{ReuseDiff/hour}$ is the difference in the source lines of code between the reusable and one-use components. The difference is considered for the productivity of reusable components because the reusable components studied in this paper were not built from scratch; instead, they were reengineered by modifying the one-use components.

Hypothesis I-a: A reusable component is larger than its equivalent one-use component.

$$SLOC_{Reuse} > SLOC_{one\text{-}use} \tag{1}$$

Hypothesis I-b: A reusable component requires larger effort than its equivalent one-use component.

$$Effort_{Reuse} > Effort_{one\text{-}use} \tag{2}$$

Hypothesis I-c: When designing and building a reusable component, the developer is more productive (in number of SLOC written per unit time) than when the developer designs and builds an equivalent one-use component.

$$SLOC_{ReuseDiff/hour} > SLOC_{one\text{-}use/hour} \tag{3}$$

Hypothesis I-d: A reusable component has more parameters than its equivalent one-use component.

$$Parameters_{Reuse} > Parameters_{one\text{-}use} \tag{4}$$

4.2 Method

Based on the faceted classification of types of software reuse by Frakes and Terry [2], the reuse design in this study involves development scope as internal, modification as white box, domain scope as vertical, management as ad hoc, and reused entity as code.

A total of 107 subjects participated in this study. Nearly all the subjects were technical professionals with at least some experience in software engineering. The subjects were given an assignment to build a one-use software component implementing the s-stemming algorithm [13] and were later asked to convert their one-use stemmer component to a reusable component. The subjects were students either at Master's or Ph.D. level at Virginia Tech, U.S.

4.2.1 A S-Stemmer Component

Three rules specify the s-stemming algorithm as follows (only the first applicable rule is used) [134]:

If a word ends in "ies" but not "eies" or "aies" then Change the "ies" to "y",
  For example, cities → city
Else, If a word ends in "es" but not "aes", "ees", or "oes" then change "es" to "e"
  For example, rates → rate
Else, If a word ends in "s", but not "us" or "ss" then Remove the "s".
  For example, lions → lion

The subjects were given lectures on the topics of software reuse, domain engineering and reuse design principles. The mindmap of the reuse design process as given in Figure 1 (Chapter 3) was the basis of the lecture. One hundred and one of them converted their one-use components to an equivalent reusable component based on the reuse design principles in Figure 1. The reuse design process followed was the reengineering method and not from the scratch method, i.e. an existing component was modified to be reusable. The subjects were asked to follow a 'for' reuse design process i.e. design for future use.

The programming language used was Java. The reusable components were compared with one-use components based on the size (SLOC), effort (time in hours), number of parameters, and productivity (SLOC/hr).

4.2.2 Data Collection

For both the one-use and reusable components the subjects were asked to report the time required for developing the component. For the reusable components, the subjects were asked to indicate and justify the reuse design principles (from Figure 1) that they used. The subjects also reported the reuse design principles they considered but did not apply to the reusable components. They also provided feedback on why they did not use those principles.

One hundred and seven students successfully built the one-use components and 101 built the equivalent reusable component; six subjects did not build the equivalent reusable component. Three of the 101 who submitted did not report the time required for building the component. All the components, both one-use and reusable were graded as part of the assignment and required to satisfy on the basis of two criteria: (1) the components must compile and execute error-free, and (2) the components must provide the right solutions for a set of test cases. The grader also verified whether the reuse design principles the subjects claimed to use were applied.

4.2.3 Evaluation Metrics

Source lines of code or SLOC is one of the first and most used software metrics for measuring size and complexity, and estimating cost. According to a survey by Boehm et al.[135], most cost estimation models were based directly on size measured in SLOC. Some of them are COCOMO (Constructive Cost Model) [136], COCOMO II [21], SLIM (Software Lifecycle Management) [137], and SEER (System Evaluation and Estimation of Resources) [138]. In COCOMO and COCOMO II the effort is calculated in man-hours while the productivity is measured in SLOC written per hour. Many empirical studies have also been based on measuring the complexity of software components by measuring SLOC [139-142]. There are also empirical studies where productivity of software components is measured in SLOC/hr [139, 140, 142, 143].

Herraiz et al. [144] studied the correlation between SLOC and many complexity measures such as McCabe's cyclomatic complexity [145] and Halstead's metrics as given in [146]. In their study they have presented empirical evaluations showing that SLOC is a direct measure of complexity, the only exception being header files which showed a low correlation with the McCabe's cyclomatic complexity measures. Research by Graylin et al. presented evidence that SLOC and cyclomatic complexity have a stable nearly perfect linear relationship that holds across programmers, languages, code paradigms (procedural versus object-oriented), and software processes [147]. Linear models have been developed relating SLOC and cyclomatic complexity. Buse et al [148], for example, presented a study where they show a high direct correlation between the SLOC and the structural complexity of the code.

A study by Gaffney [149] reported that the number of faults in a software component is directly correlated to source lines of code (SLOC). Krishnan et al. [150] also reported an empirical study that showed a direct correlation between SLOC and the number of defects in software components.

Based on these studies, comparison between the one-use and reusable components are done based on the size (in SLOC), effort (man-hours), and productivity (in SLOC/hr).

4.3 Results and Analysis

4.3.1 Demographics

Twenty three subjects answered a questionnaire (see Appendix B for the survey) on their demographics. The questionnaire was optional. Sixteen of the respondents had a highest qualification of an undergraduate degree while 7 of them had completed a master's degree and enrolled in their second master's or doctoral program.

The experiences of the subjects in software engineering and programming are shown in Figure 2. Almost two-thirds had 4 or more years of experience in software engineering. About three-fourths (74%) of the subjects had 4 or more years of programming experience. About half (47.8%) had more than 8 years of programming experience. All of the subjects had at least some experience in software programming.

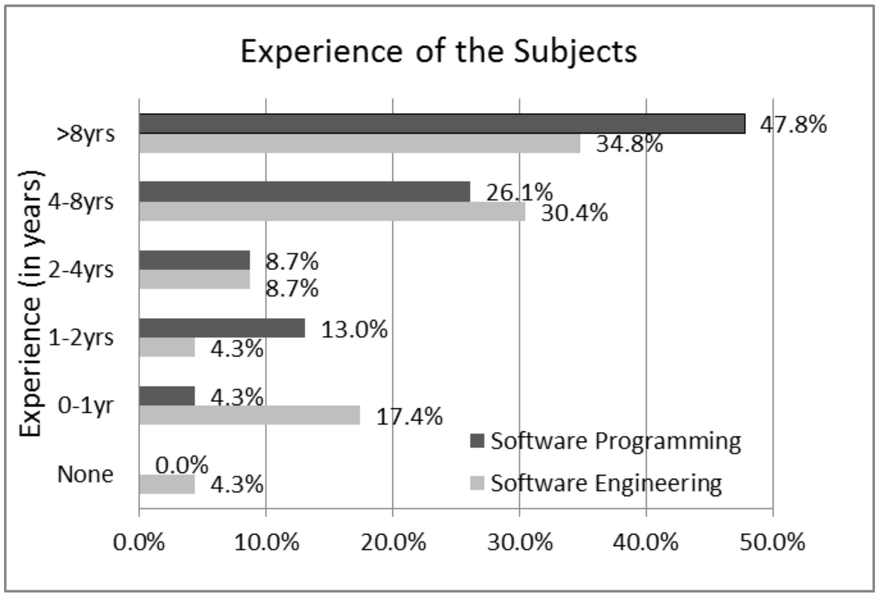


Figure 2. Experience of the subjects in software engineering and programming

The distribution of the roles of the subjects in their respective organizations is shown in Figure 3. More than two-fifths (43.5%) of the subjects held the primary role in the field of software programming as developers/programmers. Only 4.3% of the subjects had a managerial role. About one-third (34.8%) of the subjects were either a systems engineer or a systems architect. Of those who answered 'other', one was a student, one a program analyst, and one described their role as a senior software engineer.

All the subjects developed their components in Java. The experience of the subjects in the programming language is shown in Figure 4. No subject had zero experience with programming in Java. Less than 40% had very little experience (0-2 years) in Java. More than one-fifth (21.7%) had high experience (more than 4 years).

The subjects also gave their background experience with software reuse. About two-thirds (65.2%) of the subjects did not have any software reuse program in their organizations. The distribution the experience in the field of software reuse is shown in Figure 5. Almost half of the subjects (47.7%) had no or little (0-2 years) experience in software reuse. Only 13% had very high experience (more than 8 years) in software reuse.

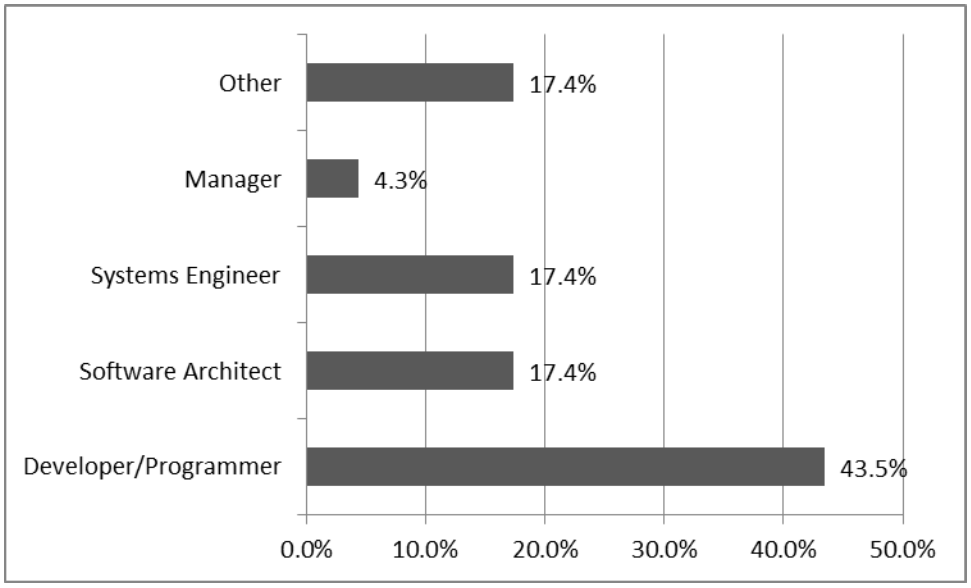


Figure 3. Distribution of subject professional roles

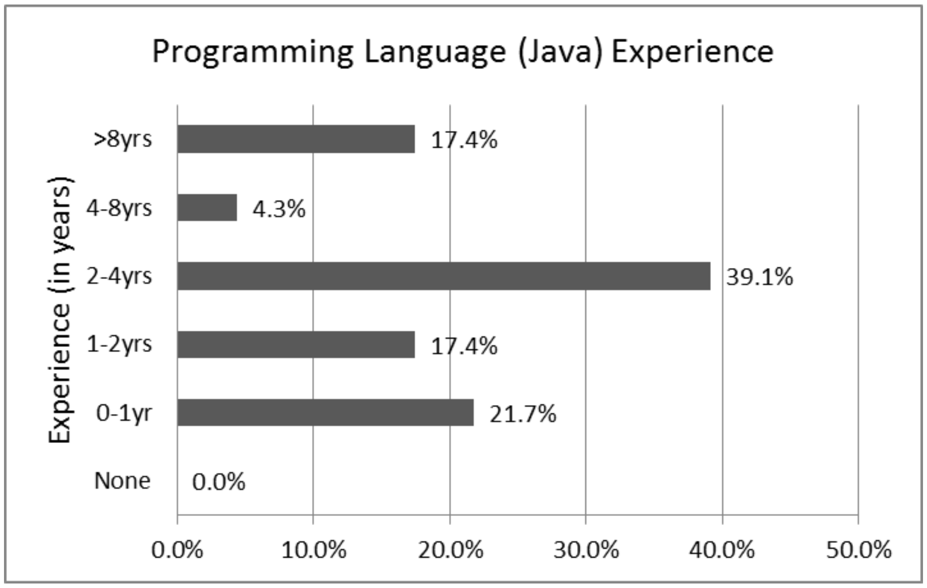


Figure 4. Experience of the subjects in Java

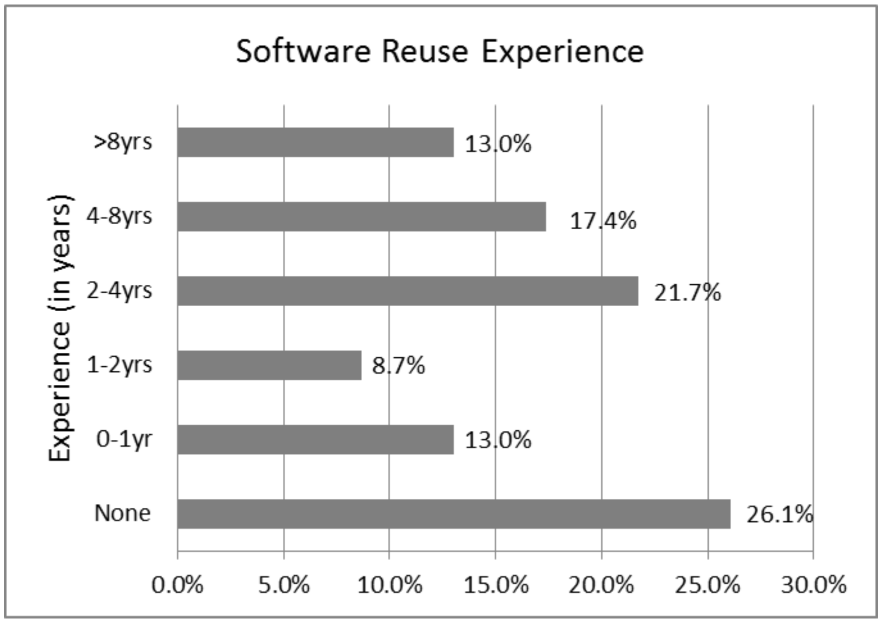


Figure 5. Experience of the subjects in the field of software reuse

4.3.2 Reuse Design Principles

Table 6 shows the summary of the usage of reuse design principles by the subjects. A Pareto ranking of the design principles by frequency is shown in Figure 6. The Pareto chart shows that 80% of the reuse design principles used were from the top eight ranked principles (Principle Rank#1-8). Figure 7 shows the distribution of the number of reuse design principles used by a subject. The mean number of principles used by the subjects was 3.4 and the median was 2. The distribution as seen in Figure 7 is unimodal and positively skewed. This is probably an indication that the subjects preferred to use the minimal number of reuse design principles. The range for the number of principles used was from 1 to 11, with 29 subjects using the minimum number and 5 using the maximum number. Eighty percent of the subjects used 5 or fewer reuse design principles.

A well-defined interface (#1) was the most used principle and was used for about half of the reusable components. Documentation was the second most used, in about 42% of the reusable components. Documentation has always been recommended and widely used in the programming world. Clarity and understandability of the code was the next most used. This principle allows the users of the component a better and easier way of comprehending the code for future use. The next three most frequently used principles were generality, separate concepts from contents, and commonality and variability analysis.

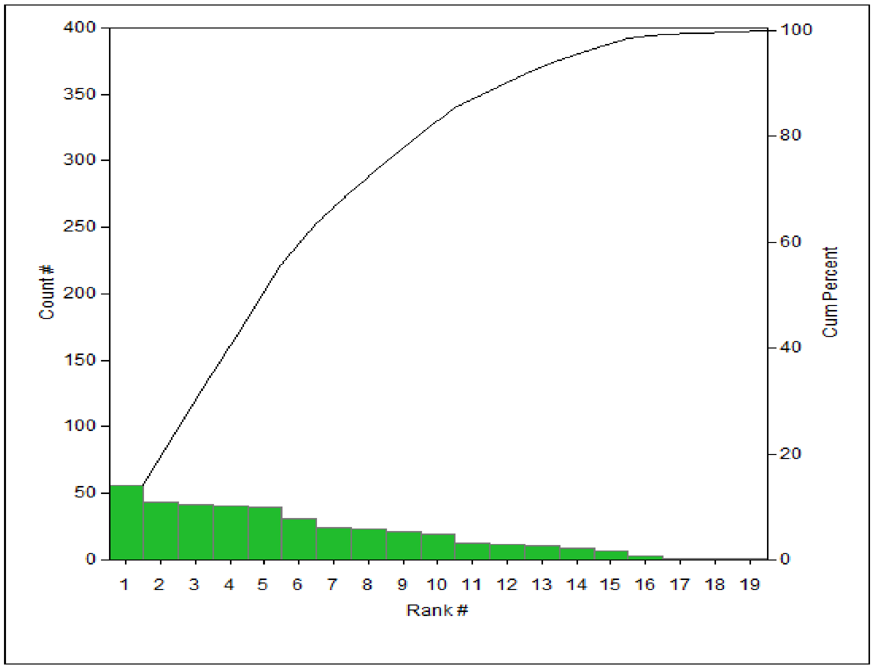


Figure 6. Pareto ranking of the reuse design principles

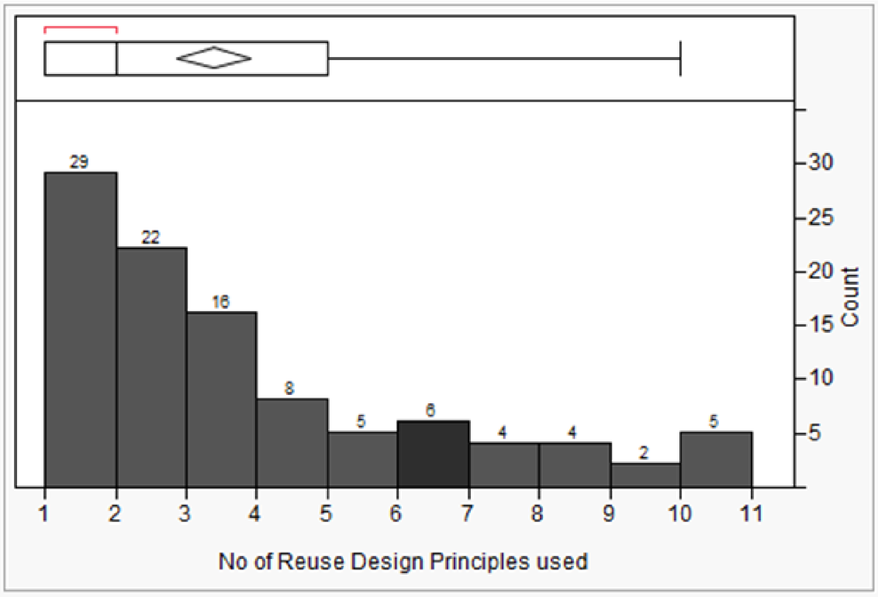


Figure 7. Distribution of the number of reuse design principles used

Table 6. Ranking of reuse design principles used

Rank#	Reuse Design Principle	Count#
1	well defined interface	56
2	documentation	43
3	clear and understandable	42
4	generality	41
5	separate concept from contents	40
6	commonality and variability	31
7	linking of test to code	24
8	encapsulation	23
9	one component use many	21
10	composition	19
11	variability mechanism	13
12	parameterization	12
13	genericity	11
14	optimization	9
15	restrictiveness	7
16	modification	3
17	isolate context and policy	1
18	abstraction	1
19	self-documenting code	1

4.3.3 Content Analysis[+]

Krippendorf [151] defines content analysis as "a research technique for making replicative and valid inferences from data to their context." Content analysis has been used as a qualitative

[+] The responses of the subjects are presented verbatim in double quotes; the words or phrases within the square brackets were not part of the subjects' responses but have been added to improve the understanding. Also, the errors in the spelling of some words in the responses have been corrected.

data analysis technique in software engineering research. For example, Niazi et al. [152] conducted interviews with software process improvement (SPI) practitioners and performed content analysis on the interview transcripts. They identified the critical success factors for implementing SPI. They followed a process similar to that followed by Badoo [153] where one seeks to identify the frequencies of occurrence of category issues. Baddoo et al. [154, 155] used the broad principles of content analysis, as given by Krippendorf [151], to analyze the responses of software practitioners in focus group discussions. From the content analysis they developed categories for motivators and de-motivators of implementing software process improvement in organizations.

In this study, a similar process is followed. The subjects in this study provided responses on the reuse design principles they used and why they used them. Content analysis on these responses was done in 3 stages:

- Categorization: The responses were categorized based on the reuse design principles. For example, the responses for why well-defined interface was used were grouped together into one category.
- Coding: The responses within a category were then interpreted and all the different reasons were identified. Each reason was assigned a code. For example, there were 4 reasons identified for well-defined interface. They were coded as ES (Component will become easier and simpler to understand), AM (Accommodate multiple future implementations), PR (Promotes Reuse), and DI (Discourage looking at the implementation details).
- Frequency Analysis: Each response within a category was interpreted for the reasons and assigned the respective codes. The frequency of each code was then calculated as the count for the respective reason.

The reasons for the reuse design principles are summarized in Table 7 and the counts of the reasons are given in parentheses. The subjects also provided any reuse design principles they considered but did not use, and why they did not use them. Content analysis was performed on these responses as well. They are summarized in Table 7.

4.3.3.1 Why the reuse design principles were used

The results of the content analysis are summarized in Table 7 with the reasons for using the reuse design principles, codes for the reasons, and counts for the reasons. Some of the subjects had given more than one reason and so some responses were assigned more than one code. Some of the respondents stated that the reuse design principle they used promoted the reuse of the component but did not specifically give a reason why or how it did so. These responses were coded as PR (promotes reuse). PR accounted for about one-fifth (19%) of the responses.

Table 7. Summary of content analysis - why the reuse design principles were used

Reuse Design Principle	Code* - why the principle was used (count)
well defined interface	ES - Component will become easier and simpler to understand (39) AM - Accommodate multiple types of future implementations (14) PR - Promotes Reuse (6) DI - Discourage looking at the implementation details (5)
documentation	JD - Javadocs (30) EI - External and internal documentation (23) IU - Improve understanding of the code and logic (22) AM - Accommodate future changes in the code (3) DH - Describe how to use the component (5) PR - Promote reuse (15)
clear and understandable	MU - Make the code and logic easy to understand (39) RL - Reduce the learning curve for using the component (1) IU - Increase the understanding of the component behavior (2) UD - Used Documentation (13) ND - No docs (2)
generality	HV - Handle variety of implementations (17) SO - Satisfy oracle hypothesis (16) AM - Accommodate future changes in specifications (12)
separate concept from contents	HI - Separate the implementation details (19) AM - Allows future modification to the content (11)

Reuse Design Principle	Code* - why the principle was used (count)
	PR - Promotes Reuse (13) UI - Using interface to separate (9) UH - Using Inheritance (5)
commonality and variability	AM - Accommodate future modifications easily (13) PR - Promotes Reuse (18)
linking of test to code	EW - Ensure the component is working properly (14) UB - Helps understand the component behavior (8) HT - Helps testing future modifications (3)
encapsulation	HI - Hide implementation details (15) PR - Promotes Reuse (8)
one component use many	DC - Simplify the component and decrease complexity (9) MF - To modularize functionality (4) PR - Promotes Reuse (8)
composition	IU - Improve understandability (11) AM - Easy to accommodate future changes (7) ER - Increases the ease of use of the component (3)
variability mechanism	AM - Allow easy future configurations (9) HI - Handle variety of implementations (4)
parameterization	IV - Increase variability (7) IN - Improve the interface (2) PR - Promotes Reuse (3)
genericity	MI - Allow multiple types of input (11)
optimization	LR - Runtime is reduced (3) PR - Promotes Reuse (6)
restrictiveness	CF - Ensure correct functioning of the component (7)
modification	AM - Allow modification of the stemming rules (3)
isolate context and policy	JS - Java supports many platforms (1)
abstraction	HI - Hide implementation details (1)

Reuse Design Principle	Code* - why the principle was used (count)
self-documenting code	IU - Improves understandability of the code (1)

*Code – two-letter code used to identify a reason (refer to section 5.3 for content analysis and coding)

The most common reason across the reuse design principles was to allow ease of changes a future user might want to implement. This reason has been coded as AM. This reason was stated in 8 of the reuse design principles: *well-defined interface* (14), *commonality and variability analysis* (13), *generality* (12), *separate concepts from contents* (11), *variability mechanisms* (9), *composition* (7), *documentation* (3), and *modification* (3). For example, a subject who used well-defined interface for allowing future modifications stated that, "This [well-defined interface] is critical to creating a reusable component. Such a component must be simple to use, yet configurable…If more complicated rules need to be created, it is possible to subclass the stemming rule and override the default behavior." Another subject who used generality and whose response was coded as AM stated that, "The design was changed drastically [compared to the one-use component] to allow a user to create their own rules that follow the same format given originally to apply to the word. This allows many more cases of potential use and thus increases reusability."

The second most stated reason is HI, found for 5 of the reuse design principles – *separate concepts from contents* (19), *encapsulation* (15), *well-defined interface* (5), *variability mechanisms* (4), and *abstraction* (1). HI refers to hiding the implementation details from the user of the component. Though the scope of the reusable components built was white box reuse (i.e. the code is available to the future users), the subjects argued that the users are more likely to reuse components if they are not exposed to the implementation details. The implementation details must be visited by a user only if necessary. Encapsulation is widely used in object-oriented programming for information hiding. Since, all the reusable components are in Java, which supports object-oriented programming, about two-thirds (65%) of the subjects used encapsulation to hide implementation details from the user. One subject who used encapsulation stated that "The usage of the encapsulation design principle allows for the business logic [i.e. the algorithm and implementation details] to be encapsulated or contained in one class and this is the ideal principle for reusability. If there are any implementation changes that are needed, those changes will not affect those using the interface."

Another reason that was most commonly stated was that the reuse design principle improved the understanding of the code and the logic, which would in turn encourage the reuse of the component. The subjects argued that the easier a component is to understand the higher the possibility that the component will be reused multiple times. This was coded as IU and was a reason for 4 of the reuse design principles – *documentation* (22), *clear and understandable* (39), *composition* (11), and *self-documenting code* (1). One subject whose response on documentation was coded as IU stated that, "Programmers are unlikely to reuse software that is not well documented or commented since it makes it harder to understand and maintain." Another subject whose response on clarity and understandability was coded as IU stated that, "Using this aspect [clarity and understandability] of reusable coding allows quick and easy navigation through the code. This also arranged the code for better understanding and to what the developer wanted to accomplish."

Well-defined interface is ranked first and four reasons were identified including PR. The most popular reason (70%) for using the principle was that a well-defined interface makes a component easier and simpler to understand which increases the component's reusability. This was coded as ES. For example one subject whose response for a well-defined interface was coded as ES stated that, "Clear, clean, simple interface facilitates component reuse by other components or programs." The second most popular reason for a well-defined interface was AM – allowing future modifications. Hiding the implementation details (HI) was also given as a reason by 5 of the subjects. Seven subjects had both AM and ES assigned to their responses. One subject reasoned for both HI and AM.

Documentation is ranked second and four reasons including PR were identified. More than two-thirds (70%) of the subjects who used documentation had used Javadocs (JD). Javadoc is a tool for generating API (Application Programming Interface) documentation in HTML (HyperText Markup Language) format from doc comments in source code [156]. Subjects also used external and internal documentation (EI). Ten subjects (23%) used both JD and EI. More than half of the subjects (51%) who used documentation argued that it helped to improve the understanding of the code and the logic (IU). One subject stated that, "In my experience, code that is well-documented is most likely to be incorporated in future software iterations, Countless projects in my agency have been abandoned [due lack of good documentation] because the previous author of the most elegant looking brilliant solution was also the only one who knew all the nuances behind the implementation." Three subjects used documentation to accommodate future changes (AM) while 5 subjects used documentation to describe how the component is to be used (DH).

Making the program *clear and understandable* was used by 42 subjects and is ranked third. Thirteen of those subjects (31%) used documentation to make the code more clear and understandable. Two subjects however argued that documentation must be minimal while the code must be clear and understandable by itself. For example, one of those subjects stated that, "…There should be little or no need for documentation; the code itself should suffice. Method names should be clear of their functions, and parameters indicative of their input and output." The majority (93%) of the subjects who used clarity and understandability reasoned that they used the reuse design principles to improve the understanding of the code and logic to improve reusability. One subject reasoned that good documentation reduces the learning curve for using the component and thus would increase the chances for reusing the component.

Generality is ranked 4th and used by 41 of the subjects. Three reasons were identified for applying generality. The top reason was to handle a variety of implementations (HI). The subjects argued that a component with more types of implementations were more likely to be reused. For example one subject stated that, "…supporting a variety of implementations makes reuse more plausible." The next most common reason was to satisfy the oracle hypothesis (SO) i.e. to predict the future uses of a component and design the component as generally as possible. Weiss [157] defines Oracle Hypothesis as, "It is possible to predict the types of changes that are likely to be needed to a system over its lifetime. In particular, the types of variations of a system that will be needed are predictable." For example one of the subjects stated that, "I tried to predict the future uses of the stemmer reusable asset, by providing multiple ways of invoking the stemmer component with overloaded methods." Twelve subjects used the principle of generality to accommodate future changes (AM). Four subjects had responses which were coded for both HI and AM.

Ranked 5th was the principle of *separating concepts from contents* used by 40 subjects. Three reasons were identified: HI, AM, and PR. Nine of the subjects used interfaces to separate concepts from interface. For example one subjects stated that, “Created an interface distinct from the implementation. [This] allows for expansion, adaptation and reuse while preserving the usefulness [of the component]. ” Five other subjects used inheritance by creating parent classes to implement this reuse design principle. One of those subjects stated, “This [separating concepts from content] was applied by creating a parent class. This allowed me to represent the concept of changing word endings [stemming], but it didn’t go with the implementation details. The implementation details were left for the child classes. In short I tried to use inheritance to applying this principle so that my subclasses implemented the content.”

4.3.3.2 Why the reuse design principles were *NOT* used

The subjects also provided feedback on the reuse design principles that they considered, but did not use, while designing their reusable component. They provided the reasons why they did not use them. For the content analysis, the same 3-stage process was applied. The results are summarized in Table 8. Seven principles that were considered and then not used are: *abstraction, clarity and understandability, encapsulation, isolate context and policy, modification, restrictiveness,* and *self-documenting code.*

The principle that was considered the most and then not used was *genericity*. Genericity was considered by 24 subjects but not used. The s-stemmer component used in this study required only manipulation of the string datatype. All but one of the subjects argued that they did not apply *genericity* because the component required manipulation of only one type of data - strings. For example, one of those subjects stated that he “did not use [genericity to accommodate multiple datatype inputs] in that there didn’t appear to be any need here for switching data types.” Another subject stated that he “did not use [genericity] because the input and output will always be strings.” One subject who considered genericity and did not use it because of the unfamiliarity with the technique (TU) stated that, “…this [genericity] could have been achieved with regular expressions. I did not use regular expression because of my unfamiliarity with them as it has been a long time since I used them.”

Table 8. Summary of content analysis - why the reuse design principles were NOT used

Rank#	Reuse Design Principle	Count#	Code - Why the principle was not used? (count)
1	Genericity	24	OS - Only string datatype required for the component (23) TU - Technique unfamiliar (1)
2	separate concept from contents	12	SC - Component is simple and not complex enough (8) WC - Would make the component more complicated to reuse (4)
3	linking of test to code	10	SC - Component is simple and not complex enough (7) ST - Self-testing instead and is sufficient (2) TU - Technique is unfamiliar (1)
4	Composition	9	SC - Component is simple and not complex enough (2) CR - Composition not required (2) PT - Preferred an alternative technique (1)
5	variability mechanism	9	SC - Component is simple and not complex enough (4) NV - Not too much variability in specifications (4) TU - Technique is unfamiliar (1)
6	Optimization	8	SC - Component is simple and not complex enough (5) EN - Efficiency was not a consideration for reusability (2) RP - Requires additional programming skills (1)
7	one component use many	5	SC - Component is simple and not complex enough (5)
8	Generality	4	SC - Component is simple and not complex enough (2) AF - Additional functionalities not worthwhile implementing (2)
9	Documentation	4	CU - Code should be sufficient understand (2) ND - Documentation not required (2)
10	well defined interface	3	SC - Component is simple and not complex enough (3)
11	commonality and variability	1	MN - Multiple systems not involved (1)
12	Parameterization	1	UT - Used alternate technique (1)

*Rank# - rank based on the number of times a reuse design principle is used, Count# - the number of times a reuse design principle is used, Code – two-letter code used to identify a reason (refer to section 5.3 for content analysis and coding)

The most common reason why a reuse design principle was not used is that the component being built is simple and not complex enough to warrant an implementation of the reuse design principle (SC). It was stated for eight of the reuse design principles: *one component use many helper components* (9), *separate concepts from content* (8), *linking of test to code* (7), *optimization* (5), *variability mechanism* (4), *well-defined interface* (3), *generality* (2), and *composition* (2). Even though these frequencies are small compared to the frequencies for principles used, this shows that the scope of this study is limited by the size and complexity of the components. One subject stated that "this simple component did not require helper components other than the standard Java strings." Another subject who did not use JUnit to implement linking of tests to code stated that "…this program [component] is so simple, I didn't feel the need to create JUnit tests, or any other kind of test suite."

The next most common reason that was identified across reuse design principles was the unfamiliarity in implementing a reuse design principle (TU). It was identified for 3 reuse design principles: *genericity* (1), *linking of test to code* (1), and *variability mechanism* (1). The subjects stated that they were unfamiliar with implementing the technique and hence did not use them. For example, one of the subjects who considered linking of test to code stated that "I did not link tests to code. I always write unit tests for programs, however, I am unfamiliar with unit testing in Java. I am not sure how to embed it in my code, as I've usually used NUnit with C#." This is in line with the issues identified in the past for the success of software reuse that education and training play an important role [1, 12, 158].

4.3.4 Correlation between reuse design principles

Table 9 shows the correlation between the reuse design principles used. A positive correlation between two reuse design principles would indicate that when one of those principles is used the other is also likely to be used. A negative correlation would indicate that when one of those principles is used, the other is not likely to be used. About 42% of the correlations was negative ranging up to -0.28. Among the negative correlations, most of them (75%) were between 0 and -0.10. Only 4 four pairs of design principles had correlation coefficient values of 0.5 or more: (variability mechanism, genericity), (documentation, linking of test to code), (linking of test to code, composition), and (composition, variability mechanism).

The maximum positive correlation value of 0.62 is between the pair of variability mechanism and genericity. From the content analysis of the feedback this was evident because the subjects argued that implementing type independence for input parameters was a way to achieve variability mechanism. For the other 3 pairs, there was no indication from the feedback that one was used because of the other. Also 50% of the correlation coefficient values were between -0.05 and 0.21. Such low values of the correlation coefficients and the content analysis of the feedback from the subjects show that, in general, the reuse design principles were orthogonally used i.e. the reuse design principles were used independently of each other.

Table 9. Correlation between the reuse design principles used (pearson's coefficient)

Reuse Design Principles	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19
well defined interface (R1)	x	0.13	0.11	0.17	0.07	0.21	0.13	-0.13	0.31	0.13	0.23	0.02	0.06	0.14	-0.07	0.16	-0.11	-0.11	0.09
documentation (R2)	0.13	x	0.33	0.35	0.04	0.25	0.51	-0.28	0.25	0.46	0.39	-0.07	0.21	0.01	-0.08	0.09	-0.09	0.12	0.12
clear and understandable (R3)	0.11	0.33	x	0.28	0.1	0.22	0.47	-0.27	0.36	0.31	0.28	-0.12	0.16	0.09	0.17	0.21	-0.08	0.12	0.12
generality (R4)	0.17	0.35	0.28	x	0.07	0.11	0.39	-0.16	0.32	0.38	0.28	0.01	0.1	0.17	0.09	0.21	-0.08	0.12	0.12
separate concept from contents (R5)	0.07	0.04	0.1	0.07	x	0.08	0.02	-0.15	0.28	0.02	0.29	-0.05	0.04	0.1	-0.06	0.22	-0.08	-0.08	0.12
commonality and variability (R6)	0.21	0.25	0.22	0.11	0.08	x	0.28	-0.21	0.29	0.28	0.39	-0.05	0.32	0.09	-0.01	0.01	-0.07	-0.07	0.15
linking of test to code (R7)	0.13	0.51	0.47	0.39	0.02	0.28	x	-0.14	0.34	0.56	0.48	-0.06	0.25	0.15	0.21	0.18	-0.06	-0.06	0.18
Encapsulation (R8)	-0.13	-0.28	-0.27	-0.16	-0.15	-0.21	-0.14	x	-0.1	-0.26	-0.14	-0.13	-0.11	-0.09	-0.15	0.04	-0.05	-0.05	0.18
one component use many (R9)	0.31	0.25	0.36	0.32	0.28	0.29	0.34	-0.1	x	0.44	0.46	0.04	0.29	0.27	0.05	0.34	-0.05	-0.05	-0.05
composition (R10)	0.13	0.46	0.31	0.38	0.02	0.28	0.56	-0.26	0.44	x	0.5	-0.18	0.4	0.12	-0.03	0.21	-0.05	-0.05	-0.05
variability mechanism (R11)	0.23	0.39	0.28	0.28	0.29	0.39	0.48	-0.14	0.46	0.5	x	-0.14	0.62	-0.02	-0.1	0.11	-0.04	-0.04	-0.04
parameterization (R12)	0.02	-0.07	-0.12	0.01	-0.05	-0.05	-0.06	-0.13	0.04	-0.18	-0.14	x	-0.13	0.1	0.14	-0.06	0.27	-0.04	-0.04
genericity (R13)	0.06	0.21	0.16	0.1	0.04	0.32	0.25	-0.11	0.29	0.4	0.62	-0.13	x	0	-0.1	-0.06	-0.03	-0.03	-0.03
optimization (R14)	0.14	0.01	0.09	0.17	0.1	0.09	0.15	-0.09	0.27	0.12	-0.02	0.1	0	x	0.33	0.35	-0.03	-0.03	-0.03
Restrictiveness (R15)	-0.07	-0.08	0.17	0.09	-0.06	-0.01	0.21	-0.15	0.05	-0.03	-0.1	0.14	-0.1	0.33	x	0.18	-0.03	-0.03	-0.03
modification (R16)	0.16	0.09	0.21	0.21	0.22	0.01	0.18	0.04	0.34	0.21	0.11	-0.06	-0.06	0.35	0.18	x	-0.02	-0.02	-0.02
abstraction (R17)	-0.11	-0.09	-0.08	-0.08	-0.08	-0.07	-0.06	-0.05	-0.05	-0.05	-0.04	0.27	-0.03	-0.03	-0.03	-0.02	x	-0.01	-0.01
self-documenting code (R18)	-0.11	0.12	0.12	0.12	-0.08	-0.07	-0.06	-0.05	-0.05	-0.05	-0.04	-0.04	-0.03	-0.03	-0.03	-0.02	-0.01	x	-0.01
isolate context and policy (R19)	0.09	0.12	0.12	0.12	0.12	0.15	0.18	0.18	-0.05	-0.05	-0.04	-0.04	-0.03	-0.03	-0.03	-0.02	-0.01	-0.01	x

4.3.5 SLOC, Effort, Productivity and Parameters

The source lines of code for the one-use (N=107) and reusable components (N=101) were measured using the SLOCCount tool [159]. The notched box plots of the SLOC measured for the one-use and reusable components are shown in Figure 8. $SLOC_{ReuseDiff}$ is the new lines of code measured by comparing the one-use and its equivalent reusable component line by line.

$$SLOC_{ReuseDiff} = (SLOC_{Reuse} - SLOC_{one\text{-}use}) \tag{5}$$

The effort taken in terms of time (hours) and the number of parameters are shown in Figure 9 and Figure 10 respectively. Figure 11 compares the productivity (in terms of SLOC/hour) of the developers for one-use vs. reuse components. Productivity is measured for the entire life cycle of component development. $SLOC_{ReuseDiff}$ is considered for the productivity of reusable components because the reusable components studied in this paper were not built from scratch; instead, they were reengineered by modifying the one-use components.

$$SLOC/hr_{ReuseDiff} = (SLOC_{Reuse} - SLOC_{one\text{-}use}) \,/\, time\ for\ reusable\ component \tag{6}$$

Understanding and interpreting box plots can be found in [23]. If the notches of boxplots of different groups overlap, then there is no statistically significant difference between the groups and if they do not overlap, there is significant difference between the groups.

The median of SLOC significantly increased for the reusable components to 92 lines of code as compared to 51 for the one-use components, an increase of 80%. The average SLOC was 62 and 110 for the one-use components and the reusable components respectively. The notches of the two box plots do not overlap and this indicates a statistically significant difference between the sizes of the two components. This increase is due to incorporating more functionality in the reusable components. The boxplots in Figure 8 also shows that there is much more variability in the SLOC measure for the reusable components. This may be because the reusable components have more functionalities and those functionalities vary from subject to subject based on the understanding of the reuse design principles; while for the one-use components the subjects may have had a more similar understanding of the functionality.

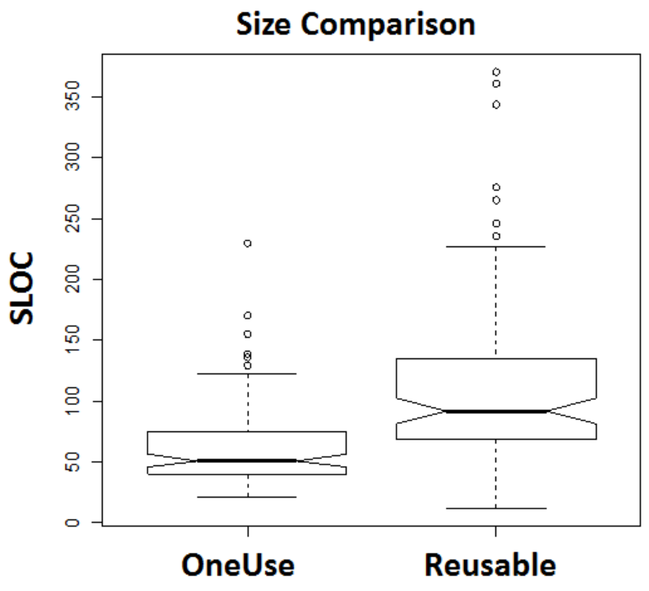


Figure 8. Comparison of actual size (SLOC)

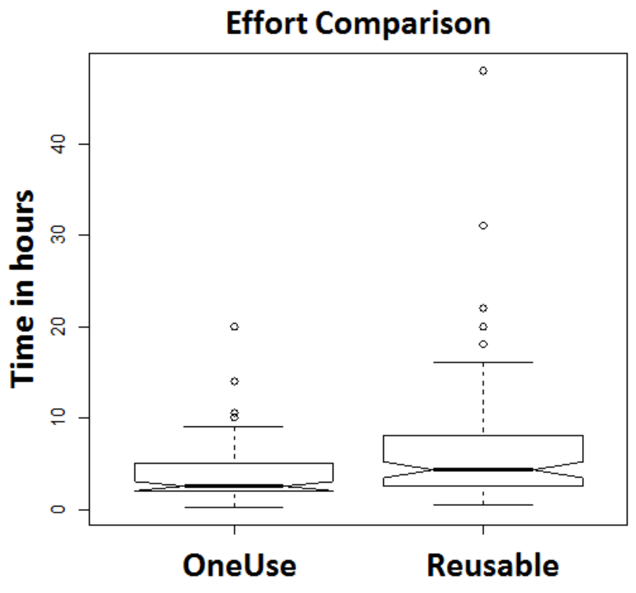


Figure 9. Comparison of effort (hours)

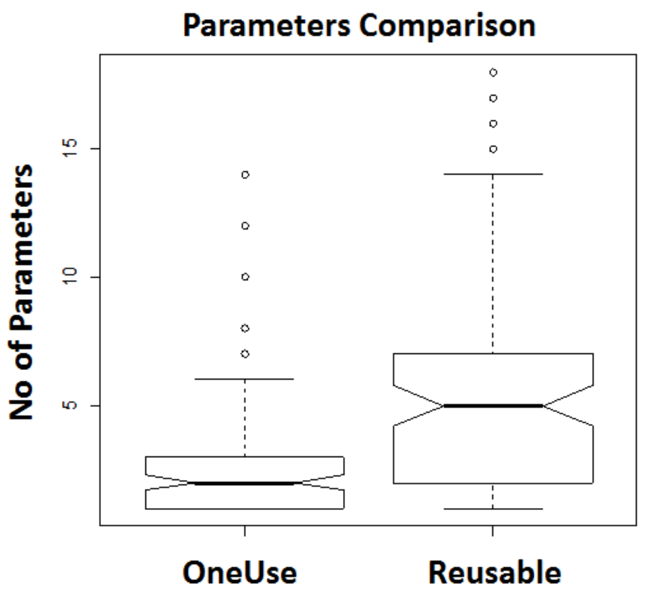


Figure 10. Comparison of #parameters

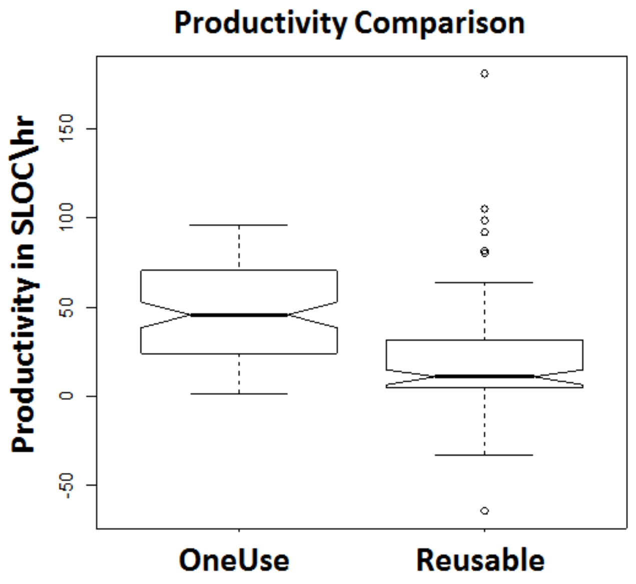


Figure 11. Comparison of productivity (SLOC/hr)

From Figure 9, the median of the time taken to implement the components was 5.0 hours and 8.0 hours respectively for the one-use and reusable components. Average time taken was 3.6 and 6.5 hours for one-use components and reusable components respectively. The notched areas of the box plots overlap for the two and this indicates no significant difference. As was the case for SLOC, the variability is higher for the reusable components. The inter quartile range for the one-use and reusable components are 3.0 hours and 5.5 hours respectively while the standard deviations are 3.0 hours and 6.8 hours respectively.

As can be seen in Figure 10, the number of parameters for the reusable components was significantly higher than for the one-use components. The medians were 2 and 5 for one-use and reusable components respectively. The mean number of parameters for the one-use components was 2.6 while the reusable components were 5.4. 40% of the one-use components had only a single parameter. In this case the variability is somewhat larger for the reusable components.

As can be seen in Figure 11, the median of the productivity was 21.0 and 6.45 SLOC/hr for the one-use and reusable components respectively. The mean for the productivity of one-use components (30.0 SLOC/hr) is almost three times the productivity the reusable components (10.6 SLOC/hr). The notches do not overlap and indicate a significant difference. This may be because more time may have been spent on the design of the reusable component than on coding when compared to the one-use component. For the productivity, the variability of the one-use component is higher than the equivalent reusable components. The standard deviations are 29.2 SLOC/hr for one-use and 15.1 SLOC/hr for reusable components. The inter quartile range for the one-use components is 25.75 SLOC/hr while it is only 9.3 SLOC/hr for the reusable components.

For SLOC comparisons, as seen in Figure 8, there are about 7 outliers for both the one-use and reuse components. A particular subject was the cause for an outlier in both groups (170 lines in one-use and 361 lines in reuse component) – the subject had the second most number of lines of code in both the one-use component group and the reusable component group. This was probably because the programmer was inefficient in programming. A second outlier in the reusable component was 370 lines whose one-use component had only 89 lines. This subject had included an additional test harness component that provided the basic console interface for stemming – this component was itself about 266 SLOC while the main stemming component had only 104 SLOC. This same subject is also the cause for an outlier in the reusable component group in Figure 9. Another outlier in one-use component in Figure 8 was 155 SLOC and the same subject had 175 SLOC for the reuse component. The one-use component was very little modified to make it reusable.

An outlier in the effort for one-use component was the same subject who had an outlier in the SLOC (the subject who had the second most SLOC in one-use component). This indicates a higher effort for higher SLOC. The outliers for the number of parameters for one-use components and the same subject also caused outliers for reusable components. Using more parameters might be a programming style followed by the subjects.

The density distributions of SLOC, Effort, Productivity and Parameters are compared for the one-use and reusable components in Figure 12. They are all positively skewed to the right and have unimodal distributions. The SLOC and effort have sharp peakedness for the one-use components. Also, the shapes of the one-use components are similar for both the SLOC and effort. The same is true for the reusable components as well. This results in the productivity distributions being similar for the one-0075se and reusable components. In one-use components the functionality is minimal and common as the emphasis is on the implementation of the algorithm. The reusable components on the other hand have more functionality and objectives of the functionalities vary from programmer to programmer due to the variability in choosing the reuse design principles given in Figure 1. This may be the cause for higher variability in the reusable components. Lower variability may be the cause for higher peakedness of the one-use components.

4.3.5.1 Design, coding, and testing efforts for reusable components

To better understand how the subjects spent their time in building the reusable components, they were asked to break down their effort into three phases – design, coding, and testing. The requirements of the reusable component were the same as the one-use component. So, the subjects did not have to spend any time on the requirements analysis for building the reusable components.

The boxplot comparison of the times spent on design, coding, and testing is shown in Figure 13. The subjects spent more time on writing code than designing the components but not significantly so. The summary statistics of the time spent on design, coding, and testing phases are summarized in Table 10.

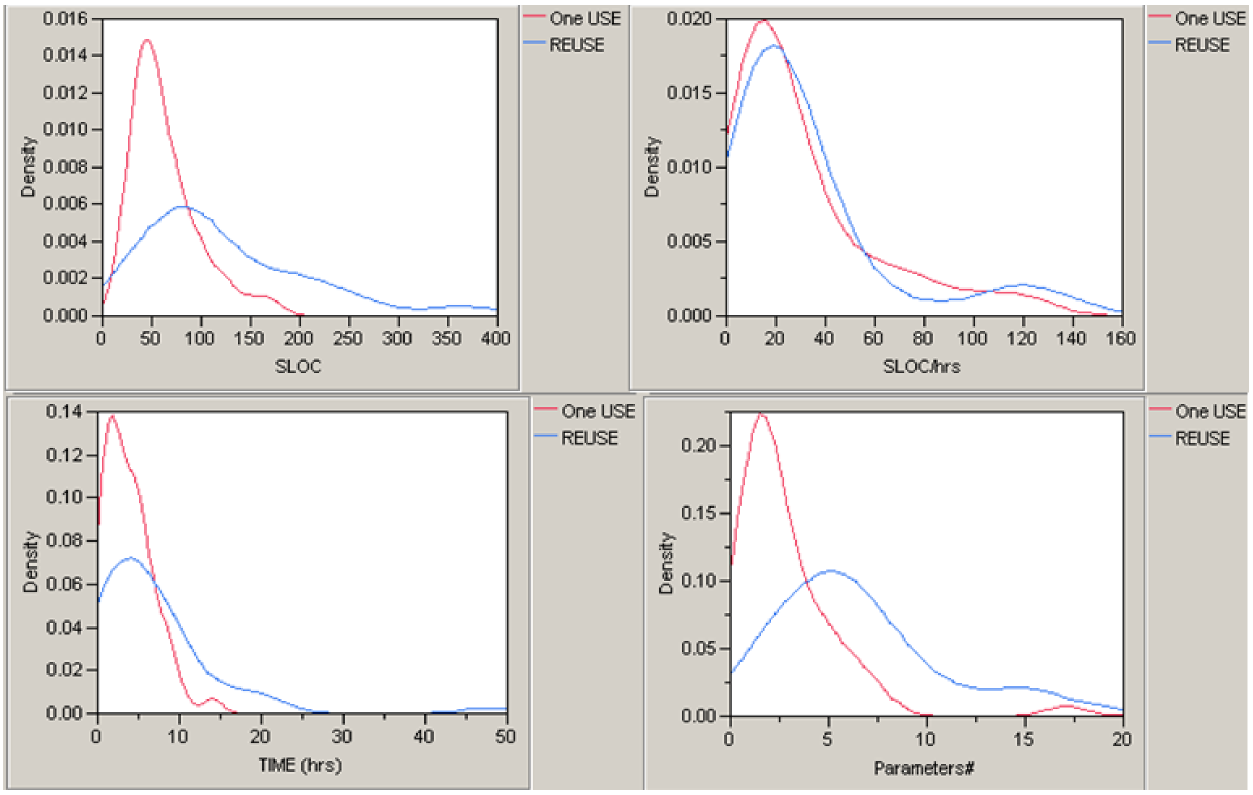


Figure 12. Density distributions of SLOC, effort, productivity and number of parameters

Table 10. Descriptive statistics of the times spent for designing, coding, and testing the reusable components

Phase	Min-Max	Mean	Median	Std. Dev.
Design	0.25-7.0	2.0	1.25	1.6
Coding	0.25-16.0	2.9	2	3.2
Testing	0.25-8.0	1.7	1	1.7

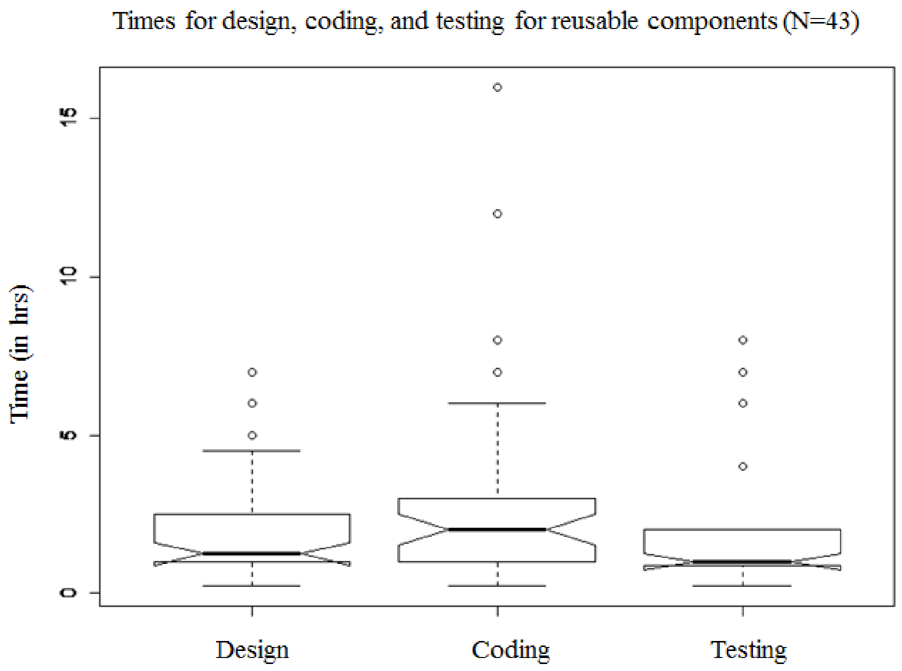


Figure 13. Distributions of the times for design, coding, and testing for reusable components

4.3.5.2 Matched Pair t-tests

SLOC, effort, productivity and the number of parameters were compared using matched pair t-tests. For this analysis, the difference in the values of the one-use and reusable components was first calculated and this difference was then analyzed using one-sample t-test with a hypothetical test mean of zero. The results are shown in Table 11.

Table 11 shows that the SLOC, effort, and the number of parameters are statistically significantly higher. The productivity also shows a statistically significant difference. The reusable components have significantly lower productivity. Comparing the values of Cohen's *d* [160] the effect sizes are "*large*" for SLOC, number of parameters, and productivity, and "*medium*" for effort.

Table 11. Matched pair t-test statistics

Variable	Mean	Std. Dev.	df	t	p-value	Cohen's *d*
$(SLOC_{Reuse} - SLOC_{one\text{-}use})$	48.6	53.7	100	9.09	0.001	0.89
$(Effort_{Reuse} - Effort_{one\text{-}use})$	3.09	5.9	97	5.1	0.001	0.56
$(Parameters_{Reuse} - Parameters_{one\text{-}use})$	2.76	2.87	100	9.64	0.001	0.88
$(SLOC_{ReuseDiff/hr} - SLOC_{one\text{-}use/hr})$	-20.53	31.67	97	-6.4	0.001	-0.81

4.3.6 Size vs. Reuse Design Principles

The effect of reuse design principles on the complexity (measured in SLOC) was studied by comparing the boxplots as shown in Figure 14-19. The top six most used reuse design principles were studied. As can be seen, the notches in the boxplots overlap for all the six reuse design principles indicating that the use of a reuse design principles does not have a significant effect on the size of the reusable components.

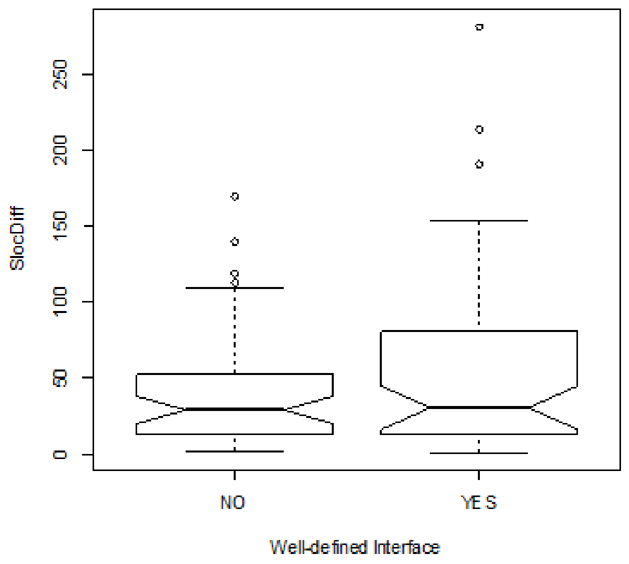


Figure 14. Size comparison of components when well-defined interface was used vs. when not used

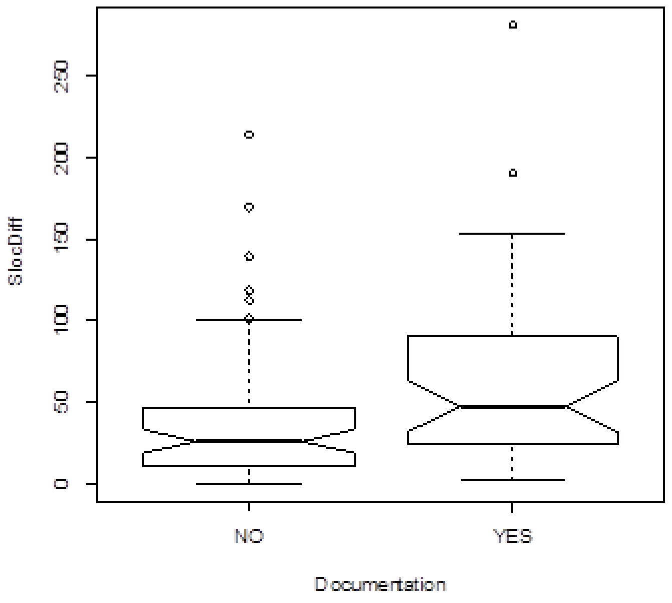


Figure 15. Size comparison of components when documentation was used vs. when not used

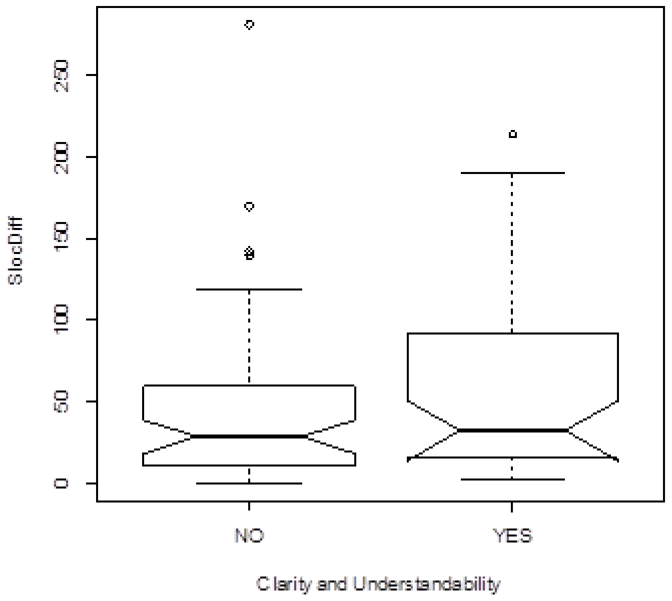


Figure 16. Size comparison of components when clarity and understandability was used vs. when not used

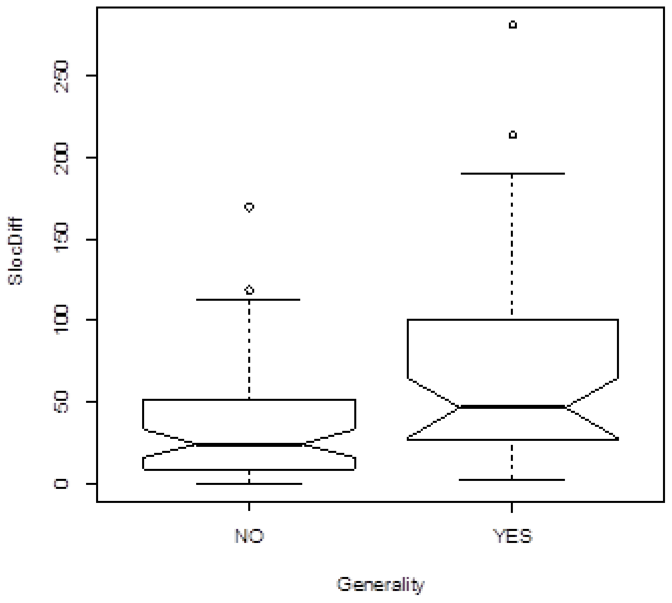


Figure 17. Size comparison of components when generality was used vs. when not used

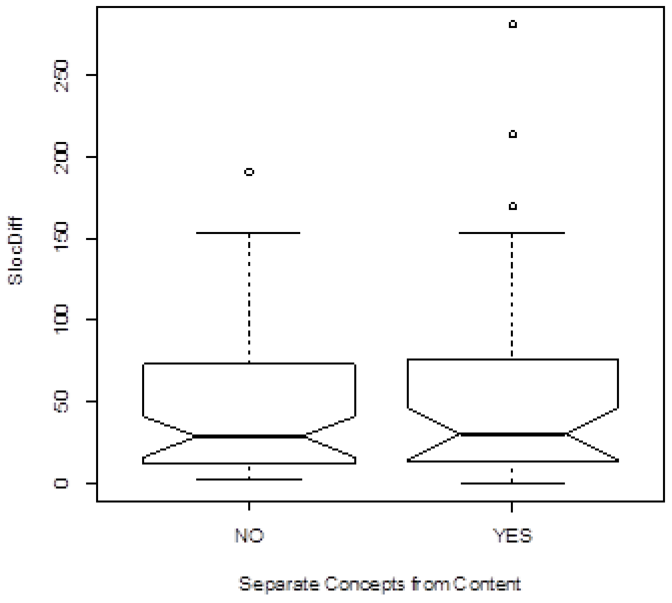


Figure 18. Size comparison of components when separate concepts from content was used vs. when not used

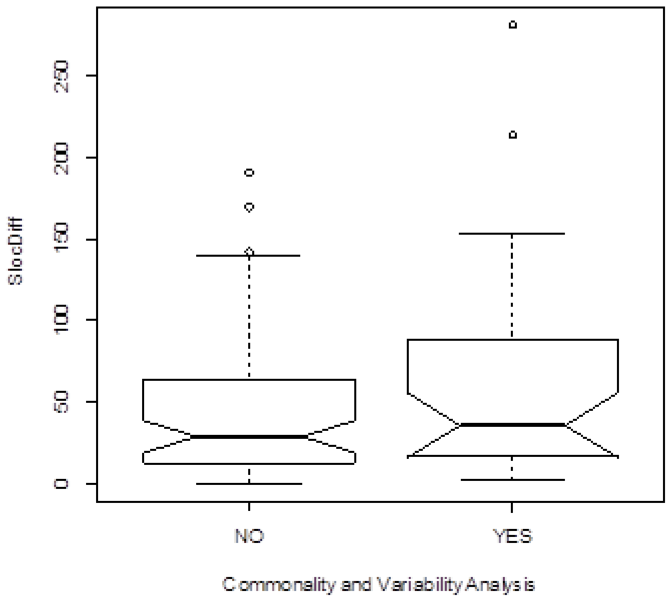


Figure 19. Size comparison of components when commonality and variability was used vs. when not used

4.4 Code Examples – Illustrating Reuse Design Principles

In this section we look at the code examples to illustrate how the subjects implemented some reuse design principles to convert their one-use components to reusable components. The five most used design principles are selected – well-defined interface, documentation, clarity and understandability, generality, and separation of concepts from content.

4.4.1 Well-defined interface

Many subjects who implemented a well-defined interface reasoned that an easy to understand and use interface is necessary for making a component easily reusable. The subjects minimized the number of functions to be public and also made them simple and easy to understand. The goal is to discourage subsequent users from looking into the implementation details but still be able to understand and use the component. One subject (refer Appendix E for the code) who implemented the well-defined interface reasoned that, “The interface for this class was pretty straight forward. I kept the helper functions private in order to simplify the interface for the user. The function `getStem` requires one parameter and returns a string. This is the only function that is seen from the classes that reference `Stemmer`. This [well-defined interface] was definitely a consideration during design time as offering too complicated an interface would be hard for another user to follow. By only making `getStem` public (and naming it something that makes sense), any other user should be able to follow what the program does and what method to use."

Some subjects also used the `interface` class in Java to implement a well-defined interface. For example, one subject used public classes in the one-use component while in the reusable component an `interface` class was written as given below. The subject reasoned that, "I introduced an `interface` class called `StemmerInterface`. The `interface` class with its methods of `setStem, getStem and displayStem` can be reused to implement Stemmers for several other words like a S-Stemmer etc. This also keeps the inner workings of the `SStemmer` hidden from the user."

```
.......
public interface StemmerInterface {

     // method signature
     void getStem(String stemWord);
     void setStem();
     void displayStem();

}
public class SStemmer implements StemmerInterface {
     private String result ;
     private SStemmer ()
     {
        result = new String ();
        result = "";
     }

     public void getStem(String stemWord)
     {
          .......
     }
     public void setStem()
     {
          .......
     }

     public void displayStem()
     {

          .......
     }
  }
.......
```

4.4.2 Documentation

For documentation, the subjects used either Javadocs, internal documentation, external documentation, or a combination of these. Internal documentation refers to the commenting within the code. External documentation is done by providing additional documents related to the component. Some subjects provided `README.txt` files as external documentation. An example of a `README.txt` provided by a subject for the reusable component is given below that has details such as the contents, different ways to use the component, example input/output, compiler compatibility, and assumptions.

```
Contents:
---------
ReusableSStemmer
  src
    SStemmer.java      --> Reusable S Stemmer java class Source
code
    TestStemmer.java --> testing program java source code
  bin
    SStemmer.class   --> Reusable S Stemmer compiled class
    TestStemmer.class    --> testing program compiled class
  doc --> Javadoc folder
  Web Service
     SStemmer.wsdl --> the stemmer Web Service WSDL file
```

```
Notes:
-----
- The SStemmer.java header, or the more readable SStemmer.html
under the doc folder contain the answers to the assignment
questions.
- A representative set of rules for French and German languages
is used. The rules are based on the reference:
J. Savoy, "Light stemming approaches for the French, Portuguese,
German and Hungarian languages," Proceedings of the 2006 ACM
symposium on Applied computing,  Dijon, France: ACM, 2006, pp.
1031-1035.

Compiler Compatibility:
-----------------------
Java 1.5

Usage 1 (running a set of predefined tests):
--------------------------------------------
ReusableStemmer\bin> java TestStemmer
     Output
     ------
     Running Stemmer Tests.
     Applying English Rules
     bunnies  -> bunny
     toes  -> toe
     classes  -> classe
     class  -> class
     bass  -> bass
     exodus  -> exodus
     fires  -> fire
     fries  -> fry
     frees  -> free
     enemies  -> enemy
     aies  -> aie
     eies  -> eie

     Applying French Rules

     chevaux  -> cheval
     fleurs  -> fleur
     voudrais  -> voudrais
     faux  -> faux

     Applying German Rules

     jahre  -> jahr
     motoren  -> motor
     hauser  -> haus

     To provide a word for stemming using basic S Stemmer Rules,
use
     > java TestStemmer <space-separated words To stem>


Usage 2 (User-supplied test):
-----------------------------
ReusableStemmer\bin> java TestStemmer <space-separated words To
stem>

     Example:
     --------
     ReusableStemmer\bin>  java  TestStemmer  turtles  bunnies
fields

     Output:
     -------

     Applying Basic S Stemmer Rules...

     turtles  -> turtle
     bunnies  -> bunny
     fields  -> field

Assumptions
-----------

  The input word must be of length equal or greater than 2
```

4.4.3 Generality

While implementing generality, the subjects looked to make the component as configurable as possible to include multiple future possible implementations. For the stemmer component the focus was to allow adding or modifying the stemming rules. For example, consider the code example given in Appendix F. A user can add or modify rules in the function `initializeSStemmer ()`. Also, during execution the rules are used in the order in which they are added. The subject reasoned that, "To be truly reusable, any component needs to be general purpose. Based on the description in #1 [one-use component], I feel this [reusable] version of the stemmer is configurable enough to handle a wide variety of stemming cases that might be presented…I created a `Stemmer` class that accepted `StemmingRules`... any kind of `Stemmer` class can be created, as long as it adheres to the rules in step 1. If more complicated rules need to be created, it is possible to subclass the stemming rule and over-ride the default behavior. A critical point here: after a stemmer is configured, its use is deceptively simple. One only has to call the `stem()` method. Based on this design, it should be possible to create a wide variety of stemmers." Another example is given in Appendix G. Here also the subject designed the component for using rules with generic endings.

4.4.4 Clarity and Understandability

Subjects who used the clarity and understandability principle looked to make the code easy to understand by just reading through the code. Clarity and understandability was applied by a subject to create the reusable component given in Appendix E. The subject reasoned that, "Making the program clear and understandable was a consideration which is the main reason that I built the helper functions and decided to use nested switch commands. This was also the reason that I placed the error checking within the helper functions. A programmer who saw the `getStem` function should be able to read it fairly easily."

4.4.5 Separate Concept from Content

Concepts refer to the representation of the abstract semantics of a component while content represents the implementation details of the component. For the stemmer component, content is the implementation of the stemming algorithm and the concept is the stemming algorithm. The subject who developed the reusable component in Appendix F used separation of concept from content. The subject reasoned that, "this [separation of concept from content] is a critical feature of reusability. Essentially, it boils down to NOT hardcoding logic, but designing a system that can have its rules changed by composition and configuration. In this program, the concept is stemming, the content are each Stemmer's individual `StemmingRules`, and the order in which they are fired (described here by the order in which they are added.)" Another subject who developed the reusable component given in Appendix H also used separation of concept from content and reasoned that, "The code is not dependent on the way Java implements various methods and data types. This could be written in any language and still could be structured very much like it is structured now. The `Stemmer` is the parent class which really is the main interface to the implementer. The `StemmerRuleManager` simply handles or abstracts away the details of how it accomplishes its tasks."

4.5 Threats to Validity

The threats to external and internal validity for this study are presented based on the discussion in Chapter 6 of the book by Wohlin et al. [161].

4.5.1 Threats to External Validity

All components were developed only in Java. So, the results may not be valid for other languages. The components are also small in size. Realizing that the components in this study are small and only in Java, similar studies may be needed with larger reusable components and in other languages as well.

The issue of using students as subjects in software engineering experiments has been discussed in the past [24-28] and there has been mixed results on whether students could provide the same results as using professionals. However, the students considered in these studies were full-time students. Most of the subjects in this study are working professionals with varying experiences in the software industry and enrolled as part-time students at the University. Almost two-thirds had four or more years of experience in software engineering. About three-fourths (74%) of the subjects had four or more years of programming experience. About half (47.8%) had more than 8 years of programming experience. None of the subjects had absolutely no experience in software programming. Carver et al. [27] have mentioned that the gap between students and novice professionals are decreasing especially in the context of the US educational climate.

4.5.2 Threats to Internal Validity

Carver et al. [27] have also identified that the most important threat to internal validity in having students as subjects is that they can exchange answers to improve the grades. However, in this study, it was made clear to the students that the grade is based on whether the components submitted worked or not, and did not depend on the reuse design principles used. It was also verified by the instructor that no two components had common lines of code.

The reuse design principles for a given component were identified by the developer of that component. The course grader validated the reuse design principles and those are used in this study. The developers also had to report why they chose the reuse design principles they used. This helped to alleviate the threat to the validity of the reuse design principles used. Also, the choice of reuse design principles can be influenced by the application type. The type of application in this study is a simple rule-based algorithm. It is intuitive that applications implementing stacks or queues would encourage more use of principles like genericity.

Chapter 5: Designing and Building *with* Reusable Components

In the previous study (Chapter 4), subjects built one-use stemming components [13]. The subjects were then trained on software reuse design based on a set of reuse design principles and converted their one-use components to be reusable. The one-use components were found to be significantly smaller in size compared to their equivalent reusable components. The six most commonly used reuse design principles were identified in the study and they were *well-defined interface, documentation, clarity and understandability, generality, separate concepts from contents,* and *commonality and variability analysis*.

In the *with* reuse process, successful reuse of the components depends on how easily a user can integrate them into a system. It is important to understand the factors that affect the ease of reuse. Through an empirical study presented in this chapter, some human factors that may affect the ease of reuse are analyzed. The human factors studied are the experience level of the user in software reuse and experience level in a programming language. Whether component testing makes it easier to reuse or not is also analyzed. This study also analyzes the effect of the size of components on the ease of reuse. The effect of each reuse design principle on the ease of reuse is also analyzed.

The ease of reuse is measured on a reusability scale in this study. The reusability score of a component was measured as the ease of reuse as perceived by the subjects reusing the component using a 5-point Likert scale: (1 – not used, 2 – difficult to reuse, 3 – neither difficult nor easy to reuse, 4 – easy to reuse, 5 – very easy to reuse). The Likert scale is similar to the one used in [162]. Few or no empirical studies were found similar to this study.

Thirty-four subjects participated in the study with each subject reusing 5 components, resulting in 170 cases of reuse. The components were randomly assigned to the subjects from a pool of 25 components which were designed and built for reuse. The effect of the complexity of a component on the ease of reuse is analyzed by a regression analysis. It was observed that the higher the complexity the lower the ease of reuse, but the correlation is not significant. An analysis of the effect of a set of reuse design principles, used in designing and building the components, on the ease of reuse is also reported. The reuse design principles: well-defined interface, clarity and understandability, generality, and separation of concept from content significantly increase the ease of reuse. Documentation does not have a significant impact on the ease of reuse while the reuse design principle of analyzing commonalities and variabilities has a significant negative impact.

5.1 Hypotheses

When reusable components are used in other applications, we now revisit the four hypotheses related to deign with reuse presented in Chapter 1. In general components are considered less complex when smaller in size measured by source lines of code (SLOC). Hence, smaller components should be easier to reuse. When a component is built for reuse, the reuse design principles used must aid improvement in the ease of reuse. Generally, experience is an indicator of expertise. Hence, a programmer with higher experience should be reusing components with greater ease. Also, when a programmer tests a component before using it, the programmer gets a better understanding of the component. This should improve the ease of reusing the component.

Hypothesis II-a: The smaller the component the easier it is to reuse. The size is measured in SLOC (source lines of code).

Hypothesis II-b: A component designed and built with a given reuse design principle will be easier to reuse than a component which is not built using that reuse design principle. In this study, the effect of the six most used reuse design principles as identified in the study in Chapter 4 are considered: *well-defined interface, documentation, clarity and understandability*, *generality*, *separate concepts from contents*, and *commonality and variability.*

Hypothesis II-c: The more the experience a programmer has, the easier it is for the programmer to reuse a component. For Hypothesis II-c, three types of experiences in a programmer are considered – programming experience, software reuse experience, and programming language experience.

Hypothesis II-d: A component, when tested by the user before reuse, is easier to reuse than a component which is not tested by the user before reuse.

5.2 Method

Based on the faceted classification of types of software reuse by Frakes and Terry [2], the reuse design in this study involves *development scope* as internal, *modification* as white box, *domain scope* as vertical, *management* as ad hoc, and *reused entity* as code.

A total of 34 subjects participated in this study. Almost all the subjects had some experience level in software engineering and programming. The demographics of the subjects are discussed next.

5.2.1 Subject Demographics

All of the 34 subjects who participated were students of a graduate level course: *Software Design and Quality*. All were enrolled either at the Master's or Ph.D. level at Virginia Tech, U.S. Nine subjects (27%) already had a master's degree and had enrolled for a second master's or at the doctoral level. The rest of the subjects (73%) had an undergraduate degree and were enrolled at master's level. The subjects completed an online questionnaire hosted on SurveyMonkey (http://www.surveymonkey.com/) answering questions on their demographics (refer Appendix C for the survey). The questionnaire was completed by the subjects before they were given the assignment of reusing the components.

5.2.1.1 Roles of the subjects

The subjects were asked their roles in their respective organizations. They could choose multiple roles and 8 had at least 2 roles; five of them mentioned that they had 2 roles while three had 3 roles in their organizations. Figure 20 shows the distribution of the roles.

Almost two-thirds were involved in development and programming. One-fifth of the subjects were system architects. Five subjects (14.7%) were both system architects as well as developers/programmers. Less than one—fifth (17.6%) of the subjects were systems engineers. Two of them were system architects as well. Less than one-fifth (17.6%) of the subjects were managers; one of them was only a manager, 2 were system architects as well and 2 were systems engineers as well. Four of them mentioned their role as 'other', 2 of them were data consultants, 1 a software consultant and 1 held a military position with no affiliation to software engineering.

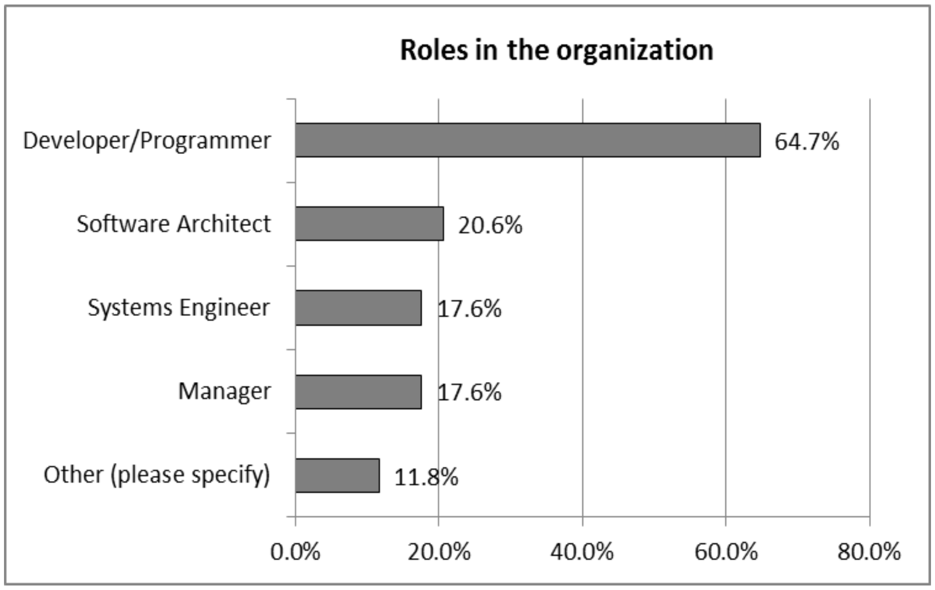


Figure 20. Distribution of the roles the subjects has in their organizations

5.2.1.2 Experience in software engineering and programming

Figure 21 shows the distribution of the subjects' experience in software engineering and programming. As can be observed, half of the subjects had more than 8 years of experience in programming as well as in the field of software engineering. Only 1 subject mentioned having no experience in software programming. Two subjects, including the subject having no experience in software programming had no experience in software engineering. Less than 15% of the subjects had none or very little experience (0-1year) in software programming and software engineering. More than one-fourth (26.5%) of the subjects had at least 2 years of experience in programming but less than 8 years.

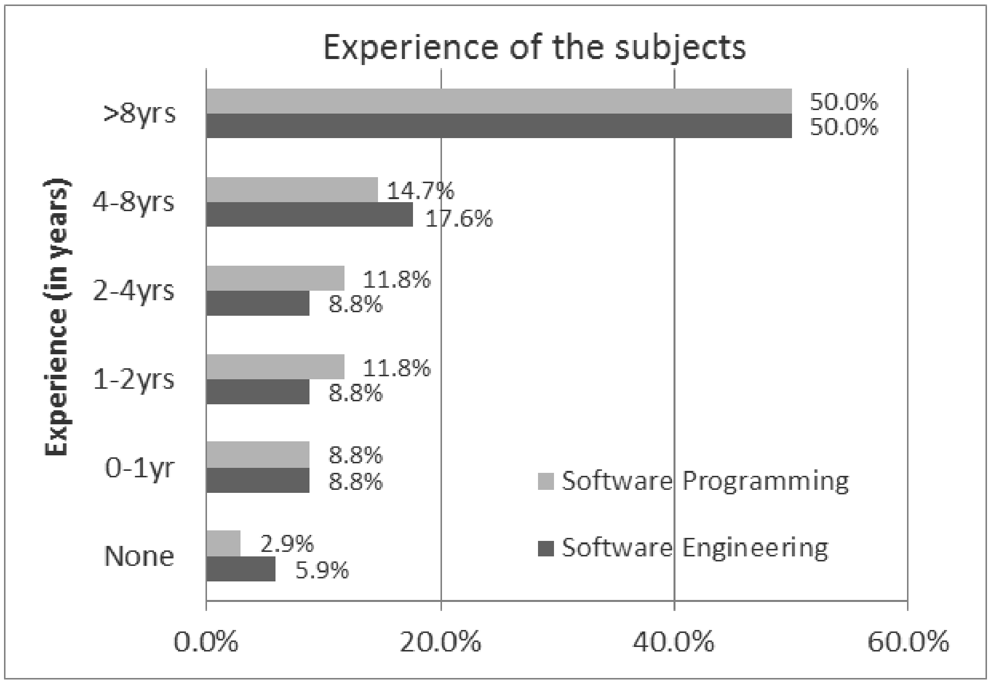


Figure 21. Distribution of the subjects' experience in software engineering and software programming

5.2.1.3 Experience in software reuse

More than four-fifths (82%) of the subjects had mentioned they had no software reuse program in their organization. Only 19% of the subjects responded that they were trained to design and build components for reuse. In an earlier study [158] too, the percentage of respondents who said they had been educated on software reuse was low (13%). The percentage of respondents who said they had training a program on software reuse in their organization was also low (19%).

Figure 22 shows the distribution of the subjects' experience in the field of software reuse. Over one-third (35.3%) had no experience in the field of software reuse and another one-fifth had very little experience (0-2 years). Less than one-tenth (8.8%) had considerable experience (>8 years). The distribution is bi-modal and represents two samples of population, one with no experience and the other with at least some experience. The sample with at least some experience is negatively skewed and shows that the subjects with experience had relatively higher experience than most in the sample.

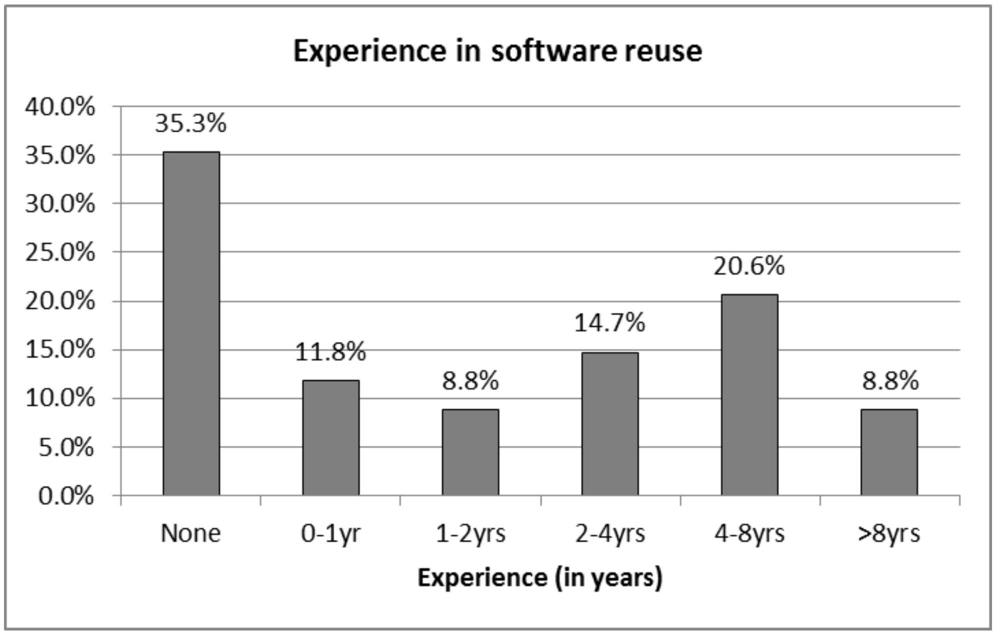


Figure 22. Distribution of the subjects' experience in the field of software reuse

5.2.1.4 Experience Levels in Java Programming

Figure 23 shows the distribution of the experience levels of the subjects in java programming. More than half of the subjects (61.8%) had very low experience (less than 2 years) in java programming. About one-third (29.4%) had a moderate experience of 2-8 years. Less than one-tenth (8.8%) had very high experience in java programming. The distribution is fairly normal and is unimodal unlike the distribution for software reuse experience (Figure 22).

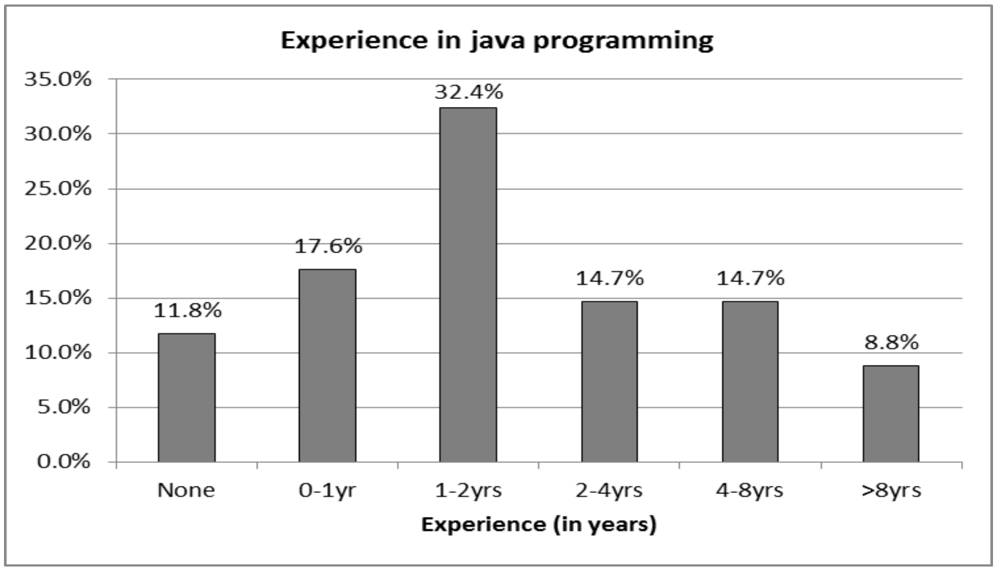


Figure 23. Distribution of the subjects' experience levels in Java programming

5.3 Data Collection

5.3.1 Component allocation

In the previous study (Chapter 4), one-use components and their equivalent reusable components were analyzed. In that study, the subjects were given an assignment to build a one-use software component implementing the s-stemming algorithm [13]. This was followed by training for the subjects on designing and building components for reuse. One hundred and one subjects then converted their one-use stemmer component to a reusable component. All the components were developed in Java. The s-stemming algorithm implemented was specified by 3 rules as given below (only the first applicable rule was used) [134]:

If a word ends in "ies" but not "eies" or "aies" then Change the "ies" to "y",

For example, cities → city

Else, If a word ends in "es" but not "aes", "ees", or "oes" then change "es" to "e"

For example, rates → rate

Else, If a word ends in "s", but not "us" or "ss" then Remove the "s".

For example, lions → lion

Twenty-five components from the sample of 101 components from the study in [163] were randomly selected for this study. From the pool of the selected 25 components, each of the 34 subjects participating in this study was randomly allocated 5 components. While every subject was given 5 components, each component was not allocated the same number of times due to the random process. Component allocation varied from 5 to 8 times. Table 12 shows the distribution of the component allocation; for example 2 of the selected 25 components were allocated to 5 subjects resulting in 10 (2*5) cases of reuse. The total number of reuse cases analyzed in this study is thus 170 (34*5). The subjects in this study are entirely different from the subjects of study in [163].

Table 12. Component allocation matrix

# of components (A)	Frequency of their allocation (B)	# of reuse (= A*B)
2	5	10
5	8	40
6	6	36
12	7	84
		170

In this study, the subjects were given an assignment as given below. The task was to create a user-interface application that accepts an input string of characters in a text box. On the click of a button the stemmed string should be displayed in another textbox. The subjects were to use the 5 components to stem the string and display the result from the component in the output box. The subjects chose the way they wanted to reuse the components. Some chose to display the results from all the components on the user interface by the click of a single button while some gave the option on the user interface of choosing the component to be used. The subjects also had the freedom to choose any operating system, programming language, and development environment. The subjects had to turn in the source code and the executables for the assignment. The subjects then completed an online questionnaire. The results are discussed in section 5.4.

ASSIGNMENT: Reusing component in an application.

Create a user-interface application that accepts an input string of characters in a text box. On the click of a button the stemmed string should be displayed in another textbox. The implementation of the stemming algorithm is provided as a java component. Some ideas of applications are:

1. A web-page written in JavaScript, JSP, ASP.NET etc.
2. A mobile app in Android, iPhone or others smart phones.
3. A desktop application written in C#, VB or any other language
4. As add-ons in other applications like writing a macro in excel or in Firefox etc.
5. If you are choosing any other option than the above 4, please contact Reghu Anguswamy (reghu@vt.edu) with the necessary details for approval before starting your assignment.

Five java components will be given, each implementing the stemming algorithm:

Three rules specify the s-stemming algorithm as follows (only the first applicable rule is used):

If a word ends in "ies" but not "eies" or "aies" then Change the "ies" to "y",

For example, cities → city

Else, If a word ends in "es" but not "aes", "ees", or "oes" then change "es" to "e"

For example, rates → rate

Else, If a word ends in "s", but not "us" or "ss" then Remove the "s".

For example, lions → lion

Build the application using all the 5 components and complete the questionnaire for all the components at the link below (the questionnaire is to be taken after using all the five components):

http://www.surveymonkey.com/s/ZN58YZ7

Deliverables: Source code and executables using all the 5 components, COMPLETE documentation (like a README file) on how to compile, run, and test the source code and executables.

Grading: Compiling and executing - 50% (10% for each component), completing questionnaire for all 5 components - 50% (10% for each component)

5.3.2 Description of selected components

In the previous study in Chapter 4, the subjects were given training on designing and building reusable components. Nineteen reuse design principles were taught to the subjects via class lectures. That study identified six most frequently used reuse design principles as – *well-defined interface*, *documentation*, *clarity and understandability*, *generality*, *separate concepts from contents* and *commonality and variability analysis*. The distribution of the reuse design principles in the 25 components selected for this study is shown in Table 13. For example, 13 of the 25 components in this study were designed and built using a well-defined interface.

Table 13. Distribution of the reuse design principles in the components selected for this study

Reuse Design Principle	Count#
Well defined interface	13
Documentation	15
Clarity and understandability	13
Generality	14
Separate concept from contents	11
Commonality and variability	9

As previously discussed in section 4.2.3, the complexity of the components was measured in terms of their size in SLOC (source lines of code). SLOC is one of the first and most used software metrics for measuring size and complexity. In a survey by Boehm et al. [135], many cost estimation models were based directly on size measured in SLOC. COCOMO [136], COCOMO II [21], SLIM [137], and SEER [138] are some of them. Complexity of software components has been measured based on SLOC in many empirical studies [139-142].

The correlation between SLOC and many complexity measures such as the McCabe's cyclomatic complexity [145] and Halstead's metrics as given in [146] was studied by Herraiz et al. [144]. In their study they presented empirical evaluations showing that SLOC is a direct measure of complexity, the only exception being header files, which showed a low correlation with the McCabe's cyclomatic complexity measures. In work presented by Graylin et al. [147], evidence was provided that SLOC and cyclomatic complexity have a stable practically perfect linear relationship that holds across programmers, languages, code paradigms (procedural versus object-oriented), and software processes. Linear models have been developed relating SLOC and cyclomatic complexity. Buse et al [148], for example, presented a study where they showed high direct correlation between the SLOC and the structural complexity of the code.

A direct correlation between the number of faults in a software component and source lines of code (SLOC) has been reported in a study by Gaffney [149]. Another empirical study that showed a direct correlation between SLOC and the number of defects in software components was reported by Krishnan et al. [150]. Based on these studies, the complexity of the reusable components in this study is based on their size (in SLOC).

Figure 24 shows the distribution of the size of the 25 selected components. The smallest component had 37 SLOC and the largest component had 361 SLOC. Half of the components had SLOC between 77 (25th percentile) and 136 (75th percentile). Twenty of the components were between 50 and 150 SLOC. The largest component (361 SLOC) is an outlier and the rest of the components have fairly a normal distribution.

5.4 Results and Analysis

After completing the assignment on reusing the components, the subjects completed an online questionnaire hosted on SurveyMonkey (http://www.surveymonkey.com/) giving feedback on the applications they built and on the components they had used (refer Appendix D for the survey). The subjects gave details of the environment they used for building the application including the operating system (OS), programming language, and the IDE (Integrated Development Environment) used. They are summarized in Table. Thirty subjects developed their applications in Windows XP/Vista/7 while four others used the Mac OS. The most favored language was Java being used by 28 subjects. Others developed in C#, JSP or JRuby. About two-thirds (67%) used the NetBeans Version 6.9 or higher as the IDE while one-fifths (20.4%) used the Eclipse IDE.

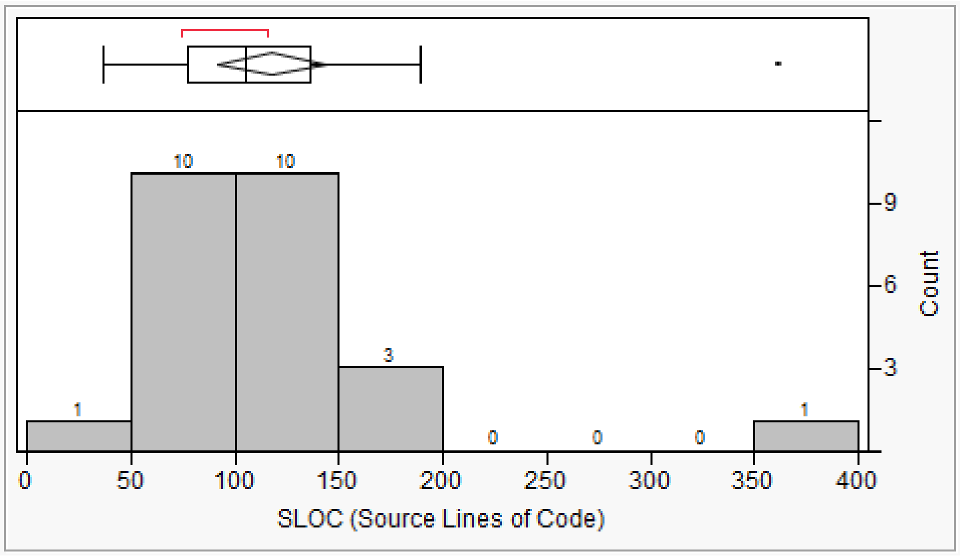


Figure 24. Distribution of the 25 components' SLOC (source lines of code)

Table 14. Distribution of the OS, programming language, and IDE used by the subjects to develop their applications

Operating System (OS)	Windows XP/Vista/7	30
	Mac OS	4
Programming Language	C#	2
	Java	28
	JSP	3
	JRuby	1
Integrated Developing Environment (IDE)	Netbeans 6.9 or higher	23
	Eclipse	7
	Visual Studio 2005 or higher	2
	Other	2

Thirty-four subjects participated in this study with each subject reusing 5 components resulting in a total of 170 cases of reuse. In the online questionnaire, the subjects rated each of their 5 components separately for a reusability score on a scale of 1-5 (1 – not used, 2 – difficult to reuse, 3 – neither difficult nor easy to reuse, 4 – easy to reuse, 5 – very easy to reuse). The distribution of the reusability scores is given in Figure 25. Almost half of the reuse cases (48.8%) were either easy (score of 4) or very easy (score of 5). Twelve of the cases (7%) were not reused at all. About one-fifth (19.4%) of them were neither easy nor difficult (score of 3).

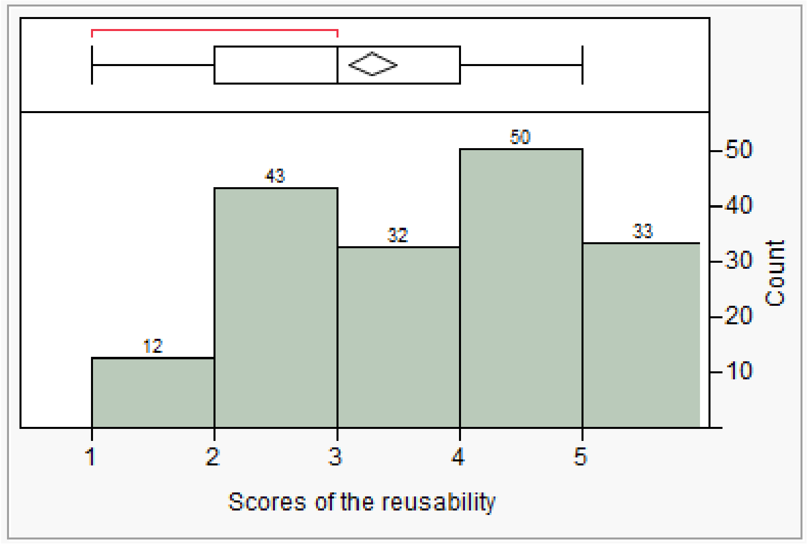


Figure 25. Distribution of the ease of reusability scores

Each of the 25 selected components in this study was allocated to from 5 to 8 subjects (refer Table 12). The average ease of reusability score for a component was calculated as the sum of all the reusability scores for that component divided by the number of reuses. For example, consider a component that was allocated to 5 subjects. The 5 subjects then reused the component and each subject gave the component a reusability score. Let the reusability scores of the component by the 5 subjects be 2, 4, 2, 3, and 1. The sum of the reusability scores is 12 (2+4+2+3+1). The average reusability score for the component is then 2.4 (=12/5). The distribution of the average reusability scores for the 25 components used in this study is given in Figure 26. The mean of the average reusability scores was 3.2 and the median was 3.3 with a standard deviation of 0.8. Four components had an average reusability score greater than 4. The highest average score for a component was 4.4. That component was reused by 7 subjects with three of them giving it a score of 5 and the other four giving it a score of 4. Two components had average reusability scores less than 2. One component which had an average score of 1.4 could not be used by 5 of the 7 subjects who were allocated the component. Another component which had an average reusability score of 1.7 was the largest of the 25 components with 361 SLOC. It was allocated to 7 subjects but not reused by 2 subjects and the 5 who reused it, all gave a score of only 2. This might be an indication that the larger the component the more difficult it is to reuse.

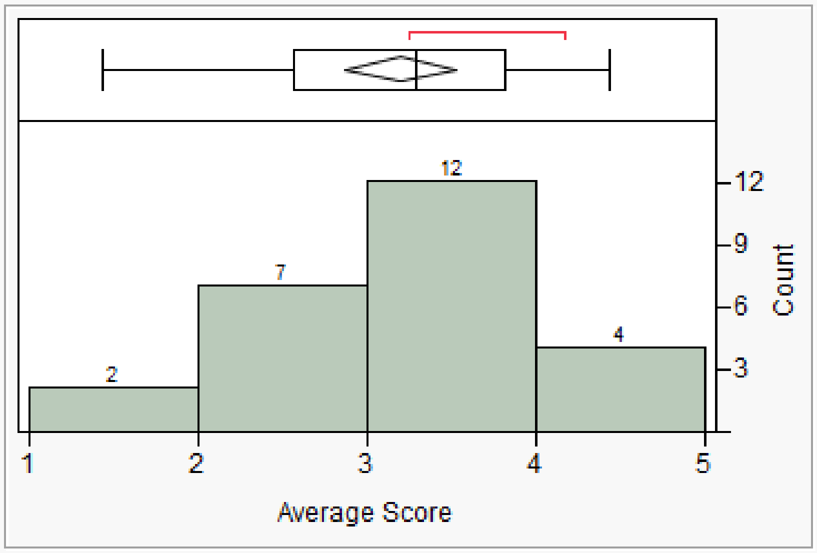


Figure 26. Distribution of the average scores of reusability for the 25 components

5.4.1 Complexity of components vs. reusability of the components

A bivariate plot with a linear fit of the SLOC vs. the average reusability scores of the components is shown in Figure 27. The regression equation of the line fit is: Average Reusability Score = 3.67 – 0.004*SLOC. The negative slope indicates a negative correlation (i.e. the higher the SLOC the lower the reusability score for a component). Because this was an ease of use measure, with a score of 5 = very easy to use, the negative relationship implies that smaller, less complex components (fewer SLOC) tend to be easier to reuse. Although this is consistent with Hypothesis I, results were not statistically significant ($F = 2.63$, $p = 0.12$). Also, the R^2 was very low (0.102), indicating that only 10% of the variability in ease of reusability was explained by SLOC.

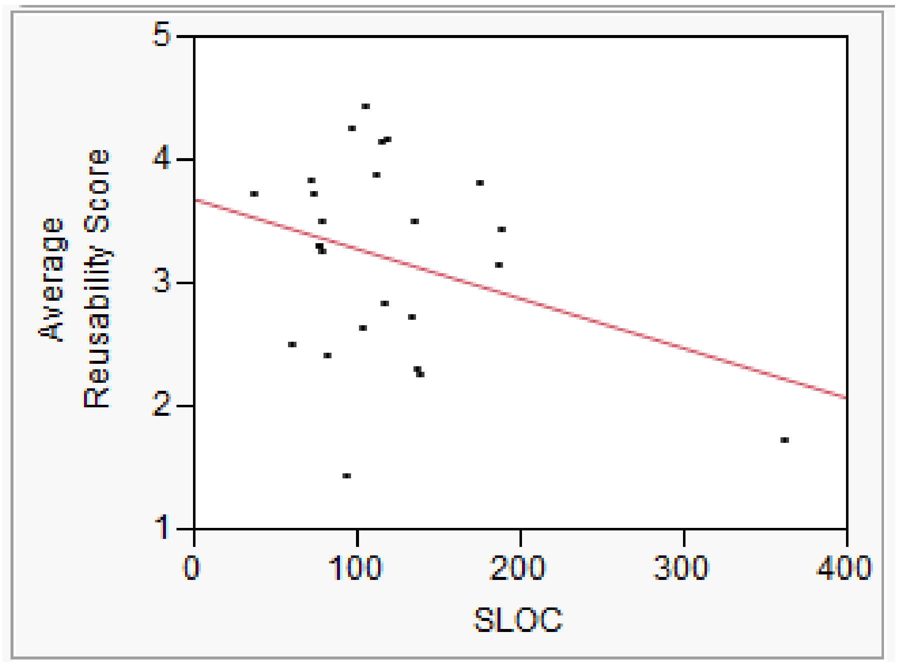


Figure 27. Bivariate fit of SLOC vs. the average reusability scores of the components

5.4.2 Reuse design principles vs. average reusability of the components

For each of the six main reuse design principles, the 170 reuse cases were repeatedly divided into two groups: cases where the reused component was built using a given principle and cases where it was not. Table 15 gives the number of components that fell into each group and summarizes the statistics comparing cases with and without each reuse design principle. The effect of a reuse design principle on the ease of reuse was further explored by comparing the boxplots of each set of reusability scores. Understanding and interpreting box plots can be found in [23]. If the notches of boxplots of different groups overlap, then there is no significant difference between the medians of the groups and if they do not overlap, there is significant difference between the medians of the groups. The boxplots were generated using the statistical software `R 2.14.2` (http://cran.r-project.org/).

Table 15. Ease of reusability for components built with and without reuse design principles

Reuse Design Principle	Reuse cases WITH the principle			Reuse cases WITHOUT the principle		
	N	Mean	Std. Dev.	N	Mean	Std. Dev.
Well-defined interface	88	3.42	1.21	82	3.15	1.26
Documentation	102	3.35	1.22	68	3.19	1.26
Clarity and Understandability	92	3.40	1.20	78	3.15	1.27
Generality	97	3.41	1.28	73	3.12	1.17
Separate concept from content	74	3.40	1.19	96	3.19	1.27
Commonality and Variability analysis	62	2.92	1.27	108	3.50	1.17

5.4.2.1 Well-defined interface

Of the 25 components used in this study, 13 of them had a well-defined interface. Of the 170 cases of reuse 88 of them were had a well-defined interface the rest 82 were without a well-defined interface. Figure 28 shows the distribution of the reusability scores of the 88 components which had a well-defined interface. The distribution is negatively skewed and shows that most components were easier to reuse resulting in higher reusability scores. The mean reusability score is 3.4 with a standard deviation of 1.2. More than half of the reuse cases with a well-defined interface (47) had either a score of 4 or 5 indicating that they were easy to reuse.

Figure 29 shows a boxplot comparison of the reusability scores of components with and without a well-defined interface. For the group with a well-defined interface the median was 4.0 and the group without a well-defined interface had a median of 3.0. The notches of the boxplots do not overlap and the notch is greater for components with a well-defined interface. This indicates that components with a well-defined interface have significantly higher reusability scores.

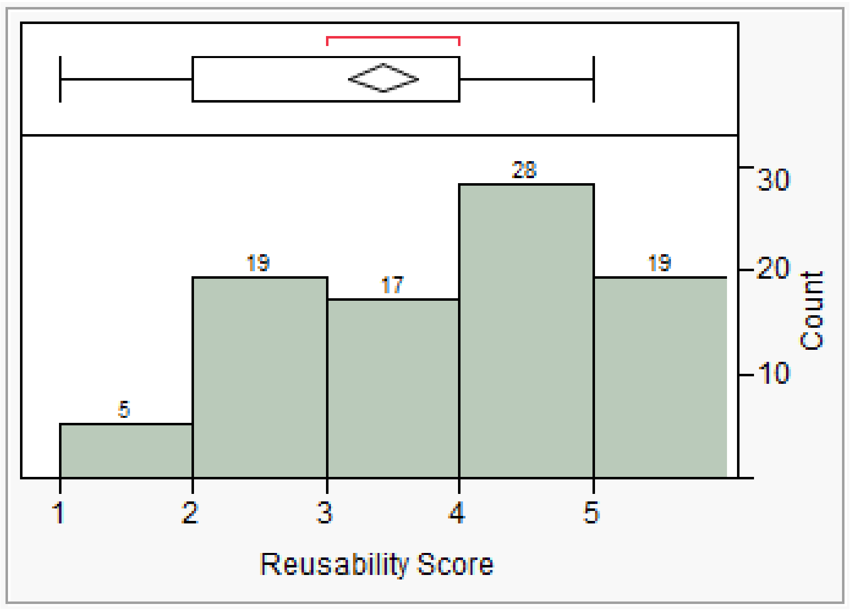


Figure 28. Distribution of the reusability scores of the components which had well-defined interfaces

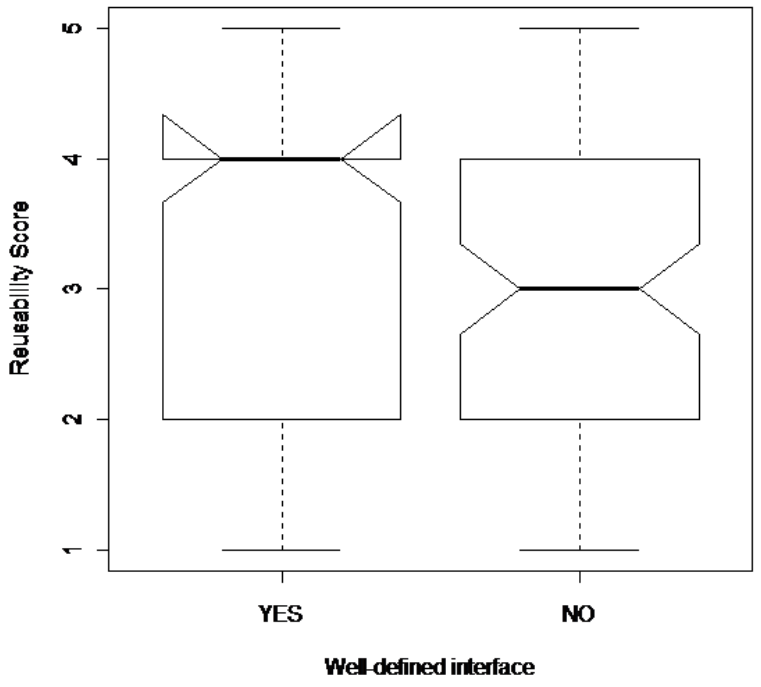


Figure 29. Box-plot comparison of the reusability scores of components with and without a well-defined interface

5.4.2.2 Documentation

Of the 25 components used in this study, 15 of them had documentation and of the 170 cases of reuse 102 of them had documentation (44 had only Javadocs, 29 had internal/external documentation, and 29 had both Javadocs and internal/external documentation). Figure 30 shows the distribution of the reusability scores of the components which had documentation. The mean reusability score is 3.3 with a standard deviation of 1.2. About half of the reuse cases (49.5%) with documentation had either a score of 4 or 5 indicating they were easy to reuse.

Figure 31 shows a boxplot comparison of the reusability scores of components with and without documentation. For the group with documentation the median was 3.5 and the group without documentation had a median of 3.0. The notches of the boxplots overlap. The notch is higher for components with documentation but not significantly as the notches of the boxplots overlap.

5.4.2.3 Clarity and Understandability

Of the 25 components used in this study, 13 of them used the reuse design principle of clarity and understandability. Of the 170 cases of reuse 92 of them used clarity and understandability. Figure 32 shows the distribution of the reusability scores of the components which were built with clarity and understandability. The mean reusability score is 3.4 with a standard deviation of 1.2. More than half of the reuse cases (53.2%) had either a score of 4 or 5 indicating they were easy to reuse.

Figure 33 shows a boxplot comparison of the reusability scores of components with and without clarity and understandability. For the group with clarity and understandability the median was 4.0 and the group without clarity and understandability had a median of 3.0. The notches of the boxplots do not overlap and the notch is greater that for components with clarity and understandability. This indicates that components with clarity and understandability have significantly higher reusability scores.

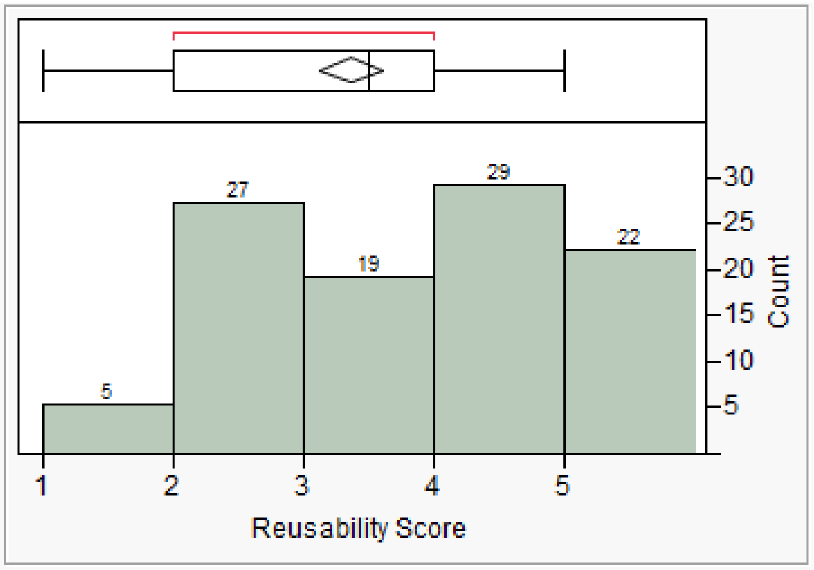


Figure 30. Distribution of the reusability scores of the components which had documentation

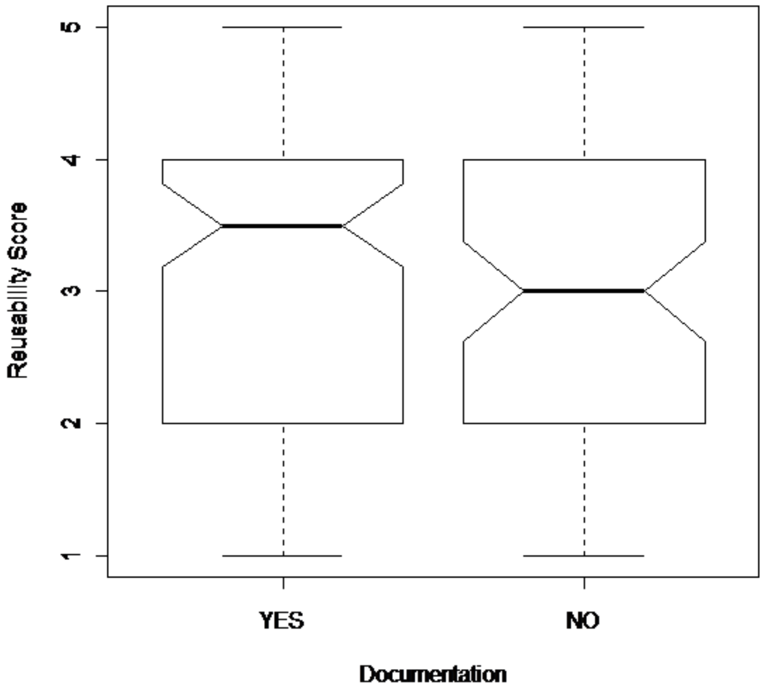


Figure 31. Box-plot comparison of the reusability scores of components with and without documentation

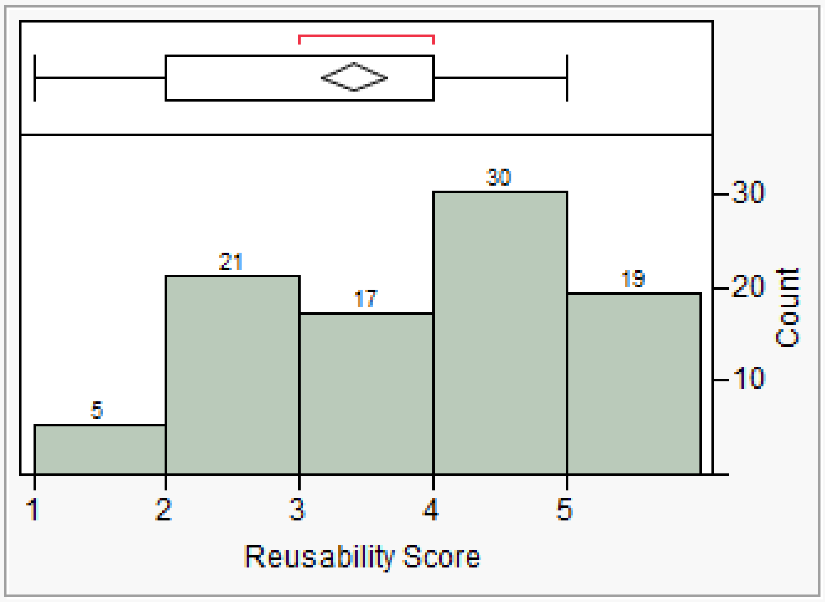


Figure 32. Distribution of the reusability scores of the components which had the reuse design principle "clarity and understandability"

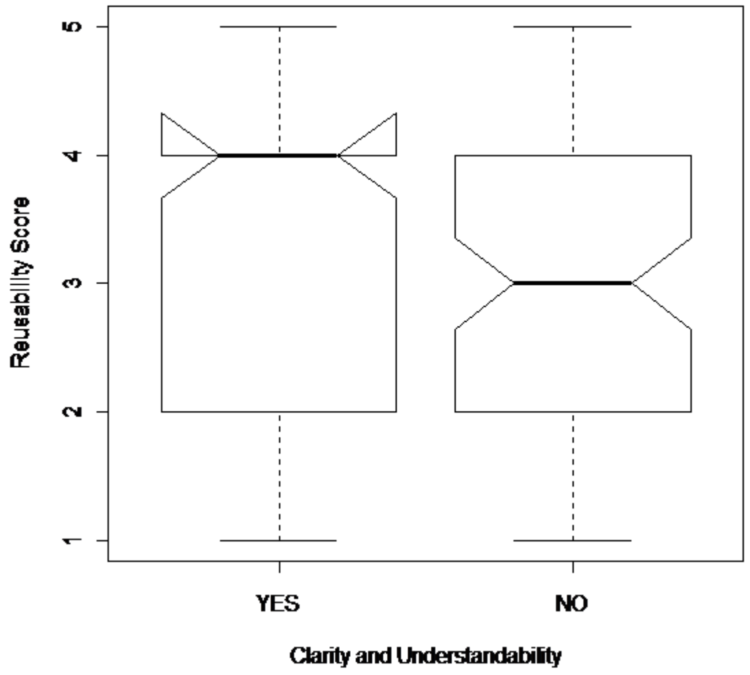


Figure 33. Box-plot comparison of the reusability scores of components with and without clarity and understandability

5.4.2.4 Generality

Of the 25 components used in this study, 14 of them had the reuse design principle of generality. Of the 170 cases of reuse 97 of them were built with generality. Figure 34 shows the distribution of the reusability scores of the components that were built with generality. The mean reusability score is 3.4 with a standard deviation of 1.3. Nearly 56% had either a score of 4 or 5 indicating they were easy to reuse.

Figure 35 shows a boxplot comparison of the reusability scores of components with and without generality. For the group with generality the median was 4.0 and the group without generality had a median of 3.0.The notches of the boxplots do not overlap. Also the notch is higher for components built with generality. This indicates that components with generality have significantly higher reusability scores.

5.4.2.5 Separate concept from content

Of the 25 components used in this study, 11 of them were built by separating concept from content. Of the 170 cases of reuse 74 used separating concept from content. Figure 36 shows the distribution of the reusability scores of the components which were built by separating concept from content. The mean reusability score is 3.4 with a standard deviation of 1.2. About 51% had either a score of 4 or 5 indicating they were easy to reuse.

Figure 37 shows a boxplot comparison of the reusability scores of components built by separating and not separating concept from content. For the group with separation of concept from content the median was 4.0 and the group without separation of concept from content had a median of 3.0.The notches of the boxplots do not overlap. Also the notch is higher for components built with this reuse design principle. This indicates that components built by separating concept from content have significantly higher reusability scores.

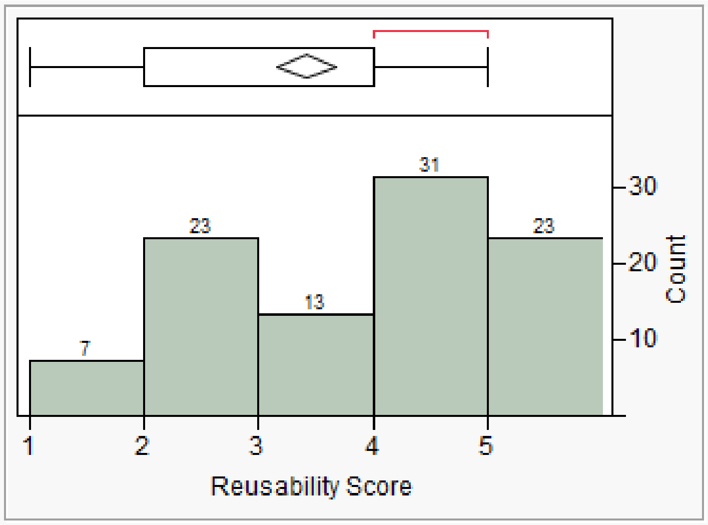


Figure 34. Distribution of the reusability scores of the components built with generality

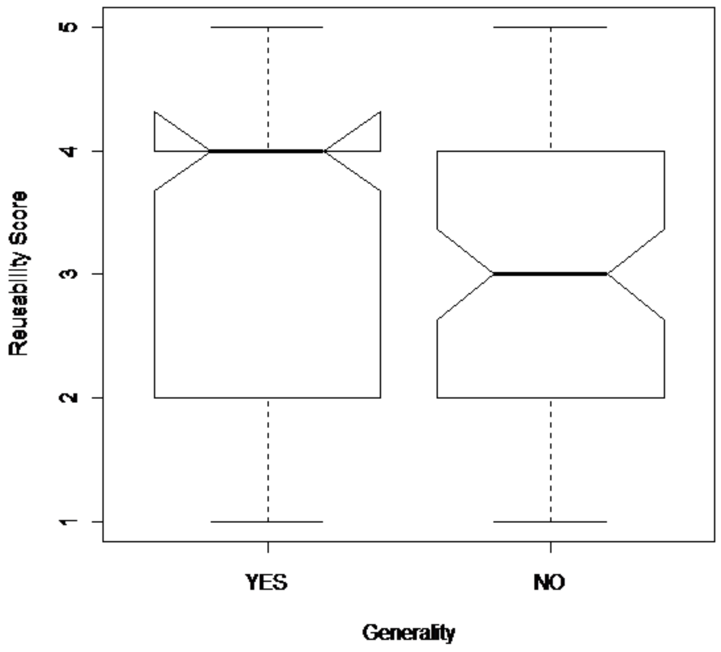


Figure 35. Box-plot comparison of the reusability scores of components with and without generality

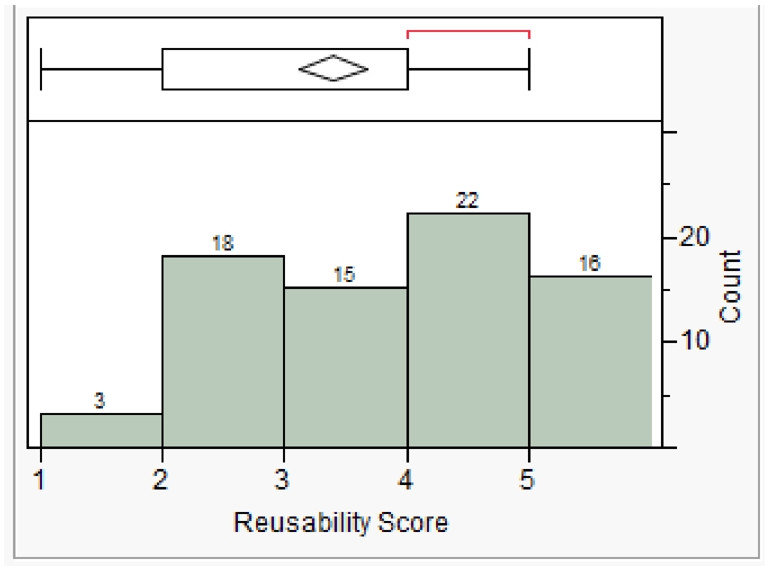


Figure 36. Distribution of the reusability scores of the components which separated concept from content

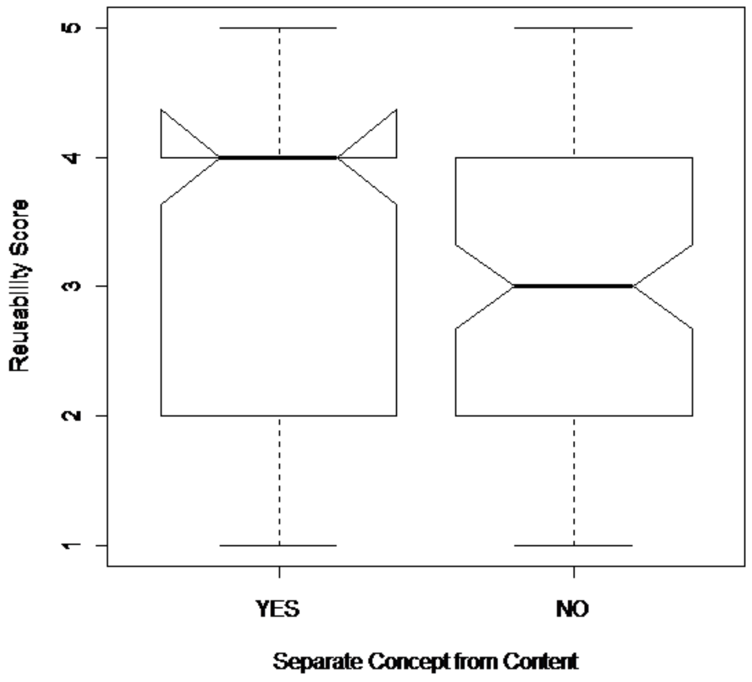


Figure 37. Box-plot comparison of the reusability scores of components with and without separated concept from content

5.4.2.6 Commonality and variability analysis

Of the 25 components used in this study, 9 of them were built by separating concept from content. Of the 170 cases of reuse 62 had commonality and variability analysis. Figure 38 shows the distribution of the reusability scores of the components which were built by analyzing commonality and variability. The mean reusability score is 2.9 with a standard deviation of 1.3. About 37% had either a score of 4 or 5 while 45% had low scores (either 2 or 1).

Figure 39 shows a boxplot comparison of the reusability scores of components built by analyzing and not analyzing commonality and variability. For the group with commonality and variability analysis the median was 3.0 and the group without commonality and variability analysis had a median of 4.0. The notches of the boxplots do not overlap. Also the notch is lower for components built with this reuse design principle. This indicates that components built by analyzing commonalities and variabilities have significantly lower reusability scores.

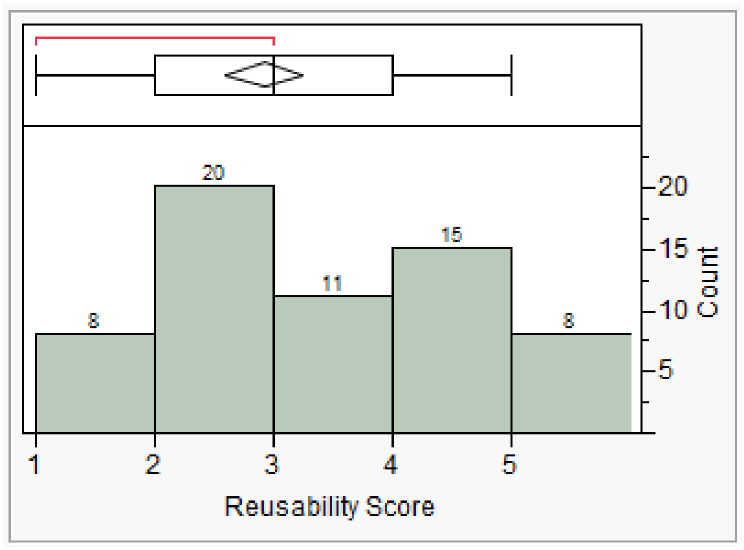


Figure 38. Distribution of the reusability scores of the components built by analyzing commonalities and variabilities

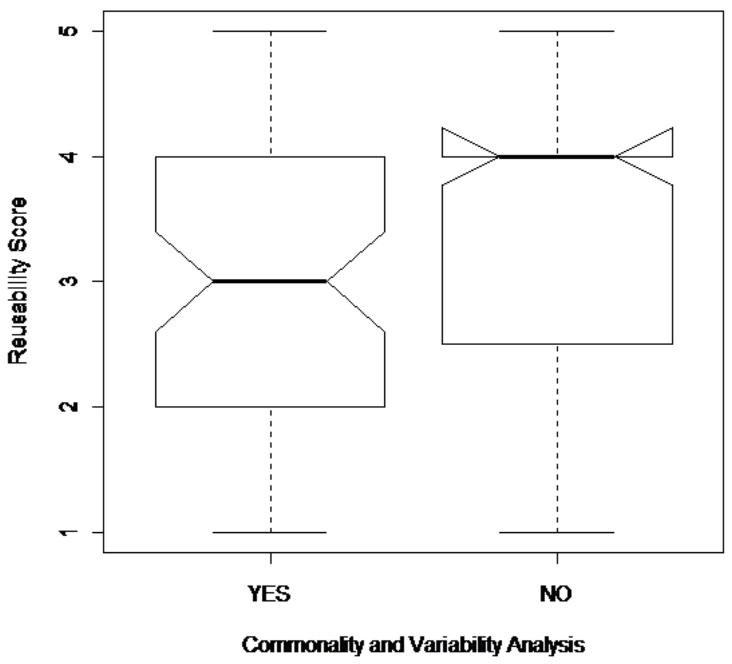


Figure 39. Box-plot comparison of the reusability scores of components with and without analysis of commonality and variabilities

From the boxplot comparisons, we see that the components had significantly higher ease of reusability scores for four of the reuse design principles – *well-defined interface, clarity and understandability, generality,* and *separate concepts from content*. The components with the design principle of *commonality and variability analysis* had a significant lower ease of reusability scores than the components without the design principle. Components with and without *documentation* had no significant difference in the reusability scores. Hence, Hypothesis II-b was confirmed for 4 reuse design principles tested and not confirmed for one principle (*documentation*). An unexpected result was that designing a component with the principle of *commonality and variability analysis* appears to make reusing the component more difficult.

5.4.3 Subject experience levels vs. reusability

For hypothesis III, the independent variables are the experience levels of the subjects in software engineering, software reuse, and Java programming language. Experience in Java is used because all the components reused in this study were developed in Java. The experiences are measured using an ordinal scale and there are six levels: None, <1yr, 1 to <2yrs, 2 to <4yrs, 4 to 8yrs, and >8yrs. The dependent variable is the reusability score and is also ordinal. It is measured using a 5-point Likert scale: (1 – not used, 2 – difficult to reuse, 3 – neither difficult nor easy to reuse, 4 – easy to reuse, 5 – very easy to reuse).

Since the dependent and independent variables are measured on ordinal scales, the statistical analysis is non-parametric. The measure of association is analyzed using Chi-Square (χ^2) and effect size is Somer's *d* [164]. For calculating the Chi-square for hypothesis II-c, there are six *rows* (levels of independent variable) and five *columns* (levels of dependent variable) in the contingency table. The degrees of freedom (*df*) is then calculated as *df* = (*rows* – 1)*(*columns* - 1). So, here *df* is (6 - 1)*(5 - 1) or 20. Chi-square indicates if the association between the dependent and independent variable is significant or not. Somer's *d* [164] is the measure of the strength of the association (effect size).

The Chi-square analysis assumes that all the cells in the contingency table have an expected value of 1 or more, and is invalid if 20% or more of the cells in the contingency table have an expected count of 5 or less (Chapter 4 of [165]). In the initial analysis this was true. So the number of *rows* was reduced to three: Low (0 to <2yrs), Medium (2 to 8yrs), and High (>8yrs). By doing so, less than 20% of the number of cells had an expected count of 5 or less, thereby making the Chi-square analysis valid, with 8 *df*.

According to Hypothesis III, more experienced programmers should find reuse to be easier. While 2 of the 3 experience variables – experience levels in software engineering and software reuse - were statistically significantly related to ease of component reuse, the relationships were not as expected. Most of the subjects found their five components to be easy to use. The unexpected result was that a slightly higher proportion of subjects with low experience levels found them easy to use than subjects with a high level of experience.

5.4.3.1 Experience levels in software engineering vs. Reusability

The contingency table for the subjects' experience in software reuse vs. the reusability scores of the components is given in Table 16. More than half of the subjects (56%) with Low experience have high reusability scores (4 or 5), indicating they found their components easy to use. The relationship was found to be significant (χ^2 with 4 *df* = 17.7, p = 0.02). However, the effect size was found to be very low (Somer's *d* = -0.007).

Table 16. Contingency table: Experience in Software Engineering vs. Reusability score (N=170)

Experience in SE	Statistic	Reusability Score				
		1	2	3	4	5
Low	Count	3	3	5	8	6
	Expected	1.8	6.3	5.7	7.4	4.9
	Row %	12.0	12.0	20.0	32.0	24.0
Medium	Count	1	10	10	2	7
	Expected	2.1	7.6	5.6	8.8	5.8
	Row %	3.3	33.3	33.3	6.7	23.3
High	Count	8	30	17	40	20
	Expected	8.1	29.1	21.6	33.8	22.3
	Row %	7.0	26.1	14.8	34.8	17.4
	TOTAL	12	43	32	50	33

5.4.3.2 Experience in Reuse vs. Reusability

The contingency table for the subjects' experience in software reuse vs. the reusability scores of the components is given in

Table 17. About half of the subjects (46.3%) with Low experience have high reusability scores (4 or 5). The relationship was found to be significant ($\chi 2$ with 4 *df* = 17.2, p = 0.03). However, the effect size was found to be very low (Somer's *d* = -0.0004).

Table 17. Contingency table: Experience in Software Reuse vs. Reusability score (N=170)

Experience in Reuse	Statistic	Reusability Score				
		1	2	3	4	5
Low	Count	6	18	19	21	16
	Expected	5.6	20.2	15.1	23.5	15.5
	Row %	7.5	22.5	23.8	26.3	20.0
Medium	Count	4	8	4	20	4
	Expected	2.8	10.1	7.5	11.8	7.8
	Row %	10.0	20.0	10.0	50.0	10.0
High	Count	2	17	9	9	13
	Expected	3.5	12.6	9.4	14.7	9.7
	Row %	4.0	34.0	18.0	18.0	26.0
	TOTAL	12	43	32	50	33

5.4.3.3 Experience in Java vs. Reusability

The contingency table for the subjects' experience in Java vs. the reusability scores of the components is given in Table 18. Half of the subjects (50%) with Low experience have high reusability scores (4 or 5). The relationship was found to be not significant ($\chi 2$ with 4 *df* = 2.09, p>0.05).

Table 18. Contingency table: Experience in Java vs. Reusability score (N=170)

Experience in Java	Statistic	Reusability Score				
		1	2	3	4	5
Low	Count	3	12	10	17	8
	Expected	3.5	12.6	9.4	14.7	9.7
	Row %	6.0	24.0	20.0	34.0	16.0
Medium	Count	5	20	16	22	17
	Expected	5.6	20.2	15.1	23.5	15.5
	Row %	6.3	25.0	20.0	27.5	21.3
High	Count	4	11	6	11	8
	Expected	2.8	10.1	7.5	11.8	7.8
	Row %	10.0	27.5	15.0	27.5	20.0
	TOTAL	12	43	32	50	33

5.4.4 Component Testing vs. Reusability

For hypothesis II-d, the independent variable is whether the component is tested before being reused (Yes or No). The dependent variable is the reusability score and is also ordinal. So, the number of rows is 2 and number of columns is 5. The *df* for Chi-square analysis is then 4. All the cells have an expected value of 1 or more. None of the cells in the contingency table has an expected value of 5 or less.

The contingency table for component testing vs. the reusability scores of the components is given in Table 18. Slightly more subjects did not test their components (52.4%) than as did test (47.6%). A similar pattern was evident in the percent of high ease of use scores (4 or 5) regardless of whether the components were not tested (53.9%) or were tested (43.2%). The relationship was not statistically significant ($\chi 2$ with 4 *df* = 4.3, p = 0.37). Hypothesis II-d was not upheld.

Table 19. Contingency table: Component testing vs. Reusability score (N=170)

Component Testing	Statistic	Reusability Score				
		1	2	3	4	5
No	Count	7	22	12	30	18
	Expected	6.3	22.5	16.8	26.2	17.3
	Row %	7.9	24.7	13.5	33.7	20.2
Yes	Count	5.0	21.0	20.0	20.0	15.0
	Expected	5.7	20.5	15.2	23.8	15.7
	Row %	6.2	25.9	24.7	24.7	18.5
	TOTAL	12	43	32	50	33

5.4.5 Content Analysis[+]

In this study, the process similar to the one in Chapter 4 is followed. The subjects in this study provided responses on the ease of reuse and the reusability score they gave for each component. Content analysis on these responses was done in 3 stages:

- Categorization: The responses were categorized based on the reuse design principles. For example, the responses for why well-defined interface was used were grouped together into one category.
- Coding: The responses within a category were then interpreted and all the different reasons were identified. Each reason was assigned a code. For example, referring to Table 19, there were 4 reasons identified when a component was not reused (reusability score of 1). They were coded as IIN (there were issues with the interface and integration), DNC (component did not compile or run), TOC (component was too complex), and BND (bad or no documentation)

[+] The responses of the subjects are presented verbatim in double quotes; the words or phrases within the square brackets were not part of the subjects' responses but have been added to improve the understanding. Also, the errors in the spelling of some words in the responses have been corrected.

- Frequency Analysis: Each response within a category was interpreted for the reasons and assigned the respective codes. The frequency of each code was then calculated as the count for the respective reason.

5.4.5.1 Why components were NOT reused

There were 12 cases when the component could not be reused by the subjects and was given the lowest reusability score of 1. One component that was assigned to 7 subjects was not used by 5 of the subjects, one gave it a score of 2 and the other gave it a score of 3. This component received the lowest average reusability score (1.43) among all the components and was designed to be reusable using only the design principle of *commonality and variability analysis*. Another component that was assigned to 7 subjects was not used by 2 of them and it had received an average score of 1.71. All the other components which received a score of 1 were not reused by one subject each. The summary of the reasons why the components were not reused are summarized in Table 20 based on the content analysis of the responses.

Table 20. Summary of content analysis for feedback of components not reused (score of 1)

Code*	Description	Count#
IIN	Issues with interface and integration	6
DNC	Did not compile/run independently	5
TOC	Too complex	1
BND	Bad or no documentation	1

*Code – two-letter code used to identify a reason (refer to section 5.5 for content analysis and coding)

The most common reason identified was that the component had issues with the interface and was not easy to integrate into the application (IIN). In an earlier work by Frakes and Fox [166], which explored the reasons for failure in reusing software life cycle objects, the second most common reason for not reusing a part was that it was not integrable into the system. The component that was not reused by 5 subjects was accepting input parameters from the command line. This caused issues with the integration. One subject, for example reasoned that, “Lack of interfaces. Design assumes console-only input and output.” The same subject also said that the lack of documentation made it difficult to reuse. Another subject said, “The reusable component can only get its input from `STDIN`. This makes it impossible to reuse the component in another application." The component that was not reused by 5 subjects is given in Appendix I. As can be seen, the component does not have a well-defined interface and accepts input only through the standard input on the command line using the `System.in` in Java.

Another common reason identified was that the component did not compile or run independently (DNC). It was claimed in 5 of the reuse cases and all were for unique components. However, for all the cases where this reason was claimed, the components were successfully compiled and used by at least four other subjects. One subject, who had developed the application in Java using the NetBeans IDE [167] in Windows 7, claimed that 2 of the components assigned did not compile. The subject had very high experience in software engineering (>8yrs) but very little experience with Java programming (1-2yrs). The same components, however, were reused by at least two other subjects in the same environment.

5.4.5.2 Why components were NOT EASILY reused

There were 75 cases of reuse which received a reusability score of 2 or 3. The subjects gave feedback for the features that made the components difficult to reuse. The content analysis is summarized in Table 21. Twelve of the feedbacks indicated more than one reason stated and they were given more than one code. One-third (25) of the feedbacks did not specify any reason and just stated that the component was not easy to reuse. This is coded as DRE.

As with the cases where the components were not reused, the top reason here is also IIN – issues with the interface and integrating the component into the system. There were 20 such cases. One subject who gave a component a reusability score of 2 said that, "The reading in the file, entering a series of words separated by commas, and the print method in the Stemmer class makes this component not reusable." Another subject said, "Because this component was designed as more of a standalone, command line Java application, it really was not a very reusable component." Of the 20 cases, 18 had received a reusability score of 2. This shows that interface issues are a great hindrance for reusing components. In a previous study [166] also, "part not integratable" was identified as a factor for failure in software reuse.

Bad or no documentation (BND) was also a reason for at least 17% of the cases. One subject said, “Commenting was sparse which made tracing code harder and lack of good supporting documentation made it more difficult.” Another subject reasoned, “No documentation provided …At a minimum, the developer should have generated Javadocs providing implementation and execution details.”

Component being too complex (TOC) was also a reason in 12% of the cases. Component was also not understandable (NUN) in 8 (11%) cases. One subject said, “The component was primarily one function with some calls to other functions. It was difficult to follow, and the user was exposed to all aspects of the component. The developer had to follow how the actual String was parsed.” There were 6 cases where it was mentioned that test cases would have improved the reusability (NTS). In 4 cases, it was said the component did not serve the specific application (DSA) where they were trying to build. For example, one subject who was developing a web-interface application mentioned that the component did not have any web service implementation and so was difficult to reuse. There were four cases where the subjects felt that the component was too general and supported more features than necessary. It has been identified in the past [166] that “part not understandable” was a failure factor for software reuse.

Table 21. Summary of content analysis for feedback of components not easily reused (reusability scores of 2 or 3)

Code*	Description	Count#
IIN	Issues with interface and integration	20
BND	Bad or no documentation	13
TOC	Component was too complex	9
NUN	Component was not understandable	8
NTS	No test cases	6
DSA	Did not serve the specific application	4
TGE	Too general	4
DRE	Difficult to reuse, no specific reason	25

*Code – two-letter code used to identify a reason (refer to section 5.5 for content analysis and coding)

5.4.5.3 Why components were EASILY reused

There were 83 cases of reuse which received a reusability score of 4 or 5. The subjects gave feedback for the features that made the components easy to reuse. The content analysis is summarized in Table 22. Twenty one of the feedbacks had more than one reason stated and they were given more than one code. Nine subjects did not specify any reason and just stated that the component was easy to reuse. This is coded as ERE.

Documentation (DOC) was the most stated reason for easy reuse of the components. It was the reason in about one-fourth of the feedbacks (25.3%). One subject stated that the component was easy to reuse because, “The Read-Me was very helpful. The code itself was also very well-commented.” Another subject stated, “Complete documentation provided with [the] Component providing information on how to implement and execute component [made the component easy to reuse] ” Javadocs also helped in easy reuse. One subject acknowledged by stating that, “The component included a Javadocs [and so for] the methods it's easy to understand their function.”

The second most stated reason was that the component was easy to compile and integrate into their applications. One feedback stated, “it was easy to use this simple Java class in a Java environment without needing to do any additional work. I simply instantiated the class and used it in my code.” Another feedback stated, “You could simply instantiate this class and call the `stemmedWord` public method, it was very straight forward [to reuse the component].” The clarity and easy understandability of the code (CLA) was the third most stated reason (13.3%). Generality was the reason in about 10.8% of the cases. One feedback stated that the component was easy to reuse because, “[the component was] obviously designed to be extensible so that more rules could be added.” Though issues with the interface were the most stated reason for the component not being reused or being difficult to reuse, a well-defined interface was not a popular reason for easy reuse.

Documentation was a popular reason for both a component being not easy to reuse as well as being easy to reuse. The components had documentation in the form of Javadocs, internal/external documentation (IE-Docs), or both. Fifteen of the 25 components had documentation – 7 had only Javadocs (44 cases of reuse), 4 had only IE-Docs (29 cases of reuse), and 4 had both Javadocs and IE-Docs (29 cases of reuse). The boxplot comparison of the reusability scores is shown in Figure 40. When only Javadocs were used, the reusability scores were significantly lower than when internal/external documentation is used. When both Javadocs and internal/external documentation was used, the reusability scores were found to be not significantly different than when only one form of documentation was used.

Table 22. Summary of content analysis for feedback of components that were easily reused (reusability scores of 4 or 5)

Code*	Description	Count#
DOC	Documentation helped easy reuse	21
INT	Easy to compile and integrate	18
CLA	Code was clear and understandable	11
GEN	Generality of the implementation logic	9
NOM	No or little modification required	9
CSI	Component is simple and not complex	7
WDI	Well-defined interface	6
MOD	Code was well modularized	5
TES	Test cases were provided	5
CON	Naming convention within the code	4
SEP	Implementation details were separated and not exposed	4
ENC	Encapsulation of data/methods	1
ERE	Easy to reuse, no specific reason	9

*Code – two-letter code used to identify a reason (refer to section 5.5 for content analysis and coding)

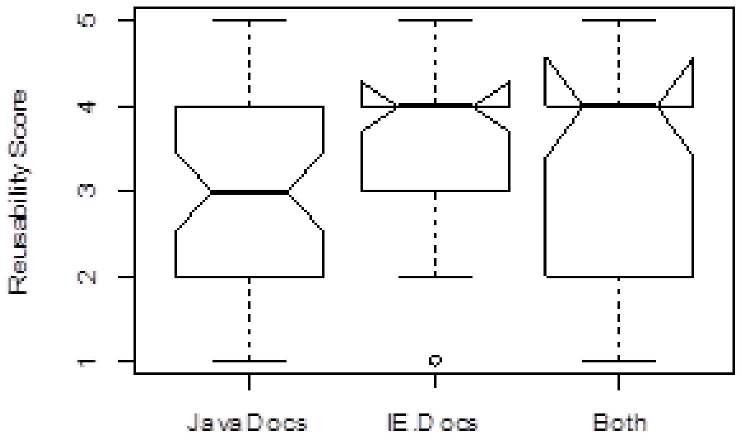


Figure 40. Comparison of reusability scores for components with only Javadocs, only internal/external documentation (IE-Docs), and those with both

5.4.6 Mahalanobis-Taguchi Strategy

The Mahalanobis-Taguchi Strategy (MTS) is a discriminatory analysis strategy for decision making. MTS is also widely used as a pattern recognition tool for various applications that deal with data classification [168, 169]. MTS is a combination of the Mahalanobis Distance (MD) and the Taguchi method using Taguchi's Orthogonal Arrays [170]. Mahalanobis Distance (MD) is a distance measure to detect and analyze patterns based on the correlation between variables [171]. MD method is used for constructing a measurement scale while the Taguchi method is used to optimize the system and make it robust by choosing the right number of parameters required for decision making. Most applications based on MTS normally differentiate the normal group from the abnormal group, for example, healthy people from unhealthy ones. The MTS methodology can also be further extended for classification within the abnormal group. This is extremely useful where there are multiple failure modes to be detected.

Mahalanobis distance (MD) is highly sensitive towards inter-variable changes in data [170]. MD is preferred over classical methods (like Euclidean Distance) because it is dependent on the variance and covariance of the data rather than its average, which makes the calculations robust. MD can be calculated for a set of any type of variable (nominal, ordinal, interval, and/or ratio). MD can be also obtained without an assumption of distribution of the variables - for proof refer to [170]. MTS can be used to identify the variables that contribute to distinguishing between normal and abnormal groups unlike traditional methods of pattern recognition like Principal Component Analysis (PCA) and Artificial Neural Networks. Also, MTS is dependent on the correlations between the variables, unlike techniques like stepwise regression which assume the variables are independent of each other.

One of the primary objectives of the Mahalanobis Taguchi Strategy (MTS) method is to introduce a scale based on all input characteristics to measure the degree of abnormality of the failure modes. To construct such a scale, Mahalanobis Distance (MD) is suitably scaled by dividing the original distance by the number of variables. The MTS is proposed as a diagnosis and forecasting method using multivariate data. Many areas of application for the MTS including inspection and sensor systems in manufacturing, fire detection, earthquake forecasting, weather forecasting, credit scoring and voice recognition [170]. There have also been case studies involving engineering applications of the MTS in many large companies including Nissan Motor, Mitsubishi Space Software, Xerox, Delphi, ITT, Ford Motor, Fuji Photo film and others [170].

In the MTS, the Mahalanobis space (reference group) is obtained using the standardized variables of healthy or normal data. The Mahalanobis space (MS) can be used to discriminate between normal and abnormal objects. Once the MS is established, the number of attributes is reduced using orthogonal arrays (OA) and signal-to-noise ratio (SN) by evaluating the contribution of each attribute. Each row of the OA determines a subset of the original system by including and excluding attributes of the system [172]. Orthogonal Arrays are discussed in depth by Taguchi in [173]. "The S/N Ratio is a measure of the functionality of the system, which exploits interaction between control factors and noise factors. A gain in S/N ratio indicates a reduction in the variability, which will result in cost savings [170]." The S/N ratio, obtained from the abnormal MDs, is used as the response for each combination of OA. The useful set of variables is obtained by evaluating the "gain" in the S/N ratio.

5.4.6.1 Construction of the Mahalanobis Space (MS)

Sample “normal” observations are used to construct a reference space, which is called the Mahalanobis space (MS). MS consists of the mean vector, standard deviation vector, and correlation matrix of the normal group [22]. Mahalanobis distance (MD) is calculated using the normalized measured variables to determine if MD has the ability to differentiate the normal group from an abnormal group.

In this study, the reuse cases with the highest reusability score of 5 (very easy to reuse) are considered as the set of “normal” observations. The measured variables are size of the component, the reuse design principles used to build the component, experience levels of the subjects (software engineering, software programming, Java programming, and software reuse), and if the component has been tested or not. The ordinal variables need to be coded for the calculation of MD. A value of 1 was given when a reuse design principle was used for a component and 0 when not used. The experience levels are coded and given the following values: 0 (no experience), 1 (0-1yr), 2 (1-2yrs), 3(2-4yrs), 4(4-8yrs), and 5 (>8yrs). A value of 1 was given when a component was tested and 0 when not tested.

A measured variable is normalized as in the equation below:

$$Z_{ij} = \frac{X_{ij} - x_i}{S_i} \tag{7}$$

where, x_i is the mean of the ith measured variable X_i ; and S_i is the standard deviation of the variable, index j means jth observation. The MD is then calculated as in the equation below:

$$MDj = \frac{1}{k} Z_{ij}^{T} C^{-1} Z_{ij} \tag{8}$$

where, C is the correlation matrix of Z (Z^T is the transpose of Z) and k is the total number of variables.

For validation of MS, observations outside the normal group are selected and respective MD values are calculated. The variables of the abnormal group are normalized using the mean and standard deviations of the corresponding characteristics in the normal group. The correlation matrix corresponding to the normal group is used to compute the MDs of the abnormal cases. If MS is suitable for the application domain with the appropriate characteristics selected, then the MDs corresponding to the abnormal group will have higher value than that of the normal group.

Four abnormal groups in this study are identified: groups with reusability scores of 1, 2, 3, and 4. Summary statistics of the MD values for the normal and abnormal groups are summarized in Table 23. The MD values for the groups are also compared using a boxplot in Figure 41. The boxplot comparison shows that the abnormal groups have significantly higher MD values than the normal group. Also, the mean MD value increases as the reusability ease decreases. This shows that as the abnormality increases (reusability ease decreases) the MD values are farther away in the Mahalanobis Space (MS). This validates that the MS constructed is valid for detecting the non-reusability of a component.

Table 23. Summary statistics of the Mahalanobis Distance (MD) values

Group	Reusability Score	Mean	Median	Std. dev.	(min, max)
Normal	5	0.97	0.91	0.41	(0.57, 1.52)
Abnormal	4	2.25	1.74	1.63	(0.32, 9.61)
	3	2.86	2.56	1.51	(0.62, 6.63)
	2	4.23	2.54	3.80	(0.82, 15.03)
	1	4.81	2.54	4.04	(1.01, 13.42)

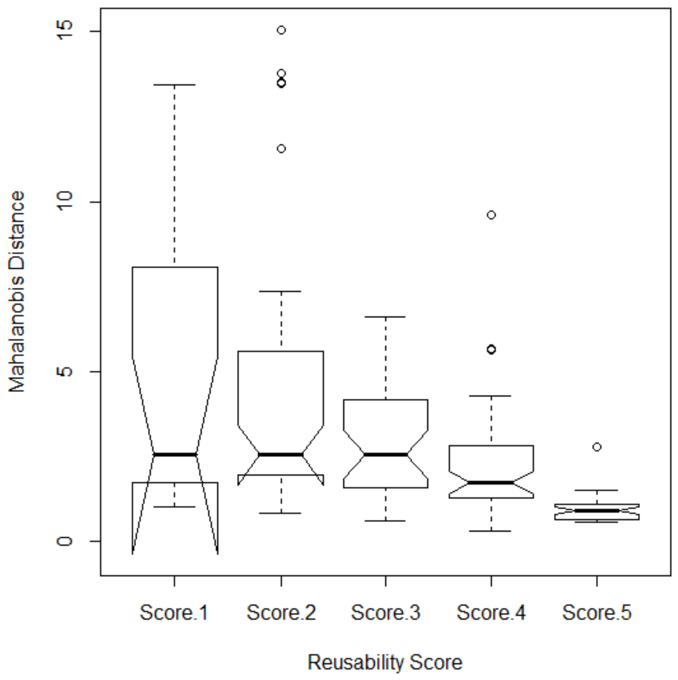


Figure 41. Comparison of Mahalanobis Distance (MD) values

5.4.6.2 Taguchi Strategy

The right set of characteristics is determined using Taguchi's orthogonal arrays (OAs) and signal-to-noise ratios (S/N). The signal-to-noise ratio, obtained from the abnormal MDs, is used as the response for each combination of OA. An orthogonal array is a table listing all the combinations of the variables. A 2-level OA to identify what variables contribute to detect the abnormality is chosen: level-1 in the orthogonal array column represents the presence of a characteristic and level-2 represents the absence of that characteristic. The size of the orthogonal array depends on the number of characteristics and the levels it can take. By varying the number of variables used, MD values are obtained for the abnormal cases and from these MD values, a larger and better signal-to-noise ratio is obtained. The S/N for the q^{th} run of t abnormal conditions may be found as in the equation below:

$$\eta_q = -10\log\left[\frac{1}{t}\right]\sum_{i=1}^{t}\frac{1}{MD_i} \tag{9}$$

The gain is then the difference between the average S/N for when the variable is present and average S/N for when the variable is not present. The S/N and the gains are given in Table 24. The gains are positive for size of the component, component testing, and for four of the reuse design principles- *generality, commonality and variability analysis, clarity and understandability,* and *well-defined interface*. The gains are negative for the reuse design principles - *documentation*, and *separate concepts from content*; and for the experience levels in software engineering and software programming. This indicates that these variables do not contribute to detecting the abnormal group based on the MS constructed from the normal group.

Table 24. S/N ratio and gain of the variables based on the Mahalanobis Space (MS)

Variable	S/N when variable present	S/N when variable absent	Gain
Size of component	57.1	35.5	21.6
Reuse Design Principle			
• Generality	47.3	45.3	2.0
• Commonality and variability	47.7	44.9	2.8
• Clear and understandable	52.9	39.6	13.3
• Well defined interface	52.9	39.7	13.2
• Documentation	41.1	51.5	*-10.4*
• Separate concept from contents	39.6	53.0	*-13.4*
Experience Level			
• Software engineering	39.1	53.5	*-14.4*
• Software programming	40.1	52.5	*-12.3*
• Software reuse	53.3	39.2	14.1
• Java programming	54.8	37.8	17.0
Component testing	47.1	45.5	1.6

5.4.7 Stepwise Regression

For further investigation, we now perform a stepwise regression analysis and compare that to the results of the MTS. The analysis was done using the tools in SAS JMP 10.1 and the final results are shown in Table 25. The threshold `p-values` for both entering and leaving a variable were set at 0.25 and 0.1 respectively (these were the default values in SAS JMP 10.1). The analysis yielded the same results for forward, backward, and mixed approach. The variables selected are the Size (*ReuseSLOC*), and the three reuse design principles – generality, commonality and variability analysis, and well-defined interface.

Table 25. Stepwise regression results

Variable	Wald/Score ChiSq	p-value
ReuseSLOC	22.22	*0.0000024*
generality	6.48	*0.01*
commonality and variability	16.36	*0.0000525*
clear and understandable	0.53	0.47
well defined interface	4.47	*0.03*
documentation	1.23	0.27
separate concept from contents	0.13	0.72
SE-experience	1.10	0.30
SP-experience	0.47	0.49
Java-Experience	0.19	0.66
Reuse-Experience	0.01	0.91
Testing	0.59	0.44

The final model is given in Table 26. As can be seen, the effect size (`RSquare` = 0.067 or 6.7%) is very small accounting for only less than 10% of the variability. Comparing it to the results from MTS, MTS had selected the same variables plus four more - clarity and understandability, software reuse experience, java programming experience, and component testing. Stepwise regression may not have selected theses variable due to the high correlations between the variables. The regression equation for the final model is:

$$Reusability = [0.012 * (ReuseSLOC) + 0.4 * (generality) + 0.64 * (commonality\ and\ variability) + 0.31 * (well\text{-}defined\ interface) + 0.41] \quad (10)$$

Table 26. Stepwise regression final model

Variable	ChiSquare	p-value	RSquare
ReuseSLOC	12.60	0.0004	0.024
commonality and variability	9.82	0.0017	0.043
generality	7.98	0.0047	0.059
well defined interface	4.48	0.0343	0.067

5.4.8 Threats to validity

The threats to validity for this study are presented based on the discussion in Chapter 6 of the book by Wohlin et al. [161].

5.4.8.1 Threats to Construct Validity

The dependent variable, reusability score of a component, in the study is not based on an objective measurement. However, a user of the component is the best judge and so the reusability ease of a component is measured as perceived by the user in integrating the component to the system. It is assumed that the higher the experience, the higher the expertise. Also, the subjects were given the source code as components and so had the choice to modify them if required. They had to reuse all the 5 components allocated to them. The reuse design principles that the subjects in the previous study in Chapter 4 attributed to the components claimed to exhibit was based on the evaluation by the course grader.

5.4.8.2 Threats to Internal Validity

The subjects were reusing components as part of an assignment in a graduate course at a University. It was made clear to the subjects that not being able to reuse a component would not negatively affect their grade in the assignment. The subjects were also instructed to freely choose any software platform and packages for doing their assignments. These instructions should have made the users concentrate only on the reuse aspect of the assignment. Also, the subjects could have interacted with each other. To mitigate this, no two subjects were given the same set of components to be reused. The subjects had to finish the task in 2 weeks and this time constraint has not been taken into consideration as the assignment is fairly simple and the components reused are small and simple. However, this is still a threat to internal validity.

5.4.8.3 Threats to External Validity

The issue of using students as subjects in software engineering experiments has been discussed in the past [24-28] and there has been mixed results on whether students could provide the same results as using professionals. However, the students considered in those studies were full-time students. Most of the subjects in this study are working professionals with varying experiences in the software industry and enrolled as part-time students at the University. Half of the subjects had more than 8 years of experience in programming as well as in the field of software engineering. Only 1 subject mentioned having no experience in software programming. Carver et al. [27] have mentioned that the gap between students and novice professionals are decreasing especially in the context of the US educational climate.

The study is limited only to reuse of components written in Java. Future studies will involve components in other languages as well. The components implement a simple s-stemming algorithm. The components are not very complex. Future studies will involve components implementing more complex algorithms.

Chapter 6: Summary and Conclusions

Though many reuse design principles have been proposed, there is no generally accepted list of reuse design principles which are language and domain independent. So the literatures of software reuse and reuse design over the past four decades have been analyzed, and provided a generic list of reuse design principles for component based software development as given in Chapter 3. These principles can be a guideline for designing and building reusable components and are language and domain independent. A critical evaluation of the reuse design principles is also presented and discussed. New structures are provided for the reuse design principles through a mindmap and a cross-reference table. Future work may involve exploring new structures for the reuse design principles based on the reasons for using them as well as measuring the reuse design principles for reusable components.

Two studies are presented, one based on design *for* reuse and the other based on design *with* reuse.

6.1 Design *for* Reuse

One hundred and seven implementations of a one-use stemmer component and the equivalent reusable programs were analyzed in this exploratory study. The subjects first built the one-use stemmer component after which 19 reuse design principles were taught to them. The subjects then converted their one-use components to equivalent reusable components and were asked to indicate which reuse use design principles they used. A ranking of the design principles by frequency of use is reported. The six most frequently used reuse design principles were – *well-defined interface, documentation, clarity and understandability, generality, separate concepts from contents,* and *commonality and variability analysis*. The reuse design principles of *isolation of context and policy from functionality, abstraction,* and *self-documenting code* were least used by the subjects. This may be because the components developed were relatively simple.

The subjects provided feedback on why they chose the design principles they used. A content analysis performed on the feedback is also reported. The most common reason identified for using a reuse design principles was that it allowed and improved the ease of implementing changes a future user might want. The second most common reason identified was that the developer wanted to hide the implementation details of the component from the user. Another reason that was commonly stated was that the reuse design principle improved the understanding of the code and the logic, which would in turn encourage the reuse of the component.

The subjects also provided feedback on the design principles which they considered but did not use. A content analysis performed on this feedback was also reported. The principle that was considered the most and then not used was *genericity*. All but one of the subjects argued that they did not apply this design principle because the component required manipulation of only one type of data. The most common reason why a reuse design principle was not used is that the component being built is simple and not complex enough to warrant an implementation of the reuse design principle.

The correlation analysis shows that, in general, the reuse design principles were used independently of each other.

One-use and the equivalent reusable components were analyzed using measures of the pairs in terms of size in SLOC (source lines of code), effort in hours, number of parameters, and productivity as measured by SLOC/hours to develop. Reusable components were significantly larger than their equivalent one-use components and had significantly more parameters. The effort required for the reusable components was higher than for one-use components. The productivity of the developers was significantly lower for the reusable components compared to the one-use components. This may be because of more code within the component to realize the additional functionality for reusability. Also, during the development of reusable components, the subjects spent more time on writing code than designing the components but not significantly so.

Future work may include an empirical study using a more complex algorithm. Additionally, future work may include components built in various other languages.

6.2 Design *with* Reuse

Through an empirical study, some factors were analyzed that affect the ease of reusing software components. Reusability of a component is measured as the ease of reuse as perceived by the subjects reusing the component. Thirty-four subjects participated in the study with each subject reusing 5 components, resulting in 170 cases of reuse. The components were randomly assigned to the subjects from a pool of 25 components which were designed and built for reuse.

The relationship between the complexity of a component (as measured by SLOC) and the ease of reuse was analyzed by a regression analysis. It was observed that the higher the complexity the lower the ease of reuse, but the correlation was not significant. An analysis of the relationship between a set of reuse design principles, used in designing and building the components, and the ease of reuse is also reported. When considered independently, four of the reuse design principles: *well-defined interface, clarity and understandability, generality,* and *separate concepts from content* significantly increased the ease of reuse while and *commonality and variability analysis* significantly decreased the ease of reuse, and *documentation* did not have a significant impact on the ease of reuse.

The human factors studied were the experience levels of the user in software reuse, software engineering, and the experience levels with a programming language. Experience in the programming language had no relationship with the reusability of components. Experience in software engineering and software reuse showed a relationship with reusability but the effect size was not significant. Testing components before integrating them into a system was also found to have no relationship with the reusability of components.

The subjects had also provided feedback on the reusability of the components. A content analysis of the feedback is presented identifying the challenges of components that were not easy to reuse. *Bad interface* was identified as the most important factor of components not being reused easily. Features that make a component easily reusable are also identified. Documentation was a popular reason for both the component being not easy to reuse as well as being easy to reuse. When only Javadocs were used, the reusability scores are significantly lower than when internal/external documentation is used.

The Mahalanobis-Taguchi Strategy (MTS) was employed to develop a model based on Mahalanobis Distance to identify the factors that can predict if a component is easy to reuse or not. The identified factors within the model are: size of a component, a set of reuse design principles (*well-defined interface, clarity and understandability, commonality and variability analysis,* and *generality*), and component testing.

Realizing that the components in the studies were relatively small in size and only in Java, similar studies may also be done in future for larger reusable components and in other languages as well. The impact of the development environment, OS, and programming language on the ease of reusing code components may also be studied in future.

6.3 Recommendation for Research

The recommendations for future research based on the results from this dissertation are summarized below:

- The method and empirical approach followed in this dissertation is an important contribution to the field. They can be replicated and extended to other environments and programming languages.

- The components studied in this dissertation are small in size and implement a simple stemming algorithm. Larger components may provide other challenges which need to be explored. Reuse can be done for any size components. COTS (Commercially Off The Shelf) integration projects typically involve use of large reusable components and a survey study has been reported on the challenges in integration [174]. Higher levels of expertise in software development may be required to develop large reusable components as well as to reuse them. Application of the reuse design principles may also be challenging for larger components.

- One-use and equivalent reusable components are compared based on size, effort, parameters, and productivity. These factors may be used for developing or updating cost estimation models.

- Documentation [16, 17, 78, 84, 86-88] has been identified as a key factor that affected the success of reuse. The effect of the quality of documentation on the ease of reusing components also needs to be further explored.

6.4 Recommendations for Practice

The recommendations for practice based on the results from this dissertation are summarized below:

- Various other phenomena such as quality control, software safety, and defect reduction in the software reuse field may also be explored based on the method and empirical approach followed in this dissertation.

- Studies similar to the ones presented in this dissertation may be integrated into industrial projects to study other aspects of component development and reuse.

- The list of reuse design principles used in this dissertation is a result of a review of the literature over the past four decades. The most commonly used reuse design principles used are also identified. They can be a guideline for training software engineers.

- Reengineering simple one-use components to be reusable was studied through an empirical study and the reusable components compared based on size, effort, parameters, and productivity. The results may be extended for larger components and other programming languages.

- Experience levels of programmers affected the ease of reuse. Programmers with higher levels of experience may be considered by managers for projects involving building or using reusable components. Larger components may pose other challenges and need to be explored through empirical studies.

6.5 Publications

Peer-reviewed - Given below is a list of the peer-reviewed publications that was achieved related to the dissertation:

- Reuse Design Principles. Reghu Anguswamy* and William B. Frakes, in *International Workshop on Designing Reusable Components and Measuring Reusability, DReMeR '13* held in conjunction with the 13th International Conference on Software Reuse, ICSR '13, Pisa, Italy, 18 June, 2013

This paper was related to the material in Chapter 3. Each of the 19 reuse design principles presented in Chapter in 3 was briefly introduced. During the presentation it was commented that such a list is very relevant and useful for even for large-scale systems.

- An Exploratory Study of One-Use and Reusable Software Components. Reghu Anguswamy* and William B. Frakes, *International Conference on Software Engineering and Knowledge Engineering SEKE '12,* San Jose, USA, 1-3 July, 2012, pp. 194-199

This paper was related to the study in Chapter 4. Hypotheses I-a through I-d and the related results were presented. The reviewers of the paper, in general agreed that the experiment as solid and valid. However, they also wanted to see results for larger components.

- Effect of Complexity and Reuse Design Principles on Reusable Components. Reghu Anguswamy* and William B. Frakes, *International Conference on Empirical Software Engineering and Measurement ESEM '12,* Lund, Sweden, Sep 19-20, 2012

This paper was related to the study in Chapter 5. Hypotheses II-a&b and the related results were presented. The paper was well received and the reviewers appreciated the approach in general. During the presentation at the conference, the threats to validity were further discussed. That helped make that section in the dissertation more solid.

- A Study of Factors Affecting the Design and Use of Reusable Components. Reghu Anguswamy* and William B. Frakes, *International Doctoral Symposium on Empirical Software Engineering (IDoESE'12)*, Lund, Sweden, Sep. 21, 2012

This paper was related to the dissertation in general. The method and approach along with some preliminary results were presented. The panel present during the presentation included experts in empirical software engineering. They agreed that such studies may easily be extended to the industry. However, some of them commented that the study must be done for larger components and in other languages to achieve generalizability. This has been discussed as a recommendation for future research in section 6.3.

- Reuse Ratio Metrics RL and RF. William B. Frakes, Reghu Anguswamy* and Suvelee Sarpotdar, *11th International Conference on Software Reuse, ICSR 11, Tools and Demos*, Falls Church, VA, USA, Sep. 27-30, 2009

This paper revisited the reuse ratio metrics RL (reuse level) and RF (reuse frequency) [175]. During the demo section various implementations of RL and RF tools for measuring the amount of reuse in software were presented.

- A Comparative Study of One-Use and Reusable Software Components. Reghu Anguswamy* and William B. Frakes, IEEE Software (in progress)

The complete study and results from Chapter 4 will be presented.

- Software Reusability and Mahalanobis-Taguchi Strategy. Reghu Anguswamy* and William B. Frakes, *Journal of Systems and Software* (in progress)

The complete study and results from Chapter 5 will be presented.

Technical Presentations (not peer-reviewed) - Given below is a list of technical presentations that was achieved related to the dissertation:

- Reusable Components and Reuse Design Principles. Reghu Anguswamy (Advisor: Dr. William B. Frakes), presented via online to *ESDS-SRWG: Earth Science Data Systems Software Reuse Working Group*, Mar 21, 2012

The ESDS-SRWG has experts in the software reuse industry. The empirical approach and preliminary results of the study in Chapter 4 were presented. The group in general was of the view that results from such empirical studies could be good recommendations industrial practice.

- Mahalanobis-Taguchi Strategy for Software Safety. Reghu Anguswamy* and William B. Frakes, *International Workshop on Software Reuse and Safety (RESAFE 2009)*, Falls Church, VA, Sep. 18, 2009

The Mahalanobis-Taguchi Strategy (MTS) was presented. The application for the strategy in other fields including manufacturing was also briefly presented. It was suggested that such an approach may be applicable in software engineering and safety as well.

Other Publications (peer-reviewed) - Given below is a list of peer-reviewed publications that was achieved during the course of the PhD program at the University but not related to the dissertation:

- Consistency among Domain Analysts in Selecting Domain Documents and Creating Vocabularies. Nemmallapudi C., Frakes W. B., and Anguswamy R.* *13th International Conference on Software Reuse ICSR '13,* Pisa, Italy. Jun 19-20, 2013

- Using Program Profilers for Reusable Component Optimization and Indexing. Anguswamy R.* and Frakes W. B., *International Workshop on Designing Reusable Components and Measuring Reusability, DReMeR '13* held in conjunction with the 13th International Conference on Software Reuse, ICSR '13, Pisa, Italy. Jun 18, 2013

- A Study of COTS Integration Projects: Product Characteristics, Organization, and Life Cycle Models. Megas K., Frakes W. B., Urbano J., Belli G., and Anguswamy R.* (2013). *28th ACM Symposium on Applied Computing, SAC '13.* Coimbra, Portugal. Mar 18-22, 2013

- A Comparison of Database Fault Detection Capabilities Using Mutation Testing. McCormick II D. W., Frakes W. B., and Anguswamy R.* (2012). *The 6th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM'12.* Lund, Sweden, Sep 19-20, 2012

- Evaluation of a Computer Support-based Cross Discipline Research Consortium. Anguswamy R.* and Frakes W. B. (2011). CSEDU 2011 - Proceedings of the Third International Conference on Computer Supported Education. Amsterdam, Netherlands. May 6-8, 2011

- Computer Support for a Cross-discipline Research Methods Consortium. Frakes W. B., Belli G., Urbano J., and Anguswamy R.* (2010). 2nd International Conference on Computer Supported Education - CSEDU 2010. Valencia, Spain. Apr 7-10, 2010

References

[1] W. B. Frakes and K. C. Kang, "Software reuse research: status and future," *IEEE Transactions on Software Engineering*, vol. 31, pp. 529-536, 2005.

[2] W. Frakes and C. Terry, "Software reuse: metrics and models," *ACM Computing Surveys*, vol. 28, pp. 415-435, 1996.

[3] R. van Ommering, "Software reuse in product populations," *IEEE Transactions on Software Engineering*, vol. 31, pp. 537-550, 2005.

[4] P. Mohagheghi and R. Conradi, "An empirical investigation of software reuse benefits in a large telecom product," *ACM Transactions on Software Engineering Methodology,* vol. 17, pp. 1-31, 2008.

[5] W. B. Frakes and G. Succi, "An industrial study of reuse, quality, and productivity," *Journal of Systems and Software*, vol. 57, pp. 99-106, 2001.

[6] M. Morisio, M. Ezran, and C. Tully, "Success and Failure Factors in Software Reuse," *IEEE Transactions on Software Engineering,* vol. 28, pp. 340-357, 2002.

[7] W. C. Lim, "Effects of Reuse on Quality, Productivity, and Economics," *IEEE Software*, vol. 11, pp. 23-30, 1994.

[8] A. Paul, B. Cornelia, N. David, and R. Stephen, "Adaptive Reuse of Libre Software Systems for Supporting On-line Collaboration," *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 1-4, July 2005.

[9] S. Morad and T. Kuflik, "Conventional and open source software reuse at Orbotech - an industrial experience," in *Proceedings. IEEE International Conference on Software - Science, Technology and Engineering*, Herzelia, Israel, 2005, pp. 110-117.

[10] H. Nakano, Z. Mao, K. Periyasamy, and W. Zhe, "An Empirical Study on Software Reuse," in *International Conference on Computer Science and Software Engineering*, Hubei, China, 2008, pp. 509-512.

[11] M. Ramachandran and W. Fleischer, "Design for large scale software reuse: an industrial case study," in *Proceedings of Fourth International Conference on Software Reuse, ICSR '96*, Orlando, FL, 1996, pp. 104-111.

[12] D. Lucrédio, K. dos Santos Brito, A. Alvaro, V. C. Garcia, E. S. de Almeida, R. P. de Mattos Fortes, and S. L. Meira, "Software reuse: The Brazilian industry scenario," *Journal of Systems and Software*, vol. 81, pp. 996-1013, 2008.

[13] W. B. Frakes, "Stemming Algorithms," in *Information Retrieval: Data Structures and Algorithms*. vol. 77, W. B. Frakes and R. Baeza-Yates, Eds., 2nd ed Englewood Cliffs, NJ: Prentice-Hall. , 1998, pp. 131-160.

[14] W. B. Frakes and D. Lea, "Design for Reuse and Object Oriented reuse Methods," presented at the Sixth Annual Workshop on Institutionalizing Software Reuse (WISR '93), Owego, NY, 1993.

[15] J. Sametinger, *Software engineering with reusable components*. Berlin Heidelberg, Germany: Springer Verlag, 1997.

[16] B. Weide, W. Ogden, and S. Zweben, "Reusable software components," *Advances in computers,* vol. 33, pp. 1-65, 1991.

[17] M. Ezran, M. Morisio, and C. Tully, *Practical software reuse: the essential guide*. London: Springer Verlag, 2002.

[18] M. Ramachandran, "Software reuse guidelines," *ACM SIGSOFT Software Engineering Notes,* vol. 30, pp. 1-8, 2005.

[19] W. B. Frakes and M. Tortorella, "Foundational Issues in Software Reuse and Reliability," Department of Industrial and Systems Engineering, Rutgers University, Working Paper#04-002, Rutgers University2008.

[20] R. Seepold and A. Kunzmann, *Reuse techniques for VLSI design*. Netherlands: Springer 1999.

[21] B. Boehm, C. Abts, A. W. Brown, S. Chulani, B. K. Clark, E. Horowitz, R. Madachy, D. J. Reifer, and B. Steece, "Cost estimation with COCOMO II," ed: Upper Saddle River, NJ: Prentice-Hall, 2000.

[22] P. Vitharana, "Risks and challenges of component-based software development," *Communincations of the ACM,* vol. 46, pp. 67-72, 2003.

[23] W. B. Frakes and T. P. Pole, "An empirical study of representation methods for reusable software components," *IEEE Transactions on Software Engineering,* vol. 20, pp. 617-630, 1994.

[24] B. Curtis, "Measurement and experimentation in software engineering," *Proceedings of the IEEE,* vol. 68, pp. 1144-1157, 1980.

[25] P. Runeson, "Using students as experiment subjects–an analysis on graduate and freshmen student data," in *Empirical Assessment in Software Engineering*, 2003, pp. 95-102.

[26] D. I. K. Sjoeberg, J. E. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanovic, N. K. Liborg, and A. C. Rekdal, "A survey of controlled experiments in software engineering," *Software Engineering, IEEE Transactions on,* vol. 31, pp. 733-753, 2005.

[27] J. Carver, L. Jaccheri, S. Morasca, and F. Shull, "Issues in using students in empirical studies in software engineering education," in *Software Metrics Symposium, 2003. Proceedings. Ninth International*, 2003, pp. 239-249.

[28] B. Curtis, "By the way, did anyone study any real programmers?," presented at the Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers, Washington, D.C., USA, 1986.

[29] R. Bousley, "Reusable avionics executive software," in *NAECON 1981, Proceedings of the IEEE 1981 National Aerospace and Electronics Conference*, Dayton, OH, 1981, pp. 31-36.

[30] M. J. Cavaliere, "Reusable code at the Hartford Insurance Group," in *Software reusability,* ed: ACM, 1989, pp. 131-141.

[31] B. A. Burton, R. W. Aragon, S. A. Bailey, K. D. Koehler, and L. A. Mayes, "The Reusable Software Library," *IEEE Software,* vol. 4, pp. 25-33, 1987.

[32] R. W. Selby, "Quantitative studies of software reuse," in *Software reusability,* ed: ACM, 1989, pp. 213-233.

[33] R. G. Lanergan and C. A. Grasso, "Software Engineering with Reusable Designs and Code," *IEEE Transactions on Software Engineering,* vol. SE-10, pp. 498-501, 1984.

[34] D. Bauer, "A reusable parts center [Technical forum]," *IBM Systems Journal,* vol. 32, pp. 620-624, 1993.

[35] A. Endres, "Lessons learned in an industrial software lab (software development)," *IEEE Software,* vol. 10, pp. 58-61, 1993.

[36] J. R. Tirso and H. Gregorius, "Technical forum: management of reuse at IBM," *IBM Systems Journal,* vol. 32, pp. 612-615, 1993.

[37] M. L. Griss, "Software reuse experience at Hewlett-Packard," in *Proceedings of the 16th International Conference on Software Engineering, ICSE-16,* 1994, p. 270.

[38] M. L. Griss, "Making reuse work at Hewlett-Packard," *IEEE Software,* vol. 12, pp. 105-107, 1995.

[39] R. Joos, "Software reuse at Motorola," *IEEE Software,* vol. 11, pp. 42-47, 1994.

[40] E. Mambella, R. Ferrari, F. D. Carli, and A. L. Surdo, "An integrated approach to software reuse practice," *SIGSOFT Software Engineering Notes,* vol. 20, pp. 63-80, 1995.

[41] M. Morisio, C. Tully, and M. Ezran, "Diversity in reuse processes," *IEEE Software,* vol. 17, pp. 56-63, 2000.

[42] M. Matsumoto, A. Hayano, T. Kudo, H. Yoshida, S. Imai, and K. Ohshima, "Specifications reuse process modeling and case study-based evaluations," in *Proceedings of the Fifteenth Annual International Computer Software and Applications Conference, COMPSAC '91., ,* 1991, pp. 499-506.

[43] S. Isoda, "Experience Report On Software Reuse Project: Its Structure, Activities, And Statistical Results," in *International Conference on Software Engineering*, 1992, pp. 320-326.

[44] R. Prieto-Diaz, "Implementing faceted classification for software reuse," *Communications of the ACM - Special issue on software engineering,* vol. 34, pp. 88-97, May 1991.

[45] A. Tomer, L. Goldin, T. Kuflik, E. Kimchi, and S. R. Schach, "Evaluating software reuse alternatives: a model and its application to an industrial case study," *IEEE Transactions on Software Engineering,* vol. 30, pp. 601-612, 2004.

[46] C. A. Mattmann, R. R. Downs, J. J. Marshall, N. F. Most, and S. Samadi, "Reuse of software assets for the NASA Earth science decadal survey missions," in *Geoscience and Remote Sensing Symposium (IGARSS), 2010 IEEE International,* 2010, pp. 1687-1690.

[47] V. Kotov, "Reuse in Software Development Organizations in Latvia," *Scientific Journal of Riga Technical University. Computer Sciences,* vol. 41, pp. 90-96, 2010.

[48] W. B. Frakes, T. J. Biggerstaff, R. Prieto-Diaz, K. Matsumura, and W. Schaefer, "Software reuse: is it delivering?," in *Proceedings of the 13th International Conference on Software Engineering*, Austin, Texas, United States, 1991, pp. 52-59.

[49] W. Chen, J. Li, J. Ma, R. Conradi, J. Ji, and C. Liu, "An empirical study on software development with open source components in the chinese software industry," *Software Process: Improvement and Practice,* vol. 13, pp. 89-100, 2008.

[50] M. D. McIlroy, J. Buxton, P. Naur, and B. Randell, "Mass produced software components," *Software Engineering Concepts and Techniques,* pp. 88–98, 1969.

[51] W. Kozaczynski and G. Booch, "Component-based software engineering," *IEEE Software,* vol. 15, pp. 34-36, 1998.

[52] P. Herzum and O. Sims, *Business Components Factory: A Comprehensive Overview of Component-Based Development for the Enterprise*: John Wiley & Sons, Inc., 2000.

[53] J. Hopkins, "Component primer," *Communications of the ACM,* vol. 43, pp. 27-30, 2000.

[54] C. Szyperski, D. Gruntz, and S. Murer, *Component software: beyond object-oriented programming*: Addison-Wesley Professional, 2002.

[55] W. Hasselbring, "Component-based software engineering," *Handbook of Software Engineering and Knowledge Engineering,* vol. 2, pp. 289-305, 2002.

[56] G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, J. Conallen, and K. A. Houston, *Object-Oriented Analysis and Design with Applications,* 3rd ed. Boston, MA: Addison-Wesley Professional, 2007.

[57] B. Coulange, *Software reuse*: Springer, 1998.

[58] M. L. Griss, "Software reuse: Objects and frameworks are not enough," *Object Magazine,* pp. 77-87, Feb. 1995.

[59] I. Crnkovic and M. P. H. Larsson, *Building reliable component-based software systems*: Artech House Publishers, 2002.

[60] E. Gamma, *Design patterns: elements of reusable object-oriented software*: Addison-Wesley Professional, 1995.

[61] L. Latour, T. Wheeler, and B. Frakes, "Descriptive and predictive aspects of the 3Cs model: SETA1 working group summary," 1991, pp. 9-17.

[62] I. Crnkovic and M. Larsson, "Challenges of component-based development," *Journal of Systems and Software,* vol. 61, pp. 201-212, 2002.

[63] S. Mahmood, R. Lai, and Y. S. Kim, "Survey of component-based software development," *Software, IET,* vol. 1, pp. 57-66, 2007.

[64] I. Crnkovic, "Component-based software engineering — new challenges in software development," *Software Focus,* vol. 2, pp. 127-133, 2001.

[65] G. T. Heineman and W. T. Councill, *Component-based software engineering: putting the pieces together*: Addison-Wesley Professional, 2001.

[66] A. W. Brown and K. C. Wallnan, "Engineering of component-based systems," in *Proceedings of Second IEEE International Conference on Engineering of Complex Computer Systems*, 1996, pp. 414-422.

[67] A. W. Brown and K. C. Wallnau, "The current state of CBSE," *IEEE Software,* vol. 15, pp. 37-46, 1998.

[68] ComputerHope. (2013, 19 May). *Linux and Unix wc command help and examples*. Available: http://www.computerhope.com/unix/uwc.htm

[69] C. McClure, *Software reuse techniques: adding reuse to the system development process*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997.

[70] E. Leinfuss, "Managing Class Libraries Takes Discipline," *Software Magazine-Westborough,* vol. 13, pp. 15-15, 1993.

[71] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes, *Object-oriented development*. Englewood Cliffs, N.J: Prentice Hall, 1994.

[72] I. Jacobson, M. Griss, and P. Jonsson, *Software reuse: architecture, process and organization for business success*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1997.

[73] B. Stroustrup, "Language-technical aspects of reuse," in *Proceedings of Fourth International Conference on Software Reuse*, 1996, pp. 11-19.

[74] M. Ramachandran, "Automated improvement for component reuse," *Software Process: Improvement and Practice,* vol. 11, pp. 591-599, 2006.

[75] B. Liskov and S. Zilles, "Programming with abstract data types," *SIGPLAN Notes,* vol. 9, pp. 50-59, 1974.

[76] J. Sodhi and P. Sodhi, *Software reuse: domain analysis and design processes*. New York, NY, USA: McGraw-Hill, Inc. , 1998.

[77] C. W. Krueger, "Software reuse," *ACM Computing Surveys,* vol. 24, pp. 131-183, 1992.

[78] R. Leach, *Software Reuse: Methods, Models and Costs*. New York, NY, USA: McGraw-Hill, Inc. , 1996.

[79] G. Booch, *Software components with Ada: Structures, tools, and subsystems*: Benjamin/Cummings Pub. Co., 1987.

[80] J. M. Neighbors, "Draco: A method for engineering reusable software systems," *Software reusability,* vol. 1, pp. 295-319, 1989.

[81] D. L. Parnas, P. C. Clements, and D. M. Weiss, "Enhancing reusability with information hiding," in *Software reusability: vol. 1, concepts and models,* ed: ACM, 1989, pp. 141-157.

[82] T. A. Standish, "An Essay on Software Reuse," *IEEE Transactions on Software Engineering,* vol. SE-10, pp. 494-497, 1984.

[83] P. Wegner, "Capital-intensive software technology," *IEEE Software,* pp. 7-10, 1984.

[84] B. Liskov and J. Guttag, *Abstraction and specification in program development*. Cambridge, MA, USA: The MIT Press, McGraw-Hill Book Company, 1986.

[85] IEEE, "IEEE Standard Glossary of Software Engineering Terminology," in *IEEE Std 610.12-1990*, ed: IEEE Press, 1990, pp. 1-84.

[86] C. Braun, "Making Software Reusable," presented at the GTE Workshop, GTE Federal Systems Division, 1993.

[87] E. Karlsson, *Software reuse: a holistic approach*. New York, NY, USA: John Wiley & Sons, Inc. , 1995.

[88] Y. Matsumoto, "Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels," *IEEE Transactions on Software Engineering,* vol. SE-10, pp. 502-513, 1984.

[89] M. Ramachandran and P. Allen, "Commonality and variability analysis in industrial practice for product line improvement," *Software Process: Improvement and Practice,* vol. 10, pp. 31-40, 2005.

[90] J. Coplien, D. Hoffman, and D. Weiss, "Commonality and variability in software engineering," *IEEE Software,* vol. 15, pp. 37-45, 1998.

[91] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Report#CMU/SEI-90-TR-211990.

[92] R. Prieto-Diaz and J. M. Neighbors, "Module interconnection languages," *Journal of Systems and Software,* vol. 6, pp. 307-334, 1986.

[93] F. DeRemer and H. H. Kron, "Programming-in-the-Large Versus Programming-in-the-Small," *IEEE Transactions on Software Engineering,* vol. SE-2, pp. 80-86, 1976.

[94] M. Shaw and D. Garlan, *Software architectures. Perspectives on an emerging discipline*. Upper Saddle River,NJ: Prentice-Hall, Inc., 1996.

[95] W. B. Frakes and B. A. Nejmeh, "An information system for software reuse," in *Software reuse: emerging technology,* T. Will, Ed., ed: IEEE Computer Society Press, 1988, pp. 142-151.

[96] A. Snyder, "Encapsulation and inheritance in object-oriented programming languages," *ACM SIGPLAN Notices,* vol. 21, pp. 38-45, 1986.

[97] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communincations of the ACM,* vol. 15, pp. 1053-1058, 1972.

[98] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, *Object-oriented software engineering: a use case driven approach*. New York, NY: Addison-Wesley, 1992.

[99] J. Blustein. (1996). *An Example of Data Encapsulation in C*. Available: http://www.csd.uwo.ca/~jamie/C/encapsulatedC.html (Accessed: 25 May 2012)

[100] J. Kienzle, "On exceptions and the software development life cycle," presented at the Proceedings of the 4th International Workshop on Exception Handling, Atlanta, Georgia, 2008.

[101] J. W. Hooper and R. O. Chester, *Software reuse: guidelines and methods*. New York and London: Plenum Publishing Corporation, 1991.

[102] B. Meyer, "Reusability: The Case for Object-Oriented Design," *IEEE Software*, vol. 4, pp. 50-64, 1987.

[103] B. Stroustrup, "Parameterized types for C++," *Journal of Object Oriented Program.*, vol. 1, pp. 5-16, 1989.

[104] B. Meyer, *Eiffel: the language*. Upper Saddle River, NJ: Prentice-Hall, Inc., 1992.

[105] D. Ghosh, "Generics in Java and C++: a comparative model," *ACM SIGPLAN Notices*, vol. 39, pp. 40-47, 2004.

[106] M. Sloman, "Policy driven management for distributed systems," *Journal of Network and Systems Management*, vol. 2, pp. 333-360, 1994/12/01 1994.

[107] T. Elrad, R. E. Filman, and A. Bader, "Aspect-oriented programming: Introduction," *Communications of the ACM*, vol. 44, pp. 29-32, 2001.

[108] M. Aoyama, "New age of software development: How component-based software engineering changes the way of software development," presented at the paper presented at the 1998 International Workshop on Component-Based Software Engineering, Kyoto, Japan, 1998.

[109] Microsoft. (2013, May 19). *What is Active X | What is an Active X Control - Microsoft*. Available: http://www.microsoft.com/security/resources/activex-whatis.aspx

[110] Oracle. (2013, May 19). *Trail: JavaBeans(TM) (The Java™ Tutorials)*. Available: http://docs.oracle.com/javase/tutorial/javabeans/

[111] K. Beck, "Simple Smalltalk Testing: With Patterns," October 1994.

[112] D. Brenner, C. Atkinson, R. Malaka, M. Merdes, B. Paech, and D. Suliman, "Reducing verification effort in component-based software engineering through built-in testing," *Information Systems Frontiers*, vol. 9, pp. 151-162, 2007/07/01 2007.

[113] H.-G. Gross, C. Atkinson, and F. Barbier, "Component Integration through Built-in Contract Testing," in *Component-Based Software Quality*, A. Cechich, P. Mario, and V. Antonio, Eds., ed New York: Springer-Verlag New York, Inc., 2003, pp. 159-183.

[114] H.-G. Gross, *Component-based software testing with UML*. Germany: Springer-Verlag Berlin Heidelberg 2005.

[115] K. Beck, *Test-driven development: by example*: Addison-Wesley Professional, 2003.

[116] Microsoft. (2012). *Chapter 10: Component Guidelines*. Available: http://msdn.microsoft.com/en-us/library/ee658121.aspx (Accessed: 25 May, 2012)

[117] S. McConnell, *Code Complete: A Practical Handbook of Software Construction*: Microsoft Press, 2004.

[118] D. E. Knuth, "An empirical study of FORTRAN programs," *Software: Practice and Experience*, vol. 1, pp. 105-133, 1971.

[119] B. W. Boehm, "Industrial software metrics top 10 list," *IEEE Software*, vol. 4, pp. 84-85, 1987.

[120] J. Bentley, "More programming pearls. Confessions of a coder," *Reading: Addison-Wesley, 1988*, vol. 1, 1988.

[121] W. B. Frakes, "A case study of a reusable component collection in the information retrieval domain," *Journal of Systems and Software*, vol. 72, pp. 265-270, 2004.

[122] W. B. Frakes, M. Pittkin, and R. Anguswamy, "Using Program Profilers for Reusable Component Optimization and Indexing," in *International Workshop on Designing Reusable Components and Measuring Reusability, DReMeR '13 held in conjunction with the 13th International Conference on Software Reuse, ICSR '13*, Pisa, Italy, 2013.

[123] J. Fenlason and R. Stallman, "GNU gprof: the GNU profiler," *Manual, Free Software Foundation Inc*, 1997.

[124] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: a call graph execution profiler," *ACM SIGPLAN Notices*, vol. 39, pp. 49-57, 2004.

[125] J. L. Bentley, *Writing efficient programs*: Prentice-Hall, Inc., 1982.

[126] J. Lamping, "A unified system of parameterization for programming languages," presented at the Proceedings of the 1988 ACM conference on LISP and functional programming, Snowbird, Utah, USA, 1988.

[127] W. B. Frakes, C. J. Fox, B. A. Nejmeh, A. Telephone, and T. Company, *Software engineering in the UNIX/C environment*. Englewood Cliffs, NJ: Prentice Hall, Inc., 1991.

[128] S. Sheppard, M. Borst, B. Curtis, and L. Love, "Predicting Programmers' Ability to Modify Software," General Electric Company, DTIC Document#TR 78-388100-3 May 1978.

[129] J. Raskin, "Comments are More Important than Code," *Queue*, vol. 3, pp. 64-65, 2005.

[130] B. Stroustrup, *The C++ Programming Language*, 3rd ed. Reading, MA: Addison-Wesley Professional, 2000.

[131] F. Puhlmann, A. Schnieders, J. Weiland, and M. Weske, "Variability Mechanisms for Process Models," June 30 2005.

[132] T. Martínez-Ruiz, F. García, and M. Piattini, "Towards a SPEM v2.0 Extension to Define Process Lines Variability Mechanisms Software Engineering Research, Management and Applications," in *Software Engineering Research, Management and Applications Studies in Computational Intelligence*. vol. 150, R. Lee, Ed., ed: Springer Berlin / Heidelberg, 2008, pp. 115-130.

[133] S. Meyers, "The most important design guideline? [user interfaces]," *IEEE Software*, vol. 21, pp. 14-16, 2004.

[134] D. Harman, "How effective is suffixing?," *Journal of the American Society for Information Sciences*, vol. 42, pp. 7-15, 1991.

[135] B. Boehm, C. Abts, and S. Chulani, "Software development cost estimation approaches — A survey," *Annals of Software Engineering*, vol. 10, pp. 177-205, 2000.

[136] B. W. Boehm, *Software engineering economics*. Upper Saddle River, NJ: Prentice-Hall, 1981.

[137] L. H. Putnam and W. Myers, *Measures for excellence*: Yourdon Press, 1992.

[138] R. Jensen, "An improved macrolevel software development resource estimation model," in *5th ISPA Conference*, 1983, pp. 88–92.

[139] R. W. Selby, "Enabling reuse-based software development of large-scale systems," *IEEE Transactions on Software Engineering*, vol. 31, pp. 495-510, 2005.

[140] T. Tan, Q. Li, B. Boehm, Y. Yang, M. He, and R. Moazeni, "Productivity trends in incremental and iterative software development," in *ESEM '09 Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement* Lake Buena Vista, Florida, USA, 2009, pp. 1-10.

[141] A. Gupta, "The profile of software changes in reused vs. non-reused industrial software systems," Doctoral Thesis, NTNU, Singapore, 2009.

[142] N. E. Fenton and M. Neil, "Software metrics: roadmap," presented at the Proceedings of the Conference on The Future of Software Engineering, Limerick, Ireland, 2000.

[143] M. Dinsoreanu and I. Ignat, "A Pragmatic Analysis Model for Software Reuse," in *Software Engineering Research, Management and Applications 2009*. vol. 253, R. Lee and N. Ishii, Eds., ed: Springer Berlin / Heidelberg, 2009, pp. 217-227.

[144] I. Herraiz and A. E. Hassan, "Beyond Lines of Code: Do We Need More Complexity Metrics?," in *Making Software: What Really Works, and Why We Believe It*, A. Oram and G. Wilson, Eds., 1st ed Sebastapol, CA: O' Reilly Media, Inc. , 2010, pp. 125-141.

[145] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. SE-2, pp. 308-320, 1976.

[146] S. H. Kan, *Metrics and models in software quality engineering*. India: Pearson Education India, 2003.

[147] J. Graylin, J. E. Hale, R. K. Smith, D. Hale, N. A. Kraft, and C. Ward, "Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship," *Journal of Software Engineering and Applications*, p. 137, 2009.

[148] R. P. L. Buse and W. R. Weimer, "Learning a Metric for Code Readability," *IEEE Transactions on Software Engineering*, vol. 36, pp. 546-558, 2010.

[149] J. E. Gaffney, "Estimating the Number of Faults in Code," *IEEE Transactions on Software Engineering*, vol. SE-10, pp. 459-464, 1984.

[150] M. S. Krishnan and M. I. Kellner, "Measuring process consistency: implications for reducing software defects," *IEEE Transactions on Software Engineering*, vol. 25, pp. 800-815, 1999.

[151] K. Krippendorff, *Content analysis: an introduction to its methodology*: Sage Publications, Inc, 2004.

[152] M. Niazi, D. Wilson, and D. Zowghi, "Critical success factors for software process improvement implementation: an empirical study," *Software Process: Improvement and Practice,* vol. 11, pp. 193-211, 2006.

[153] N. Baddoo, "Motivators and de-motivators in software process improvement: an empirical study," University of Hertfordshire, 2001.

[154] N. Baddoo and T. Hall, "De-motivators for software process improvement: an analysis of practitioners' views," *Journal of Systems and Software,* vol. 66, pp. 23-33, 2003.

[155] N. Baddoo and T. Hall, "Motivators of Software Process Improvement: an analysis of practitioners' views," *Journal of Systems and Software*, vol. 62, pp. 85-96, 2002.

[156] Oracle. (2013, May 19). *Javadoc Tool Home Page - Oracle*. Available: http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html

[157] D. M. Weiss. (2013, May 19, 2013). *Defining Families: The Commonality Analysis*. Available: http://pdf.aminer.org/000/361/295/defining_families_commonality_analysis.pdf

[158] W. B. Frakes and C. J. Fox, "Sixteen questions about software reuse," *Communications of the ACM,* vol. 38, pp. 75-ff., 1995.

[159] D. A. Wheeler. (2013, May 19). *SLOCCount*. Available: http://www.dwheeler.com/sloccount/

[160] J. Cohen, *Statistical power analysis for the behavioral sciences*, 2nd ed. Hillsdale, NJ: Lawrence Erlbaum Associates, Inc., 1988.

[161] C. Wohlin, P. Runeson, and M. Host, *Experimentation in software engineering: an introduction* vol. 6. Boston: Springer, 2000.

[162] S. R. Nidumolu and G. W. Knotts, "The effects of customizability and reusability on perceived process and competitive performance of software firms," *MIS Quarterly,* vol. 22, pp. 105-137, 1998.

[163] R. Anguswamy and W. B. Frakes, "An Exploratory Study of One-Use and Reusable Software Components," in *Proceedings of International Conference of Software Engineering and Knowledge Engineering, SEKE'12,* San Francisco, USA, 2012, pp. 194-199.

[164] R. H. Somers, "A new asymmetric measure of association for ordinal variables," *American Sociological Review,* pp. 799-811, 1962.

[165] G. Salvendy, *Handbook of human factors and ergonomics*, 4th ed. Hobokes, New Jersey: John Wiley & Sons, Inc., 2012.

[166] W. B. Frakes and C. J. Fox, "Quality improvement using a software reuse failure modes model," *IEEE Transactions on Software Engineering,* vol. 22, pp. 274-279, 1996.

[167] Oracle. (2013, May 19). *Welcome to NetBeans*. Available: https://netbeans.org/

[168] R. B. Chinnam, B. Rai, and N. Singh, "Tool-condition monitoring from degradation signals using Mahalanobis-Taguchi system analysis," in *Proceedings of ASI's 20th Annual Symposium of Robust Engineering*, 2004, pp. 343-351.

[169] H. C. Wang, C. C. Chiu, and C. T. Su, "Data classification using the Mahalanobis-Taguchi system," *Journal of the Chinese Institute of Industrial Engineers*, vol. 21, pp. 606-618, 2004.

[170] G. Taguchi and R. Jugulum, *The Mahalanobis-Taguchi strategy: a pattern technology system*: John Wiley & Sons, 2002.

[171] P. C. Mahalanobis, "On the generalized distance in statistics," in *Proceedings of the National Institute of Sciences of India*, 1936, pp. 49-55.

[172] A. K. Jain, R. P. W. Duin, and J. Mao, "Statistical pattern recognition: A review," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 22, pp. 4-37, 2000.

[173] G. Taguchi, *System of Experimental Design, Vols. 1 and 2*. Dearborn, Michigan, and White Plains, New York: ASI Press and UNIPUB-Kraus International Publications, 1987.

[174] K. Megas, W. B. Frakes, J. Urbano, G. Belli, and R. Anguswamy, "A Study of COTS Integration Projects: Product Characteristics, Organization, and Life Cycle Models.," presented at the 28th ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, 2013.

[175] W. B. Frakes, "An Empirical Framework for Software Reuse Research," Syracuse University CASE Center Technical Report, no. 9014, 1990.

Appendix A: Software Reuse – Expert Opinion Survey

Dear Colleague

This questionnaire is intended to capture your certain personal views in designing reusable components. Your responses are strictly confidential and only summary statistics from all the respondents will be reported.

The questionnaire takes typically 10-15mins to be completed. The contact information is OPTIONAL and will not be shared; it is intended ONLY for further correspondence/clarification.

If you have any concerns/questions/issues please contact:

Reghu Anguswamy
Software Reuse Lab., Virginia Tech.
#311, 7054 Haycock Rd.
Falls Church, VA - 22043
email: reghu@vt.edu

1. Below are some statements regarding one-use components and reusable components. For a brief description on these components please copy and paste this link in your browser:

http://rmc.ncr.vt.edu/one-use-component-vs-reusable-component

Based on your experience and knowledge please state if the following statements are TRUE or FALSE or DON'T KNOW

One-use components will be smaller than the reusable components
TRUE FALSE DON'T KNOW

Reusable components require higher effort to be built compared to its equivalent one-use components

TRUE FALSE DON'T KNOW

Reusable components will have more parameters than its equivalent one-use components

TRUE FALSE DON'T KNOW

Productivity i.e., number of lines of code written per hour will be higher when building one-use components

TRUE FALSE DON'T KNOW

Any additional comments:

……………………………………………………………………………………………………

……………………………………………………………………………………………………

……………………………………………………………………………………………………

2. Below is a list of design principles (arranged in alphabetic order) that are used to make components reusable. For a brief description of the design principles, please copy and paste the link below in your web browser:

http://rmc.ncr.vt.edu/reuse-design-principles

Please choose at least 5 design principles that you would use most to design and build to make a component reusable.

☐ abstraction
☐ clear and understandable
☐ commonality and variability
☐ composition
☐ encapsulation
☐ generality

- ☐ genericity
- ☐ isolate context and policy
- ☐ link to documentation
- ☐ linking of test to code
- ☐ modification
- ☐ one component use many
- ☐ optimization
- ☐ parameterization
- ☐ restrictiveness
- ☐ self-documenting code
- ☐ separate concept from contents
- ☐ variability mechanism
- ☐ well defined interface

3. Are there any other reuse design principles not in the above list?

- ○ Yes
- ○ No

If yes (please specify):

……
……
……

4. What is your highest educational qualification degree?

- ○ UnderGraduate
- ○ Master's
- ○ PhD
- [✓ Other (please specify):] ☐

5. How many years of experience do you have in the field of software engineering?

0yrs 0-1yr 1-2yrs 2-4yrs 4-8yrs >8yrs

6. What is your role in your organization?

Developer/Programmer

Software Architect

Systems Engineer

Manager

Do Not Wish to Reveal

[✓ Other (please specify):]

7. How many years of experience do you have in software programming?

0yrs 0-1yr 1-2yrs 2-4yrs 4-8yrs >8yrs

8. How many years of experience do you have in software reuse?

0yrs 0-1yr 1-2yrs 2-4yrs 4-8yrs >8yrs

9. Have you been trained to design and build software components for reuse?

Yes No

If yes (please specify):

…………………………………………………………………………………………………
…………………………………………………………………………………………………
…………………………………………………………………………………………………

Contact Information (OPTIONAL)

Questions in this section are optional, you may provide only the details you wish to. The contact information provided will be strictly confidential and is intended only for further clarification/correspondence.

10. Your contact information:

Name:	
Company:	
Country:	
Email Address:	

11. Any additional comments (optional) ?

Appendix B: Demographics Survey (for Chapter 4)

Dear Colleague

Thank you very much for agreeing to take part in this survey. This questionnaire is for students who were part of CS 5744 (Software Quality and Design), Fall 2009 at Virginia Tech. National Capital Region, by Prof. William B. Frakes.

This questionnaire is about your demographics and your role in your organization. Only summary statistics of the data collected will be reported. If you have any concerns or questions, please contact Reghu Anguswamy at reghu@vt.edu

Thanks and Regards

Reghu Anguswamy
Software Reuse Lab., Virginia Tech.
http://www.cs.vt.edu/node/698
#311, 7054 Haycock Rd.
Falls Church, VA, USA - 22043
email: reghu@vt.edu

At the beginning of Fall 2009, what was your highest educational qualification degree?

○ UnderGraduate

○ Master's

○ PhD

[✓ Other (please specify):] ____

At the beginning of semester, how many years of experience did you have in the field of software engineering?

0yrs 0-1yr 1-2yrs 2-4yrs 4-8yrs >8yrs

At the beginning of semester, what was your role in your organization?

Developer/Programmer

Software Architect

Systems Engineer

Manager

Do Not Wish to Reveal

[✓ Other (please specify):]

At the beginning of semester, how many years of experience did you have in software programming and coding?

0yrs 0-1yr 1-2yrs 2-4yrs 4-8yrs >8yrs

At the beginning of semester, how many years of experience did you have working in Java?

0yrs 0-1yr 1-2yrs 2-4yrs 4-8yrs >8yrs

During the semester, did you have a software reuse program in your organization?

Yes No

At the beginning of semester, how many years of experience did you have in software reuse?

0yrs 0-1yr 1-2yrs 2-4yrs 4-8yrs >8yrs

Appendix C: Demographics Survey (for Chapter 5)

This questionnaire is for students in CS 5744, Fall 2011 at Virginia Tech. National Capital Region. This questionnaire is about your demographics and your role in your organization. Personal details will be confidential and only summary statistics of the data collected will be reported.
If you have any concerns or questions, please contact Reghu Anguswamy at reghu@vt.edu

Thanks and Regards
Reghu Anguswamy (GA - CS 5744, Fall 2011)

1. Contact information

Name:	
Email Address (vt.edu):	
Student ID number:	

2. What is your highest educational qualification degree?

○ What is your highest educational qualification degree? Undergraduate

○ Master's

○ Doctoral

○ Other (please specify)

3. How many years of experience do you have in the field of software engineering?

○ 0yrs ○ 0-1yr ○ 1-2yrs ○ 2-4yrs ○ 4-8yrs ○ >8yrs

4. What is your role in your organization?

☐ Developer/Programmer

☐ Software Architect

☐ Systems Engineer

☐ Manager

☐ Do Not Wish to Reveal

☐ Other (please specify)

5. How many years of experience do you have in software programming?

None

0-1yr

1-2yrs

2-4yrs

4-8yrs

>8yrs

6. How many years have you worked in the following languages?

	None	**0-1yr**	**1-2yrs**	**2-4yrs**	**4-8yrs**	**>8yrs**
C						
C++						
C#						
Java						

Other (please specify)

7. How many years of experience do you have in software reuse?

○ 0yrs ○ 0-1yr ○ 1-2yrs ○ 2-4yrs ○ 4-8yrs ○ >8yrs

8. Have you been trained for designing software components for reuse?

○ Yes ○ No

If yes, please give details:

9. Do you have a software reuse program in your organization?

○ Yes ○ No

If yes, please give details of the program:

Appendix D: Component Reuse Survey – Chapter 5

This is the survey questionnaire for Assignment 2 in CS574-Fall 2011 class at Virginia tech. National Capital Region. First page is the general student information. Then, it is followed by 5 pages, one for each component you have reused. Personal information will be confidential and only the summary statistics will be reported.

If you have any questions or concerns, please contact me.

Thanks

-Reghu (reghu@vt.edu)

1. Enter you student information.

Name:	
VT PID	
VT Student ID number:	

2. Please provide details of your development environment (like the language used, OS, IDE, etc...)

COMPONENT A/B/C/D/E

3. How do you rate the reusability of the component on a scale of 1-5?

○ 1 - not used ○ 2 - difficult to use ○ 3 - neither easy nor difficult ○ 4 - easy to use ○ 5 - very easy to use

4. Did you have to modify the component code to use?

○ Yes ○ No

5. If YES to above question, how much of the code did you have to modify?

○ 0-20% ○ 21-40% ○ 41-60% ○ 61-80% ○ 81-100%

6. Did you test the component independently before integrating it in your application?

○ Yes ○ No

7. How much time did you spend in reusing the component?

○ 0-15min ○ 15-30min ○ 30-45min ○ 45min-1hr ○ 1-2hrs ○ >2hrs

8. How do you rate the quality of the component?

○ 1 - very low ○ 2 - low ○ 3 - moderate ○ 4 - high ○ 5 - very high

9. What design features make the component reusable?

10. What design features make the component NOT reusable?

11. Did you have any personal challenges in using the component?

☐ None

- ☐ Lack of knowledge in Java
- ☐ Lack of programming experience
- ☐ Lack of resources
- ☐ Lack of time
- ☐ Lack of experience in code reuse
- ☐ Other (please specify)

Appendix E: Code Example 1

One-Use Component

```java
import java.util.Scanner;

public class Stemming {

    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        String tmp;
        boolean stem;

        do{
               stem = false;
        System.out.print("Enter a word to find its stem: ");
        tmp = input.next();

        if (tmp.endsWith("ies")){
            if (!((tmp.endsWith("eies")) || (tmp.endsWith("aies")))){
                tmp = tmp.substring(0, tmp.length() - 3);
                tmp = tmp + "y";
                System.out.println("Your word stem is: " + tmp);
                stem = true;
            }
        }

        if ((tmp.endsWith("es")) && (stem == false) && (!(tmp.equals("exit")))){
            if (!((tmp.endsWith("aes")) || (tmp.endsWith("ees"))|| (tmp.endsWith("oes")))){
                tmp = tmp.substring(0, tmp.length() - 2);
                tmp = tmp + "e";
                System.out.println("Your word stem is: " + tmp);
                stem = true;
            }
        }

        if ((tmp.endsWith("s")) && (stem == false) && (!(tmp.equals("exit")))){
            if (!((tmp.endsWith("us")) || (tmp.endsWith("ss")))){
                tmp = tmp.substring(0, tmp.length() - 1);
                System.out.println("Your word stem is: " + tmp);
                stem = true;
            }
        }

        if ((stem == false) && (!(tmp.equals("exit"))))
             System.out.println("This word is either already a stem or has no stem: " + tmp);

        } while (!(tmp.equals("exit")));

    }
}
```

Reusable Component: 2 Java files (`Stemmer.java` and `StemmerTest.java`)

```
//Stemmer.java
```

```
public class Stemmer {

    private String TrimString(String tmp, int numChars){
         try{
             return tmp.substring(0, tmp.length()-numChars);
         }
         catch (Exception e){
             return "";
         }
     }

    private char GetCharFromEnd(String tmp, int numChar){
         try{
             return tmp.charAt(tmp.length()-numChar);
         }
         catch (Exception e){
             return ' ';
         }
     }

     public String getStem(String tmp) {
         boolean stem = false;

```

```
            if (GetCharFromEnd(tmp, 1) == 's'){
                switch(GetCharFromEnd(tmp, 2)) {
                    case 'u':
                        stem = false;
                        break;
                    case 's':
                        stem = false;
                        break;
                    case 'e':
                        switch(GetCharFromEnd(tmp, 3)){
                            case 'a':
                                stem = false;
                                break;
                            case 'e':
                                stem = false;
                                break;
                            case 'o':
                                stem = false;
                                break;
                            case 'i':
                                switch(GetCharFromEnd(tmp, 4)){
                                    case 'a':
                                        stem=false;
                                        break;
                                    case 'e':
                                        stem=false;
                                        break;
                                    default:
                                        stem=true;
                                        return "Your word stem is: " + TrimString(tmp, 3) + "y";
                                }
                                break;
                            default:
                                stem=true;
                                return "Your word stem is: " + TrimString(tmp, 2) + "e";
                        }
                        break;
                    default:
                        stem=true;
                        return "Your word stem is: " + TrimString(tmp, 1);
                }
            }
            else{
                stem = false;
            }

            if (stem == false)
                 return "There is no stem for the word: " + tmp;
            else
                return "Stem has already been returned";
        }
    }
```

```
//StemmerTest.java
```

```
public class StemmerTest {

    Stemmer s;

    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        String tmp;
        Stemmer s = new Stemmer();

        do{
            System.out.print("Enter a word to find its stem: ");
            tmp = input.next();

            if (!(tmp.equals("exit"))){
                System.out.println(s.getStem(tmp));
            }

        } while (!(tmp.equals("exit")));

    }
}
```

Appendix F: Code Example 2

One-use Component

```
public class SStemmer {
    public static void main(String[] args) {
        System.out.println("S Stemmer");
        System.out.println("Enter a single word to stem and press return.");
        System.out.println("Type quit to quit.");

        Scanner in = new Scanner(System.in);

        System.out.print("Enter word to stem: > ");

        while(in.hasNextLine()) {
            String input = in.nextLine();

            if(input.equalsIgnoreCase("quit")) return;

            System.out.println(input + " -> " + stem(input));
            System.out.println();
            System.out.print("Enter word to stem: > ");

        }
    }
    public static String stem(String s) {
        String result = s;

        if(s.endsWith("ies") && !(s.endsWith("eies") ||
s.endsWith("aies"))) {
            result = s.substring(0, s.length()-3) + "y";
        }
        else if(s.endsWith("es") && !(s.endsWith("aes") ||
s.endsWith("ees") || s.endsWith("oes"))) {
            result = s.substring(0, s.length()-2) + "e";
        }
        else if(s.endsWith("s") && !(s.endsWith("us") ||
s.endsWith("ss"))) {
            result = s.substring(0, s.length()-1);
        }

        return result;
    }
}
```

Reusable Component: 3 files (`Main.java`, `Stemmer.java`, and `StemmingRule.java`)

```
//Main.java
```

```
package cs5744.stemmer;

import java.util.Scanner;

/**
 * Main method used to test the stemmer class
 */
public class Main {
```

```java
    public static void main(String[] args) {
        Stemmer sStemmer = initializeSStemmer();

        System.out.println("S Stemmer");
        System.out.println("Enter a single word to stem and press return.");
        System.out.println("Type quit to quit.");

        Scanner in = new Scanner(System.in);

        System.out.print("Enter word to stem: > ");

        while(in.hasNextLine()) {
            String input = in.nextLine();

            if(input.equalsIgnoreCase("quit")) return;

            System.out.println(input + " -> " + sStemmer.stem(input));
            System.out.println();
            System.out.print("Enter word to stem: > ");

        }
    }


    /**
     * initializeSStemmer
     * Creates a class to stem various forms of plurality from a word.
     *
     * This stemmer uses the following rules, in the order shown:
     * If a word ends in ies□ but not eies□ or aies□
     *   Change the ies□ to y□
     * If a word ends in es□ but not aes□, ees□, or oes□
     *   Change the es□ to e□
     * If a word ends in s□, but not us□ or ss□
     *
     * @return a Stemmer class configured to stem "s" words
     */
    public static Stemmer initializeSStemmer() {
        Stemmer stemmer = new Stemmer();

        StemmingRule rule = new StemmingRule("ies", new String[] {"eies", "aies"}, "y");
        stemmer.addRule(rule);

        rule = new StemmingRule("es", new String[] {"aes", "ees", "oes"}, "e");
        stemmer.addRule(rule);

        rule = new StemmingRule("s", new String[] {"us", "ss"}, "");
        stemmer.addRule(rule);

        return stemmer;
    }
}
```

//Stemmer.java

```java
package cs5744.stemmer;

import java.util.ArrayList;

/**
 * A class to stem words based on StemmingRules.
 * StemmingRules return null if they do not fire, and the stemmed word if they do.
 *
 * The default implementation of Stemmer will return the value of the first rule
 * that does not return null.
 *
 */
public class Stemmer {
    private ArrayList<StemmingRule> rules;

    /**
     * Default Constructor.
     */
    public Stemmer() {
        rules = new ArrayList<StemmingRule>();
    }

    /**
     * Add a rule to fire.  Rules are fired in the order
     * in which they are added.
     * @param rule the StemmingRule to fire
     */
    public void addRule(StemmingRule rule) {
        rules.add(rule);
    }

    /**
     * Stem a word.
     * Loops through the StemmingRules and returns
     * the stemmed word.
     * @param s -- word to stem
     * @return the resulting stemmed word
     */
    public String stem(String s) {
        String result = null;

        for(StemmingRule rule : rules) {
            result = rule.stem(s);
            if(result != null) {
                break;
            }
        }

        return (result == null) ? s : result;
    }
}
```

```
//StemmingRule.java
```

```
package cs5744.stemmer;

/**
 * StemmingRule
 * Performs basic stemming operation on a String.
 * To configure rule, use the constructor to define
 * 1) The stem -- the part of the word that you wish to change
 * 2) An array of exceptions -- endings of the word that preclude this rule's firing.
 * 3) What to change the stem to.
 *
 * If the need for multiple stemming rules is deemed necessary,
 * you might want to create an interface for all stemming rules
 * instead of a concrete class.
 *
 */
public class StemmingRule {
    private String stem;
    private String[] exceptions;
    private String changeTo;

    /**
     * Constructor for the Stemming Rule.
     *
     * @param stem -- the stem, at the end of the word, that will cause this rule to fire.
     * @param exceptions -- a list of stems that should cause this word to NOT fire.
     * @param changeTo -- the value to change the stem to if this rule does fire.
     */
    public StemmingRule(String stem, String[] exceptions, String changeTo) {
        this.stem = stem;
        this.exceptions = exceptions;
        this.changeTo = changeTo;
    }

    /**
     * Stem a word.
     * @param s -- the word to stem.
     * @return null, if the rule does not fire.  The stemmed word, if it does.
     */
    public String stem(String s) {
        // method will return empty string if no stemming
        // occurs
        String result = null;

        // if the string is a candidate to stem
        if(s.endsWith(stem)) {
            boolean doStem = true;

            // make sure it doesn't match any exceptions
            for(int i = 0; i< exceptions.length; i++) {
                if(s.endsWith(exceptions[i])) {
                    doStem = false;
                    break;
                }
            }
            // do the stemming, if no exceptions met
            if(doStem) {
                result = s.substring(0, s.length()-stem.length()) + changeTo;
            }
        }
        return result;
    }
}
```

Appendix G: Code Example 3

One-Use Component

```
package assn1;

import java.util.Scanner;

/**
 *
 */
public class Assn1 {

    public Assn1() {
    }

    private static String endsInS(String word) {
        String returnValue;
        returnValue = word.toLowerCase();

        if ((returnValue.endsWith("ies")) && (!(returnValue.endsWith("eies")) && !(returnValue.endsWith("aies")))) {
            returnValue = returnValue.substring(0,returnValue.length()-3).concat("y");
        } else if ((returnValue.endsWith("es")) && !(returnValue.endsWith("aes")) && !(returnValue.endsWith("ees")) && !(returnValue.endsWith("oes"))) {
            returnValue = returnValue.substring(0,returnValue.length()-1);
        } else if ((returnValue.endsWith("s")) &&
!(returnValue.endsWith("us")) && !(returnValue.endsWith("ss"))) {
            returnValue = returnValue.substring(0,returnValue.length()-1);
        }

        return returnValue;
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        String input;
        String output;

        do {
            Scanner in = new Scanner(System.in);

            input = in.nextLine();

            if (input.length() > 0) {
                output = endsInS(input);
                if (input.equals(output)) {
                    System.out.println("No S Stem was found for " + input);
                } else {
                    System.out.println("The S Stem for " + input + " is " + output);
                }
            }
        } while (input.length() > 0);
    }
}
```

Reusable Component: 2 files (`S_Stemmer.java` and `SuffixStemmer.java`)

```
//S_Stemmer.java
```

```
package assignment3.Stemmers;

/**
 *
 */
public class S_Stemmer extends SuffixStemmer{

    /**
     * Executes algorithm for Stemming word which ends in 's'
     *
     *@param String word Word to be stemmed according to rules for words ending in 's'
     */
    private static String stemSuffix(String word) {
        String returnValue;
        String[] exclusionListIES = new String[] {"eies", "aies"};
        String[] exclusionListES = new String[] {"aes", "ees", "oes"};
        String[] exclusionListS = new String[] {"us",
"ss"};

        returnValue = word.toLowerCase();
        if (EndsWith(returnValue, "ies", exclusionListIES)) {
            returnValue = RemoveSuffix(returnValue, "ies", "y");
        } else if (EndsWith(returnValue, "es", exclusionListES)) {
            returnValue = RemoveSuffix(returnValue, "s");
        } else if (EndsWith(returnValue, "s", exclusionListS)) {
            returnValue = RemoveSuffix(returnValue, "s");
        }

        return returnValue;
    }

    /**
     * Performs unit tests for S_Stemmer class
     */
    public static void test() {
        assert (stemSuffix("joes").equals("joe"));
        assert (stemSuffix("dress").equals("dress"));
        assert (stemSuffix("various").equals("various"));
        assert (stemSuffix("catches").equals("catche"));
        assert (stemSuffix("tomatoes").equals("tomatoe"));
        assert (stemSuffix("trees").equals("tree"));
        assert (stemSuffix("sundaes").equals("sundae"));
        assert (stemSuffix("kidneies").equals("kidneie"));
        assert (stemSuffix("kaies").equals("kaie"));
        assert (stemSuffix("sundries").equals("sundry"));
    }

    /**
     * Calls the unit test method
     */
    public static void main(String[] args) {
        test();
    }
}
```

```
//SuffixStemmer.java
```

```
package assignment3.Stemmers;

/**
 *
 */
public class SuffixStemmer {

    /** Creates a new instance of SuffixStemmer */
    public SuffixStemmer() {
    }

    /**
     * This methood should be overridden by all subclasses
     */
    private static String stemSuffix(String word) {
        return null;
    }

    /**
     * Determines if the word passed, ends with the specified ending, but not the specified exclusions (e.g. ends in es, but not aes)
     *
     * @param String word Word to be stemmed
     * @param String ending Ending of the word to be checked for
     * @param String[] exclusions Array holding strings to check against end of word
     *
     */
    public static boolean EndsWith(String word, String ending, String[] exclusions) {
        boolean returnValue = true;

        if (word.endsWith(ending)) {
            /* If performance is an issue, consider short circuiting this loop*/
            for (int i = 0; i < exclusions.length; i++) {
                if (word.endsWith(exclusions[i])) {
                    returnValue = false;
                }
            }
        } else {
            returnValue = false;
        }

        return returnValue;
```

```java
    }

    /**
     * Determines if the word passed, ends with the specified ending
     *
     * @param String word Word to be stemmed
     * @param String ending Ending of the word to be checked for
     *
     */
    public static boolean EndsWith(String word, String ending) {
        String[] emptyString = new String[] {};

        return EndsWith(word, ending, emptyString);
    }

    /**
     * Removes the string toBeRemoved from the end of string word
     *
     * @param String word Word to be stemmed.
     * @param String toBeRemoved String to be removed from the end of word.
     *
     */
    public static String RemoveSuffix(String word, String toBeRemoved) {
        String returnString = word;

        returnString = returnString.substring(0,
                returnString.length() -
                toBeRemoved.length());

        return returnString;
    }

    /**
     * Removes the string toBeRemoved from the end of string word and then adds the String toBeAdded at the end
     *
     * @param String word Word to be stemmed.
     * @param String toBeRemoved String to be removed from the end of word.
     * @param String toBeAdded String to be added to the end of the word once the String toBeRemoved has been removed.
     *
     */
    public static String RemoveSuffix(String word, String toBeRemoved,
            String toBeAdded) {
        String returnString = word;

        returnString = RemoveSuffix(returnString, toBeRemoved).concat(toBeAdded);

        return returnString;
    }

    /**
     * Performs unit tests for SuffixStemmer class
     */
    public static void test() {
        String[] exclusion1 = new String[] {"oes"};

        assert (RemoveSuffix("joes", "s")=="joe");
        assert (RemoveSuffix("joes", "s", "y") == "joey");
        assert (EndsWith("joes", "s"));
        assert (!EndsWith("joes","s",exclusion1));
        assert (!EndsWith("joes","k"));
    }

    /**
     * calls method to perform unit tests
     */
    public static void main(String[] args) {
        test();
    }
}
```

Appendix H: Code Example 4

One-Use Component

```
public class Harman_Michael_Assn1 {

     public static void main(String[] args)
     {
          Scanner textIn = new Scanner(System.in);
     String wordIn;
     String wordOut;
     Boolean first;
     int ix;

     System.out.println("*---------------------------------*");
     System.out.println("* Stemmer Start                   *");
     System.out.println("*---------------------------------*");

     wordIn = "";
     first = true;

     while (!wordIn.contentEquals("*end") || first)
     {
     first = false;

     System.out.println("Enter plural word or '*end' to end: ");
     wordIn = textIn.next();
```

```

        wordIn = wordIn.toLowerCase();

        if (!wordIn.contentEquals("*end"))
        {
        wordOut = "";

        // If a word ends in “ies” but not “eies” or “aies”
        // Change the “ies” to “y”

        ix = 0;

        if (wordIn.endsWith("ies") &&
             !wordIn.endsWith("eies") &&
             !wordIn.endsWith("aies")) {
             ix = wordIn.lastIndexOf("ies");
             if (ix > 0) {
                  wordOut = wordIn.substring(0, ix) + "y";
                  System.out.println("Rule #1");
             }
        }

        // If a word ends in “es” but not “aes”, “ees”, or “oes”
        // Change the “es” to “e”

        else if (wordIn.endsWith("es") &&
             !wordIn.endsWith("aes") &&
             !wordIn.endsWith("ees") &&
             !wordIn.endsWith("oes") ) {
             ix = wordIn.lastIndexOf("es");
        if (ix > 0) {
            wordOut = wordIn.substring(0, ix) + "e";
            System.out.println("Rule #2");
        }
    }

    // If a word ends in “s”, but not “us” or “ss”
    // Remove the “s”

    else if (wordIn.endsWith("s") &&
        !wordIn.endsWith("us") &&
        !wordIn.endsWith("ss")) {
        ix = wordIn.lastIndexOf("s");
        if (ix > 0) {
            wordOut = wordIn.substring(0, ix) + "";
            System.out.println("Rule #3");
        }
    }

    else {
        wordOut = (wordIn + " (no change)");
        System.out.println("Rule - None");
    }

    System.out.println("Input  word = " + wordIn);
    System.out.println("Output word = " + wordOut);
    System.out.println("*--------------------------------*");
    }
    }
    System.out.println("*--------------------------------
*");
          System.out.println("* Stemmer End *");
          System.out.println("*--------------------------------- *");
          }
     }
```

Reusable Component: 3 files (`Stemmer.java, StemmerDemo.java,` **and** `StemmerRuleManager.java`**)**

```
//Stemmer.java
```

```
/**
  * Stemmer class
  * <p>
  * Describes the Stemmer object.
  *
  * This class provides the main interface to the StemmerRuleManager by
  * initializing the rules arrays with known stem rules and providing an
  * interface to access the method that stemmed the word.
  * <p>
  *
  * @version %I%, %G%
  */

```

```
public class Stemmer {
    private static StemmerRuleManager srm;

     /**
      * Stemmer Constructor
      *
      * This method instatiates the StemmerRuleManager and loads
      * all the rules into the rule collection.
      *
      * The format of the individual rule is a follows:
      * Rule[0] is the ending of the word that we're looking for
      * Rule[1] what we will replace it with if the rule is satisfied
      * Rule[2] through Rule[n] are the exceptions to the rule that must be met
      */
     public Stemmer() {
          srm = new StemmerRuleManager();
          srm.addRules("ies", "y", "eies", "aies");
          srm.addRules("es", "e", "aes", "ees", "oes");
          srm.addRules("s", "", "us", "ss");
     }

     /**
      * Gets the stemmed word based on rules provided in Stemmer Class
      *
      * @param wordIn - the word to be stemmed
      * @param showStemActivity (true/false) - log the
progress of the stemming
     * @return wordOut - the stemmed word according to stem rules
     */
    public String stem(String wordIn, Boolean showStemActivity)   {
        wordIn = wordIn.toLowerCase();
        String wordOut = wordIn;
        wordOut = srm.extractStem(wordIn, showStemActivity);

        return wordOut;
    }
}
```

//StemmerDemo.java

```
import java.util.Scanner;

/**
 * StemmerDemo class
 * <p>
 * Describes the StemmerDemo object.
 * <p>
 *
 * @version %I%, %G%
 */

public class StemmerDemo {
    private static Stemmer stm;

    /**
     * Based on user input, perform Stemmer demo for testing purposes
     *
     * @param args
     */
     public static void main(String[] args) {
          String actionIn1;
          String actionIn2;
          String wordIn;
          Boolean showStemActivity;
          String wordOut;
     Boolean first1;
     Boolean first2;

     actionIn1 = "";
     actionIn2 = "";
     wordIn = "";
     showStemActivity = false;
     wordOut = "";
     first1 = true;
     first2 = true;

          stm = new Stemmer();

          Scanner textIn = new Scanner(System.in);

     System.out.println("*--------------------------------*");
     System.out.println("* Stemmer Start *");
     System.out.println("*---------------------------------*");


     // Ask user if this is a manual test where all input is typed in or
     // if it is an automated test where predefined test data sets are run.
     while (!actionIn1.contentEquals("e") || first1)
     {
     first1 = false;
     System.out.println("Enter 'm' for manual test, 'a' for automated test or 'e' to end : ");
     actionIn1 = textIn.next();
     actionIn1 = actionIn1.toLowerCase();

     // Ask user if logging is to be done for test purposes
     if (actionIn1.contentEquals("m") || actionIn1.contentEquals("a")) {
         System.out.println("Do you want to log the stemming request for debugging purposes (y/n)?");
          actionIn2 = textIn.next();
          actionIn2 = actionIn2.toLowerCase();

          showStemActivity = false;
          if (actionIn2.contentEquals("y")) {
               showStemActivity = true;
          }

     // Automatic stemmer test requested
     // A series of words designed to test Stemmer will be run
     if (actionIn1.contentEquals("a")) {
          wordIn = "babies";
          wordOut = stm.stem(wordIn, showStemActivity);
          System.out.println("*---------------------------------*");
          System.out.println("Input  word = " + wordIn);
          System.out.println("Output word = " + wordOut);
          System.out.println("*---------------------------------*");
          System.out.println();

          wordIn = "pancakes";
          wordOut = stm.stem(wordIn, showStemActivity);
          System.out.println("*---------------------------------*");
          System.out.println("Input  word = " + wordIn);
          System.out.println("Output word = " + wordOut);
          System.out.println("*---------------------------------*");
          System.out.println();

          wordIn = "keys";
          wordOut = stm.stem(wordIn, showStemActivity);
          System.out.println("*---------------------------------*");
          System.out.println("Input  word = " + wordIn);
          System.out.println("Output word = " + wordOut);
          System.out.println("*----------------------------
-----*");
        System.out.println();

    }

    // Manual stemmer test requested
    // user will enter a word and a stemmed word will be returned
    if (actionIn1.contentEquals("m")) {
        first2 = true;
    while (!wordIn.contentEquals("*return") || first2)
    {
        first2 = false;

        System.out.println("Enter plural word or '*return' to return to test options: ");
        wordIn = textIn.next();
        wordIn = wordIn.toLowerCase();

        if (!wordIn.contentEquals("*return"))
        {
        wordOut = "";
            wordOut = stm.stem(wordIn, showStemActivity);

        System.out.println("*---------------------------------*");
        System.out.println("Input  word = " + wordIn);
        System.out.println("Output word = " + wordOut);
        System.out.println("*---------------------------------*");
```

```
          System.out.println();
          }
     }
     }
     }
     }

     // End of Demo
     System.out.println("*---------------------------------*");
     System.out.println("* Stemmer End                     *");
     System.out.println("*---------------------------------*");
     }
}
```

//StemmerRuleManager.java

```
import java.util.*;

/**
 * StemmerRuleManager class
 * <p>
 * Describes the StemmerRuleManager object.
 *
 * This class performs operations on the word to be stemmed using rules
 * that are preloaded when the Stemmer class is instantiated. The output is
 * the stemmed word which is passed back to the Stemmer class and then
 * presneted to the consumer of the method.
 * <p>
 *
 * @version %I%, %G%
 */

public class StemmerRuleManager {

    private int count;
    ArrayList collection = new ArrayList();

    /**
     * StemmerRuleManger Constructor
     */
    public StemmerRuleManager () {
    }

    /**
     * Determines stemmed word using given stem rules
     *
     * @param wordIn - the word to be stemmed
     * @param showStemActivity (true/false) - log the progress of the stemming
     * @return wordOut - the stemmed word according to stem rules
     */
    public String extractStem(String wordIn, Boolean
showStemActivity) {
           String returnWord = wordIn;
           int ix;
           int iy;
           boolean exceptFound;

           if (showStemActivity) System.out.println("*** Begin logging activity for ....... : " + wordIn);

           // Iterate through the collection of rules, extracting one rule
           // at a time and testing the input word against it.

           for (Iterator iter = collection.iterator(); iter.hasNext();)
           {
               String [] rule = (String[]) iter.next();

           // This following algorithm does the following:
      // If a word ends in a particular value indicated in rule[0]
           // and it does not end in any value indicated by rule[2] through rule[n],
      // then remove the ending as described by rule[0] and
           // replace it with the value in rule[1]

      ix = 0;
      exceptFound = false;

      if (showStemActivity) System.out.println("Checking
 Rule ... word must end in ... : " + rule[0]);
      if (wordIn.endsWith(rule[0])) {
           if (showStemActivity)
 System.out.println("Checking Rule ... status is .......... :
 PASSED");
                for (iy=2; iy<rule.length; iy++) {
                     if (showStemActivity)
 System.out.println("Checking Rule ... word must not end in : " +
 rule[iy]);
                      if(wordIn.endsWith(rule[iy])) {
                           if (showStemActivity)
 System.out.println("Checking Rule ... status is .......... :
 FAILED");
                           exceptFound = true;
                           break;
                      } else {
                           if (showStemActivity)
 System.out.println("Checking Rule ... status is .......... :
 PASSED");
                      }
                }
                if (!exceptFound) {
                ix = wordIn.lastIndexOf(rule[0]);
                if (ix > 0) {
                     returnWord = wordIn.substring(0, ix) +
 rule[1];
                     if (showStemActivity) {
                     System.out.println("Applying rule ...
 rule applied is .... : " + rule[0]);
                     System.out.println("Applying rule ...
 ending changed to .. : " + rule[1]);
```

```java
                System.out.println("Applying rule ... new stemmed word is  : " + returnWord);
                }

                break;
            }
        }
        } else {
            if (showStemActivity) System.out.println("Checking Rule ... status is .......... : FAILED");
        }
        }
        if (showStemActivity) {
            System.out.println("*** End   logging activity for ....... : " + wordIn);
            System.out.println();
        }

        return returnWord;
    }

    /**
     * Add the array that contains the rule elements to the
     * rule collection.
     *
     * @param rule
     */
    public void addRules(String ... rule) {
        collection.add(rule);
        }
}
```

Appendix I: Code Example 5

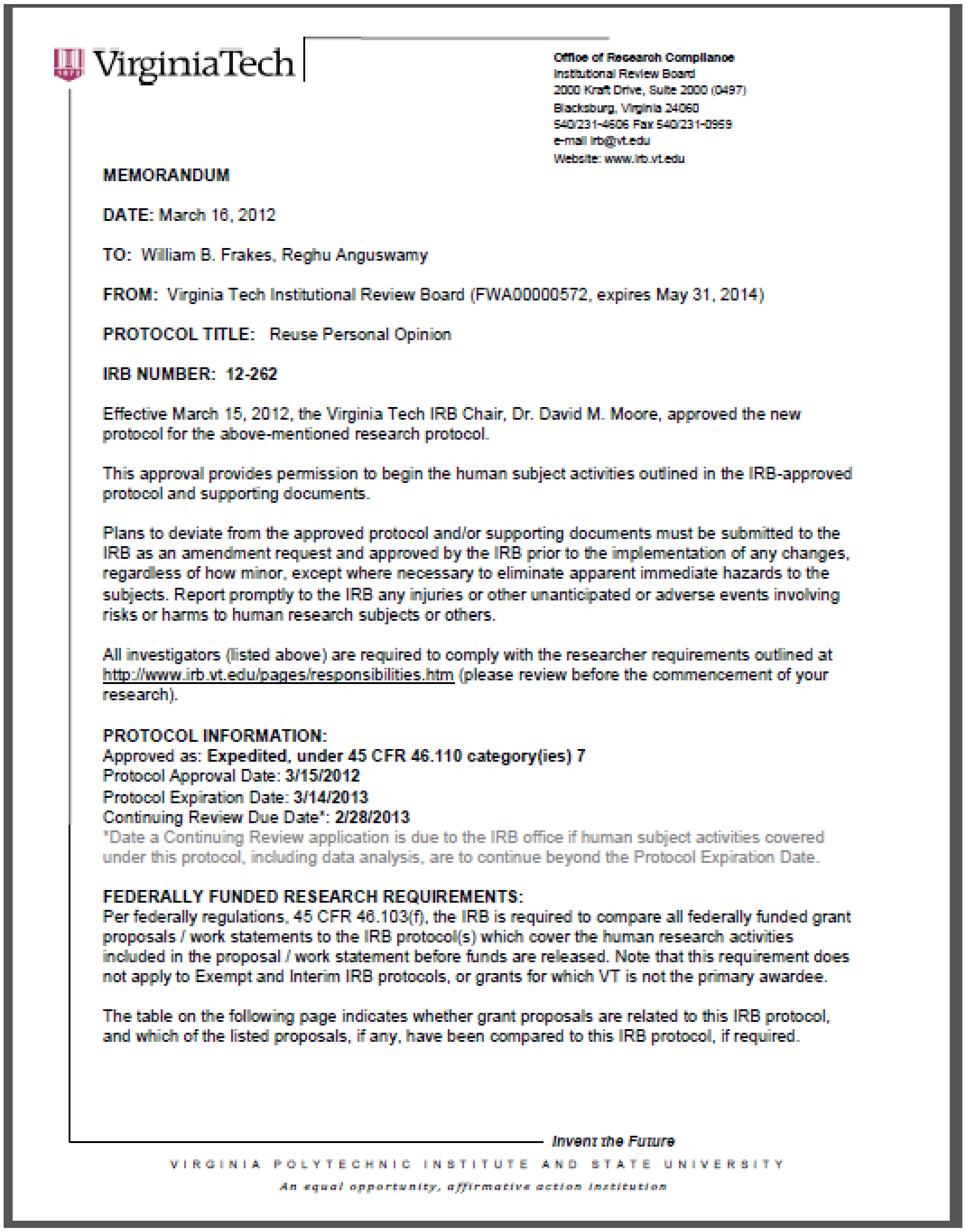
```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class ReuseableSStemmer {
    //Assign variables

    private BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
    private String inputString = null;
    private String tempString = null;
    private String outputString = "not working";

    public class ReadInput {

        public String firstQuestion() {
            System.out.println("Enter a word to stem or exit to leave the program?");
            try {

//read input
                inputString = reader.readLine();

            } //Catch exception
            catch (IOException e) {
                System.out.println("IOException: " + e);

            }
            return inputString;
        }

        public void printStem(String outputString) throws IOException {
//printout result
            System.out.println("The stemmed string is " + outputString + ".");
            //this.fooTwo();
            ReuseableSStemmer o = new ReuseableSStemmer();
            ReuseableSStemmer.ReadInput i = o.new ReadInput();
            i.lowerCase();
            i.ifExit();
            i.ifZero();
            i.stemmer();

        }

        public String lowerCase() {

            this.firstQuestion();
            inputString = inputString.toLowerCase();
            return inputString;
        }

        public String ifZero() {
//assign to input to outputString so that if string is not changed it is displayed
            outputString = inputString;
//if length is zero then prompt for another string
```

```
            if (outputString.length() == 0) {
                this.firstQuestion();
            } else {
            }
            return outputString;
        }

        public void ifExit() throws IOException {
            //test if exit
            if (inputString.equals("exit")) {
                this.leaveProgram();
            }

        }

        public String stemmer() throws IOException {

            //test for ies with exceptions

            if (inputString.endsWith("ies")) {
                if (inputString.endsWith("eies")) {
                } else if (inputString.endsWith("aies")) {
                } else {
//concat y
                    tempString = inputString.substring(0, (outputString.length() - 3));
                    outputString = tempString.concat("y");
                }
                this.printStem(outputString);
            }

```

```java
//test for es with exceptions
      if (inputString.endsWith("es")) {
        if (inputString.endsWith("aes")) {
        } else if (inputString.endsWith("ees")) {
        } else if (inputString.endsWith("oes")) {
        } else {
//concat e
          tempString = outputString.substring(0, (outputString.length() - 2));
          outputString = tempString.concat("e");
        }
        this.printStem(outputString);
      }


      //test for s with exceptions
      if (inputString.endsWith("s")) {
        if (inputString.endsWith("us")) {
        } else if (inputString.endsWith("ss")) {
        } else {
//remove the s
          outputString = inputString.substring(0, (outputString.length() - 1));
        }

        this.printStem(outputString);
      }
      return outputString;
    }

    public void leaveProgram() throws IOException {

      reader.close();
      System.out.println("Goodbye - thanks for stemming!");
      System.exit(0);
    }
}

public static void main(String[] args) throws IOException {
    //prompt the user to enter the string to stem
    System.out.println("Stemming is the process for reducing inflected (or sometimes derived) words to their stem, base");
    System.out.println("or root form â€“ generally a written word form.");
    ReuseableSStemmer o = new ReuseableSStemmer();
    ReuseableSStemmer.ReadInput i = o.new ReadInput();
    i.lowerCase();
    i.ifExit();
    i.ifZero();
    i.stemmer();
}
}
```

Appendix J: IRB Approval Letters

- **IRB Approval#12-262: Reuse Personal Opinion Survey (Chapter 4 and Appendix A)**



VirginiaTech

Office of Research Compliance
Institutional Review Board
2000 Kraft Drive, Suite 2000 (0497)
Blacksburg, Virginia 24060
540/231-4606 Fax 540/231-0959
e-mail irb@vt.edu
Website: www.irb.vt.edu

MEMORANDUM

DATE: March 16, 2012

TO: William B. Frakes, Reghu Anguswamy

FROM: Virginia Tech Institutional Review Board (FWA00000572, expires May 31, 2014)

PROTOCOL TITLE: Reuse Personal Opinion

IRB NUMBER: **12-262**

Effective March 15, 2012, the Virginia Tech IRB Chair, Dr. David M. Moore, approved the new protocol for the above-mentioned research protocol.

This approval provides permission to begin the human subject activities outlined in the IRB-approved protocol and supporting documents.

Plans to deviate from the approved protocol and/or supporting documents must be submitted to the IRB as an amendment request and approved by the IRB prior to the implementation of any changes, regardless of how minor, except where necessary to eliminate apparent immediate hazards to the subjects. Report promptly to the IRB any injuries or other unanticipated or adverse events involving risks or harms to human research subjects or others.

All investigators (listed above) are required to comply with the researcher requirements outlined at http://www.irb.vt.edu/pages/responsibilities.htm (please review before the commencement of your research).

PROTOCOL INFORMATION:
Approved as: **Expedited, under 45 CFR 46.110 category(ies) 7**
Protocol Approval Date: **3/15/2012**
Protocol Expiration Date: **3/14/2013**
Continuing Review Due Date*: **2/28/2013**
*Date a Continuing Review application is due to the IRB office if human subject activities covered under this protocol, including data analysis, are to continue beyond the Protocol Expiration Date.

FEDERALLY FUNDED RESEARCH REQUIREMENTS:
Per federally regulations, 45 CFR 46.103(f), the IRB is required to compare all federally funded grant proposals / work statements to the IRB protocol(s) which cover the human research activities included in the proposal / work statement before funds are released. Note that this requirement does not apply to Exempt and Interim IRB protocols, or grants for which VT is not the primary awardee.

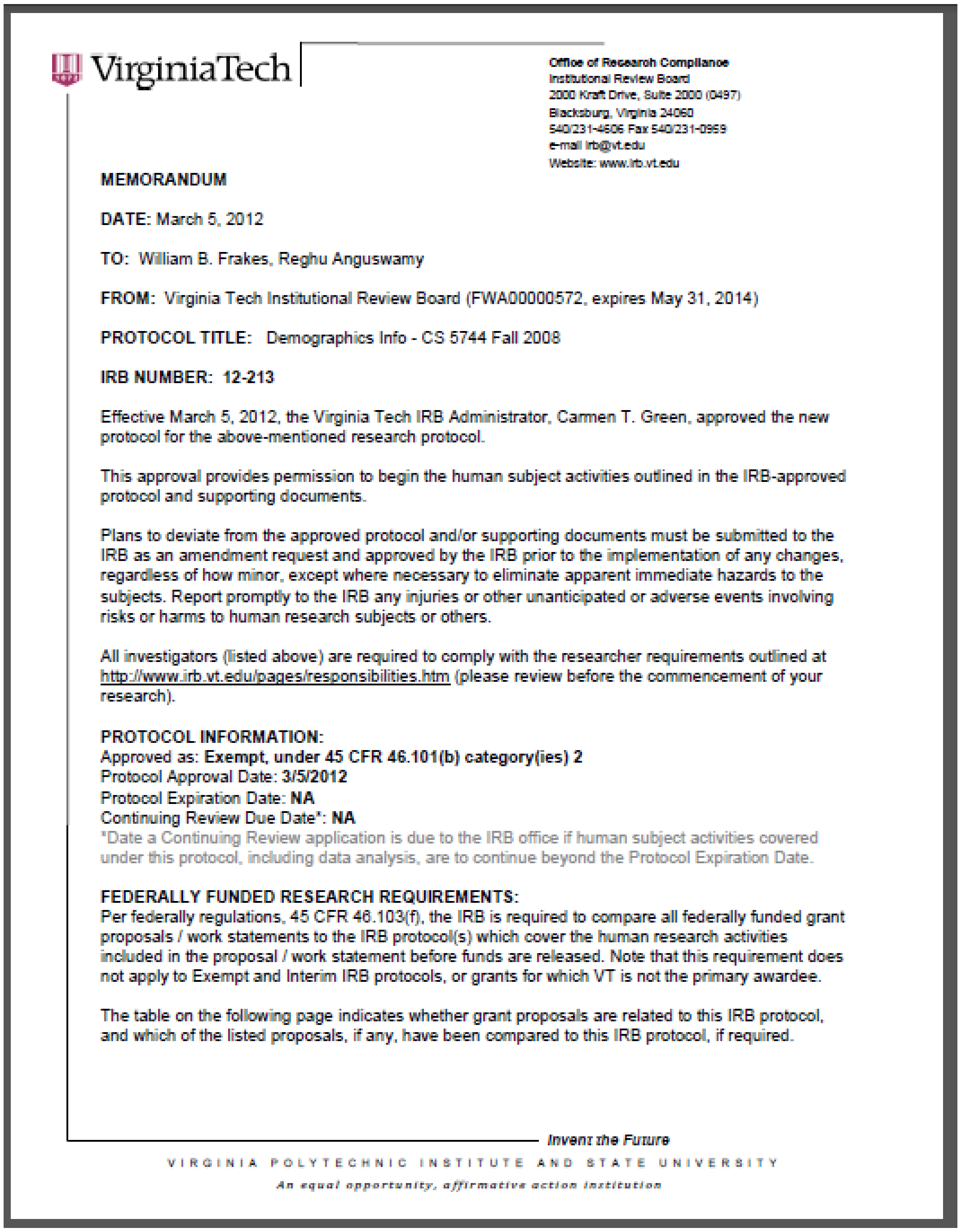
The table on the following page indicates whether grant proposals are related to this IRB protocol, and which of the listed proposals, if any, have been compared to this IRB protocol, if required.

Invent the Future

VIRGINIA POLYTECHNIC INSTITUTE AND STATE UNIVERSITY

An equal opportunity, affirmative action institution

- **IRB Approval#12-213, 12-214, and 12-215: Demographics Survey (Chapter 4 and Appendix B)**



VirginiaTech

Office of Research Compliance
Institutional Review Board
2000 Kraft Drive, Suite 2000 (0497)
Blacksburg, Virginia 24060
540/231-4606 Fax 540/231-0959
e-mail irb@vt.edu
Website: www.irb.vt.edu

MEMORANDUM

DATE: March 5, 2012

TO: William B. Frakes, Reghu Anguswamy

FROM: Virginia Tech Institutional Review Board (FWA00000572, expires May 31, 2014)

PROTOCOL TITLE: Demographics Info - CS 5744 Fall 2008

IRB NUMBER: 12-213

Effective March 5, 2012, the Virginia Tech IRB Administrator, Carmen T. Green, approved the new protocol for the above-mentioned research protocol.

This approval provides permission to begin the human subject activities outlined in the IRB-approved protocol and supporting documents.

Plans to deviate from the approved protocol and/or supporting documents must be submitted to the IRB as an amendment request and approved by the IRB prior to the implementation of any changes, regardless of how minor, except where necessary to eliminate apparent immediate hazards to the subjects. Report promptly to the IRB any injuries or other unanticipated or adverse events involving risks or harms to human research subjects or others.

All investigators (listed above) are required to comply with the researcher requirements outlined at http://www.irb.vt.edu/pages/responsibilities.htm (please review before the commencement of your research).

PROTOCOL INFORMATION:
Approved as: **Exempt, under 45 CFR 46.101(b) category(ies) 2**
Protocol Approval Date: **3/5/2012**
Protocol Expiration Date: **NA**
Continuing Review Due Date*: **NA**
*Date a Continuing Review application is due to the IRB office if human subject activities covered under this protocol, including data analysis, are to continue beyond the Protocol Expiration Date.

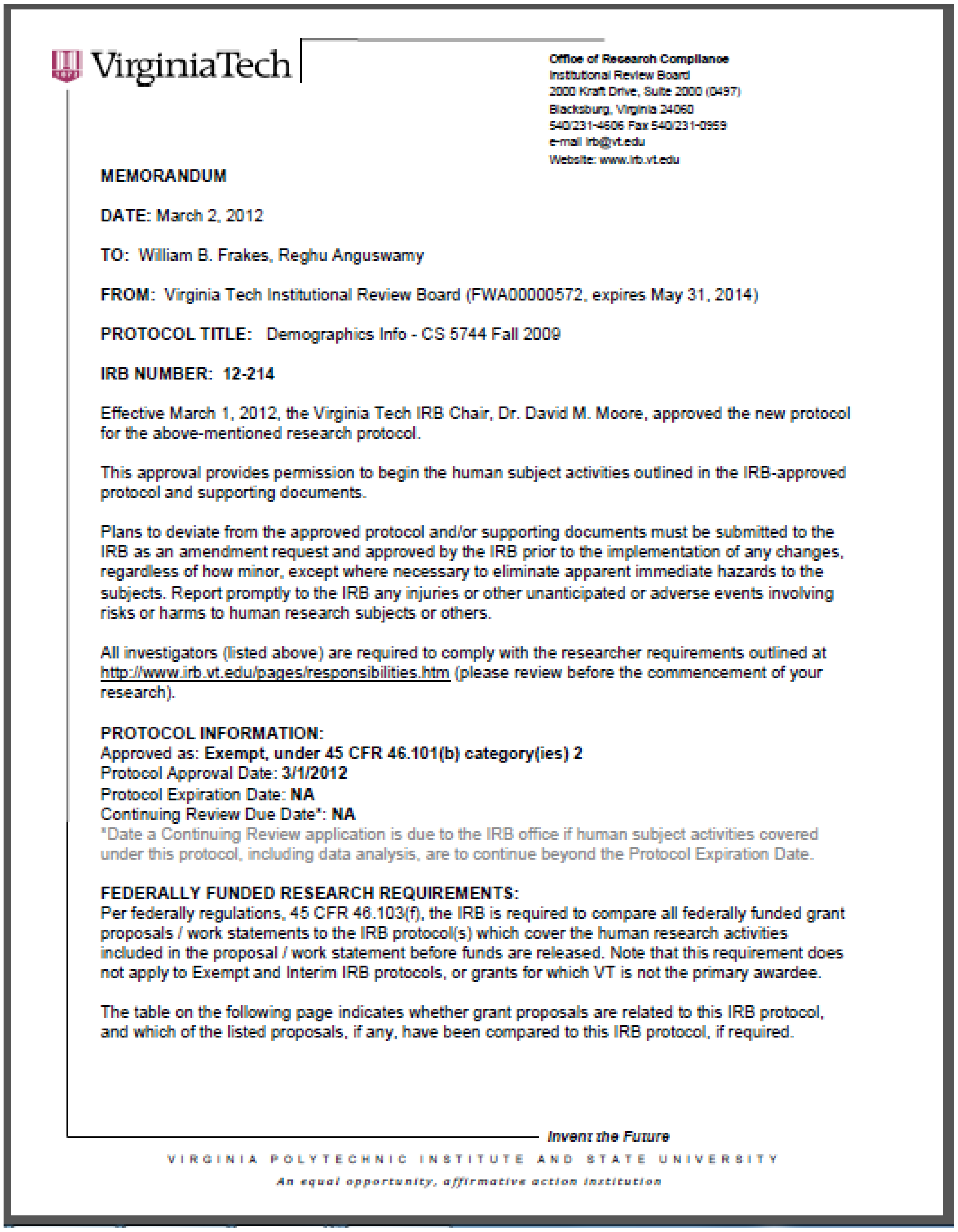
FEDERALLY FUNDED RESEARCH REQUIREMENTS:
Per federally regulations, 45 CFR 46.103(f), the IRB is required to compare all federally funded grant proposals / work statements to the IRB protocol(s) which cover the human research activities included in the proposal / work statement before funds are released. Note that this requirement does not apply to Exempt and Interim IRB protocols, or grants for which VT is not the primary awardee.

The table on the following page indicates whether grant proposals are related to this IRB protocol, and which of the listed proposals, if any, have been compared to this IRB protocol, if required.

Invent the Future

VIRGINIA POLYTECHNIC INSTITUTE AND STATE UNIVERSITY

An equal opportunity, affirmative action institution



VirginiaTech

Office of Research Compliance
Institutional Review Board
2000 Kraft Drive, Suite 2000 (0497)
Blacksburg, Virginia 24060
540/231-4606 Fax 540/231-0959
e-mail irb@vt.edu
Website: www.irb.vt.edu

MEMORANDUM

DATE: March 2, 2012

TO: William B. Frakes, Reghu Anguswamy

FROM: Virginia Tech Institutional Review Board (FWA00000572, expires May 31, 2014)

PROTOCOL TITLE: Demographics Info - CS 5744 Fall 2009

IRB NUMBER: 12-214

Effective March 1, 2012, the Virginia Tech IRB Chair, Dr. David M. Moore, approved the new protocol for the above-mentioned research protocol.

This approval provides permission to begin the human subject activities outlined in the IRB-approved protocol and supporting documents.

Plans to deviate from the approved protocol and/or supporting documents must be submitted to the IRB as an amendment request and approved by the IRB prior to the implementation of any changes, regardless of how minor, except where necessary to eliminate apparent immediate hazards to the subjects. Report promptly to the IRB any injuries or other unanticipated or adverse events involving risks or harms to human research subjects or others.

All investigators (listed above) are required to comply with the researcher requirements outlined at http://www.irb.vt.edu/pages/responsibilities.htm (please review before the commencement of your research).

PROTOCOL INFORMATION:
Approved as: **Exempt, under 45 CFR 46.101(b) category(ies) 2**
Protocol Approval Date: **3/1/2012**
Protocol Expiration Date: **NA**
Continuing Review Due Date*: **NA**
*Date a Continuing Review application is due to the IRB office if human subject activities covered under this protocol, including data analysis, are to continue beyond the Protocol Expiration Date.

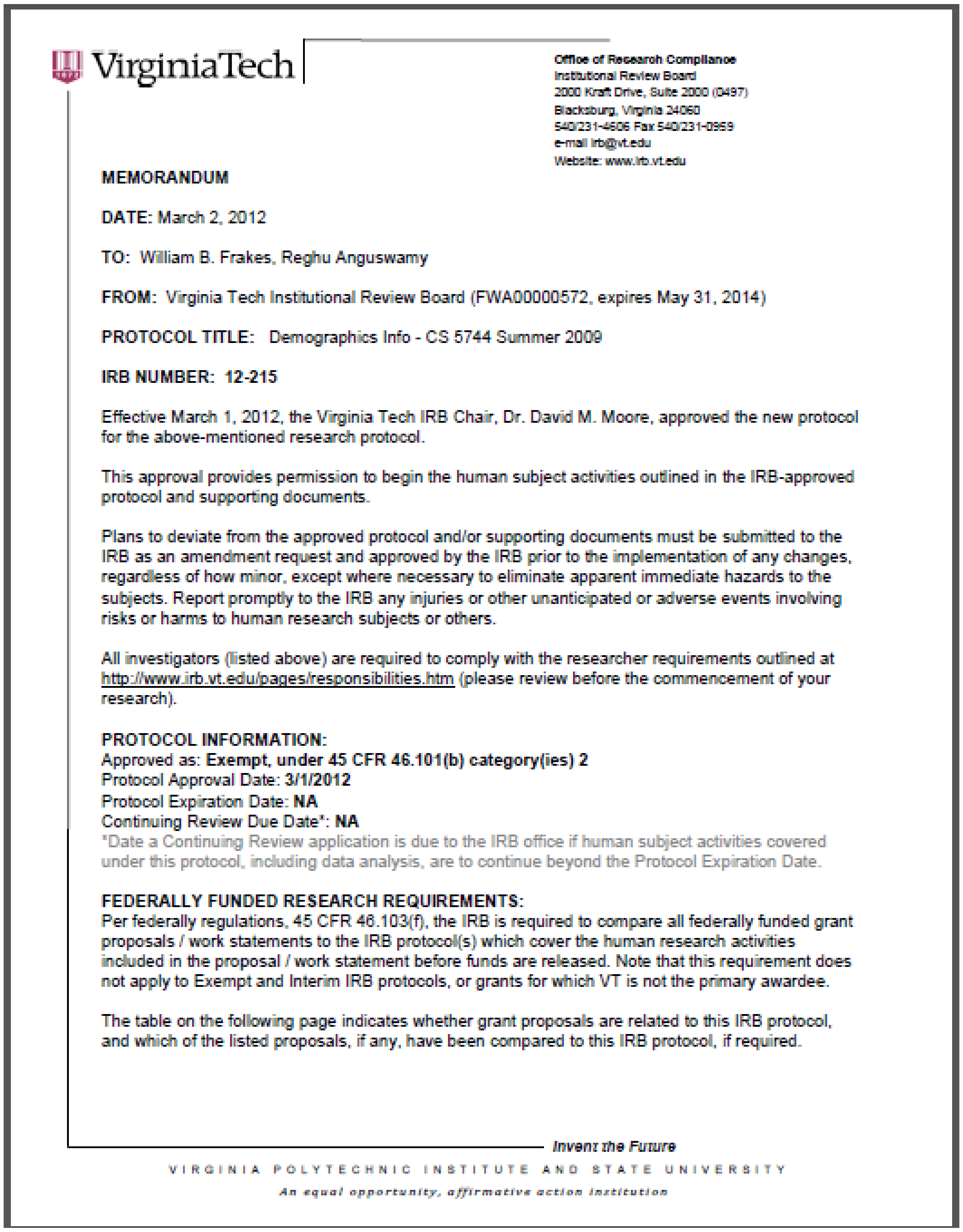
FEDERALLY FUNDED RESEARCH REQUIREMENTS:
Per federally regulations, 45 CFR 46.103(f), the IRB is required to compare all federally funded grant proposals / work statements to the IRB protocol(s) which cover the human research activities included in the proposal / work statement before funds are released. Note that this requirement does not apply to Exempt and Interim IRB protocols, or grants for which VT is not the primary awardee.

The table on the following page indicates whether grant proposals are related to this IRB protocol, and which of the listed proposals, if any, have been compared to this IRB protocol, if required.

Invent the Future

VIRGINIA POLYTECHNIC INSTITUTE AND STATE UNIVERSITY

An equal opportunity, affirmative action institution



VirginiaTech

Office of Research Compliance
Institutional Review Board
2000 Kraft Drive, Suite 2000 (0497)
Blacksburg, Virginia 24060
540/231-4606 Fax 540/231-0959
e-mail irb@vt.edu
Website: www.irb.vt.edu

MEMORANDUM

DATE: March 2, 2012

TO: William B. Frakes, Reghu Anguswamy

FROM: Virginia Tech Institutional Review Board (FWA00000572, expires May 31, 2014)

PROTOCOL TITLE: Demographics Info - CS 5744 Summer 2009

IRB NUMBER: 12-215

Effective March 1, 2012, the Virginia Tech IRB Chair, Dr. David M. Moore, approved the new protocol for the above-mentioned research protocol.

This approval provides permission to begin the human subject activities outlined in the IRB-approved protocol and supporting documents.

Plans to deviate from the approved protocol and/or supporting documents must be submitted to the IRB as an amendment request and approved by the IRB prior to the implementation of any changes, regardless of how minor, except where necessary to eliminate apparent immediate hazards to the subjects. Report promptly to the IRB any injuries or other unanticipated or adverse events involving risks or harms to human research subjects or others.

All investigators (listed above) are required to comply with the researcher requirements outlined at http://www.irb.vt.edu/pages/responsibilities.htm (please review before the commencement of your research).

PROTOCOL INFORMATION:
Approved as: **Exempt, under 45 CFR 46.101(b) category(ies) 2**
Protocol Approval Date: **3/1/2012**
Protocol Expiration Date: **NA**
Continuing Review Due Date*: **NA**
*Date a Continuing Review application is due to the IRB office if human subject activities covered under this protocol, including data analysis, are to continue beyond the Protocol Expiration Date.

FEDERALLY FUNDED RESEARCH REQUIREMENTS:
Per federally regulations, 45 CFR 46.103(f), the IRB is required to compare all federally funded grant proposals / work statements to the IRB protocol(s) which cover the human research activities included in the proposal / work statement before funds are released. Note that this requirement does not apply to Exempt and Interim IRB protocols, or grants for which VT is not the primary awardee.

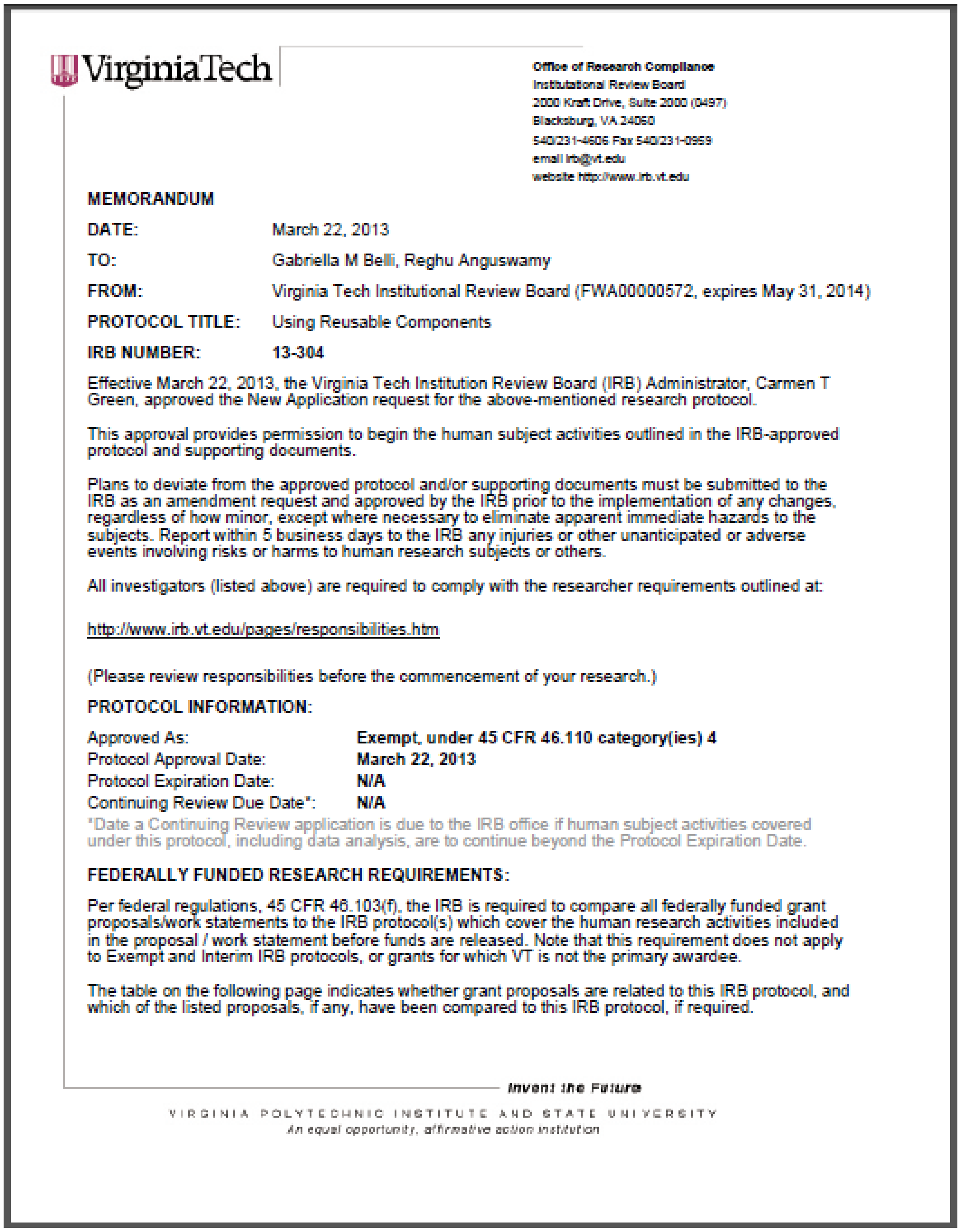
The table on the following page indicates whether grant proposals are related to this IRB protocol, and which of the listed proposals, if any, have been compared to this IRB protocol, if required.

Invent the Future

VIRGINIA POLYTECHNIC INSTITUTE AND STATE UNIVERSITY

An equal opportunity, affirmative action institution

- **IRB Approval#13-304: Demographics and Component Reuse Survey (Chapter 5, and Appendices C and D)**



VirginiaTech

Office of Research Compliance
Institutional Review Board
2000 Kraft Drive, Suite 2000 (0497)
Blacksburg, VA 24060
540/231-4606 Fax 540/231-0959
email irb@vt.edu
website http://www.irb.vt.edu

MEMORANDUM

DATE:	March 22, 2013
TO:	Gabriella M Belli, Reghu Anguswamy
FROM:	Virginia Tech Institutional Review Board (FWA00000572, expires May 31, 2014)
PROTOCOL TITLE:	Using Reusable Components
IRB NUMBER:	**13-304**

Effective March 22, 2013, the Virginia Tech Institution Review Board (IRB) Administrator, Carmen T Green, approved the New Application request for the above-mentioned research protocol.

This approval provides permission to begin the human subject activities outlined in the IRB-approved protocol and supporting documents.

Plans to deviate from the approved protocol and/or supporting documents must be submitted to the IRB as an amendment request and approved by the IRB prior to the implementation of any changes, regardless of how minor, except where necessary to eliminate apparent immediate hazards to the subjects. Report within 5 business days to the IRB any injuries or other unanticipated or adverse events involving risks or harms to human research subjects or others.

All investigators (listed above) are required to comply with the researcher requirements outlined at:

http://www.irb.vt.edu/pages/responsibilities.htm

(Please review responsibilities before the commencement of your research.)

PROTOCOL INFORMATION:

Approved As:	**Exempt, under 45 CFR 46.110 category(ies) 4**
Protocol Approval Date:	**March 22, 2013**
Protocol Expiration Date:	**N/A**
Continuing Review Due Date*:	**N/A**

*Date a Continuing Review application is due to the IRB office if human subject activities covered under this protocol, including data analysis, are to continue beyond the Protocol Expiration Date.

FEDERALLY FUNDED RESEARCH REQUIREMENTS:

Per federal regulations, 45 CFR 46.103(f), the IRB is required to compare all federally funded grant proposals/work statements to the IRB protocol(s) which cover the human research activities included in the proposal / work statement before funds are released. Note that this requirement does not apply to Exempt and Interim IRB protocols, or grants for which VT is not the primary awardee.

The table on the following page indicates whether grant proposals are related to this IRB protocol, and which of the listed proposals, if any, have been compared to this IRB protocol, if required.

Invent the Future

VIRGINIA POLYTECHNIC INSTITUTE AND STATE UNIVERSITY
An equal opportunity, affirmative action institution