A Study of Factors Affecting the Design and Use of Reusable Components

Reghu Anguswamy
(Advisor: Dr. William B. Frakes)
Software Reuse Lab, Virginia Tech.
7054 Haycock Road
Falls Church, VA, USA - 22043
Ph: +1-703.538.8371
Email: reghu@vt.edu

ABSTRACT

Design *for* Reuse: Designing and building components to be reusable is a key area in software reuse research. Practitioners and researchers need to address the problem of how to build reusable components. We will study design principles that can be applied to make components reusable. These design principles are language and domain independent. With an empirical study we will identify the most commonly used reuse design principles. This can be a guideline for designing and building reusable components. Re-engineering a component to be reusable by applying the reuse design principles is a cumbersome task. It is important to understand the overhead involved in making components reusable. We will conduct an empirical study analyzing the overhead in terms of component size, effort required, number of parameters, and productivity.

Design *with* Reuse: Reusing components in a system involves many challenges. Successful reuse of the components depends on how easily a user can use them in the system. It is important to understand the factors that affect the ease of reuse. Through an empirical study, we will analyze the relationships between the size of a component, the reuse design principles used in building the component, and the ease of reuse. We will also analyze the relationship between human factors and the ease of reuse. The human factors considered are the experiences of the user in software programming, software reuse, and programming language. We will also analyze whether testing a component makes it easier to reuse or not.

Categories and Subject Descriptors

D.2.13 [**Reusable Software**]: Component-based software engineering – *reusable components*

Keywords

Software reuse, reusable components, reuse design, reusability, empirical study

1. INTRODUCTION

Software reuse has been widely studied and researched 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 due to software reuse. There are also 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 are components 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. Through an empirical study we will identify these principles. The quality factors we will study are the component size, effort required, number of parameters, and productivity.

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 factors we will study are the reuse design principles used in building the components, and human factors such as the experiences of the user in software programming, software reuse, and programming language. We will also analyze whether testing a component makes it easier to reuse or not.

Based on the faceted classification on types of software reuse by Frakes and Terry [2], our work involves:

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

2. Problem Statement and Motivation

2.1 Design *for* Reuse

Many reuse design principles have been proposed [13-16], but there has been little empirical analysis of their use. Ramachandran [17] 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 which are independent of the language and the domain. So, we have analyzed the literatures of software reuse and reuse design over the past three decades, and provided 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 [14]

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 that it may be more efficient to build it from scratch. Hence, a guideline of design principles in building reusable components is necessary. Our purpose was to conduct an exploratory analysis to identify the most commonly chosen reuse design principles used to develop reusable components. This can be a guideline for building reusable components.

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. We have quantified the differences between one-use and reusable components in terms of their sizes, number of parameters used, and effort required based on the model in [18].

One study that is similar to ours is presented by Seepold and Kunzmann [19] for components written in VHDL . 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.

2.2 Design *with* Reuse

The 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) [20] which consider software reuse and reusing components, the cost is estimated higher for larger reusable components. Vitharana [21] 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. We analyze the effect of the size of components on the ease of reuse. We also analyze the effect of the reuse design principles on the ease of reuse.

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 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 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 a programmer's demographics and the ease of reuse. However, in one study [22], the correlation between programming and UNIX experience, and the effectiveness of searching components was studied. The relationship was found to be not significant. Through an empirical study we analyze the effect of a programmer's demographics such as experiences in programming, software reuse, and programming languages on the ease of reuse.

2.3 Expert Opinion

Based on these observations, 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 eminent 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. 3 of them had more than 16 years of software programming experience and another 3 had 8-16 years of experience. In the field of software reuse, 3 of them had more than 16 years of experience and another 3 had 8-16 years of experience. Five of the seven 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.

2.3.1 Reuse Design Principles

A list of reuse design principles were given to the survey participants (as shown in Figure 1). 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.

2.3.2 Design for Reuse

The members of the reuse group were asked to give their personal opinion 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 don't know; one of them mentioned that though the reusable components are generally larger in size, it is not so always and hence cannot definitely 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.

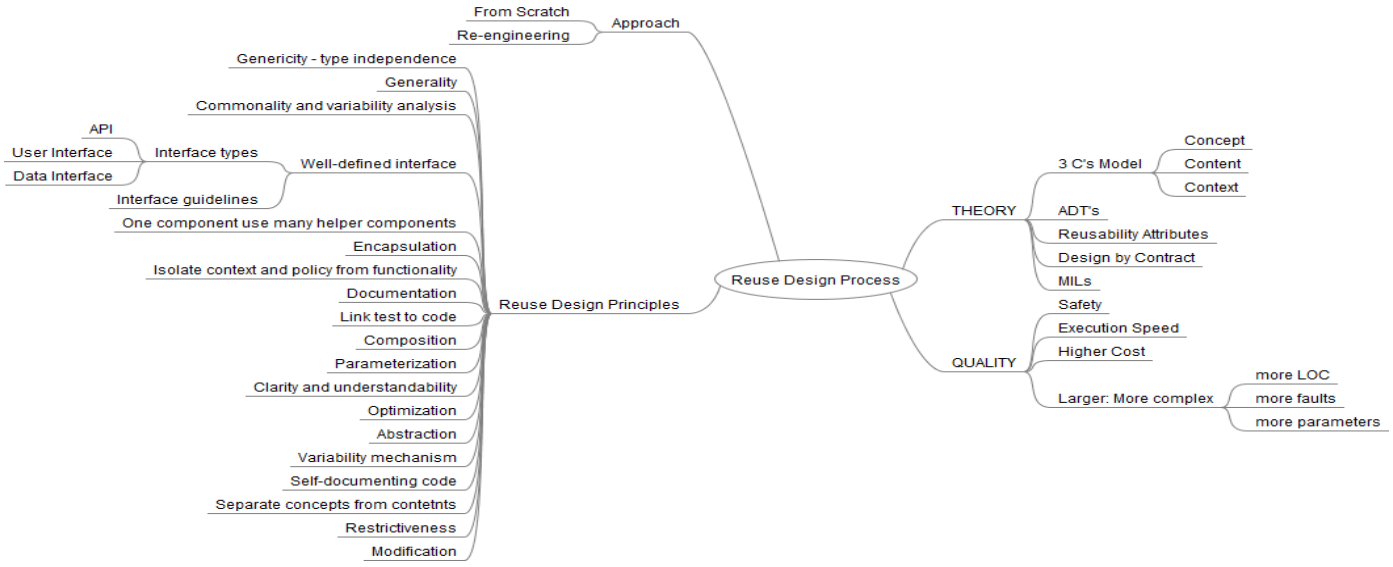

Figure 1. Mindmap of Reuse Design Process

"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 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 the 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.

2.3.3 Design with Reuse

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 to 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 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 relationship between the size of components, the demographics of programmers and the ease of reuse.

3. Reuse Design Principles[1]

"Reuse is a result of good design: it is not something you get from simple-minded use of special language features." – Bjarne Stroustrup [23].

Ramachandran [17] 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 based on the work by Frakes and Lea [13]. The principles are at various levels of abstraction. The 3 C's model of reuse design [24], 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.

4. Hypotheses

4.1 Design for Reuse

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.

$$SLOC_{ReuseDiff/hour} > SLOC_{one\text{-}use/hour} \quad (3)$$

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

$$Parameters_{Reuse} > Parameters_{one\text{-}use} \quad (4)$$

[1] A paper of the literature survey on Reuse Design Principles submitted to *ACM Computing Surveys* for review.

4.2 Design with Reuse

Hypothesis II-a: The smaller the component the easier it is to reuse. 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

Hypothesis II-c: The more the experience a programmer has the easier it is for the programmer to reuse a component

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. Empirical Studies

Two empirical studies are presented: one for the design *for* reuse process and the other for design *with* reuse process.

5.1 Empirical Study I (Design *for* Reuse)[2]

5.1.1 Method

A total of 107 subjects participated in this study. Nearly all the subjects had some experience in software engineering. The subjects were given an assignment to build a one-use software component implementing the s-stemming algorithm [25] 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., USA. 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*

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 was the basis of the lecture. One hundred and one of them converted their one-use components to an equivalent reuse 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).

5.1.1.1 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 reuse components, the subjects were also asked to indicate and justify the reuse design principles (from Figure 1.) that they used. One hundred and seven students successfully built the one-use components and 101 built the equivalent reusable component; six subjects either did not the equivalent reusable component and three of those who submitted did not report back the time required for building the component. All the components, both one-use and reusable components were graded as part of the assignment and required to satisfy 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.

5.1.1.2 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.[26], most cost estimation models were based directly on size measured in SLOC. Some of them are the COCOMO [27], COCOMO II [20], SLIM [28], SEER [29]. 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 [30-33]. There are also empirical studies where productivity of software components is measured in SLOC/hr [30, 31, 33, 34].

Herraiz et al. [35] studied the correlation between SLOC and many complexity measures such as the McCabe's cyclomatic complexity and Halstead's metrics. 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. Linear models have been developed relating SLOC and cyclomatic complexity. Buse et al [36], for example, presented a study where they show a high direct correlation between the SLOC and the structural complexity of the code. Based on these studies, we decided to compare the one-use and reusable components in our study based on the size (in SLOC), effort (man-hours) and productivity (in SLOC/hr).

5.1.2 Initial Results and Analysis

5.1.2.1 Demographics

The subjects answered a questionnaire on their demographics. The questionnaire was optional and 23 subjects completed the questionnaire. Sixteen of the respondents had a highest qualification of an undergraduate degree while 7 of them had completed a master's degree. 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 any experience in software programming. The subjects also gave their background information on software reuse. About two-thirds (65.2%) of the subjects did not have any software reuse program in their organizations. 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.

5.1.2.2 Reuse Design Principles

Table 1 shows the summary of the usage of reuse design principles by the subjects. The mean number of principles used by a subject was 5.4. A *well-defined interface* (#1) was the most highly ranked principle and was used for about half of the reusable components. *Documentation* was ranked 2 and 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 ranked next. This principle allows the users of the component a better and easier way of comprehending the code for future use. The next

[2]Preliminary results presented at SEKE'12: Anguswamy, R. and Frakes, W. B., An Exploratory Study of One-Use and Reusable Software Components, *International Conference of Software Engineering and Knowledge Engineering, SEKE'12,* San Francisco, USA, 1-3 July 2012

three principles were *generality, separate concepts from contents*, and *commonality and variability*.

5.1.2.3 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 [http://www.dwheeler.com/sloccount/]. This tool measures the ``physical SLOC." Physical SLOC is defined as follows: "a physical source line of code (SLOC) is a line ending in a newline or end-of-file marker, and which contains at least one non-whitespace non-comment character." The notched box plots of the SLOC measured for the one-use and reusable components are shown in Figure 2.

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

The effort taken in terms of time (hours) and the number of parameters are shown in Figures 3 and 4 respectively. Figure 5 compares the productivity (in terms of SLOC/hour) of the developers for one-use vs. reuse components. $SLOC_{ReuseDiff}$ 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.

$$SLOC/hr_{ReuseDiff} = (SLOC_{Reuse} - SLOC_{one\text{-}use}) / Effort_{Reuse} \quad (6)$$

Table 1. Ranking of reuse design principles used

Rank#	Reuse Design Principle	Count#
1	well defined interface	56
2	link to 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

Understanding and interpreting box plots can be found in [37]. 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 with a confidence level of 95% [37]. 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 2 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. From Figure 3, 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.

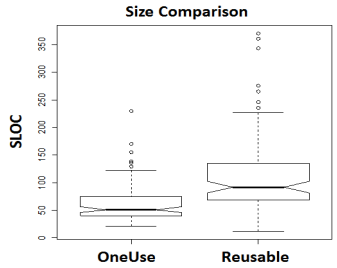

Figure 2. Comparison of actual size (SLOC)

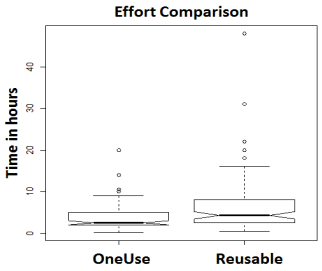

Figure 3. Comparison of Effort (hours)

As can be seen in Figure 4, 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 5, 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 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.

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

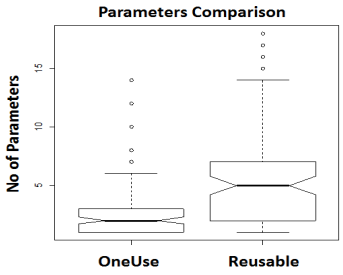

Figure 4. Comparison of #parameters

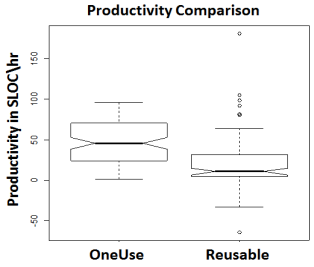

Figure 5. Comparison of productivity (SLOC/hr)

SLOC, effort, productivity and the number of parameters were compared using the 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 2. The table shows that the SLOC, effort, and the number of parameters are statistically siginificantly higher. The productivity also shows a statistically significant difference. The reusable components have significantly lower productivity. Comparing the values of Cohen's d [38] the effect sizes are "*large*" for SLOC, number of parameters, and productivity, and "*medium*" for effort

Table 2. Matched pair t-test statistics (p<0.0001)

Variable	Mean	Std. Dev.	df	t	Cohen's *d*
$(SLOC_{Reuse} - SLOC_{one\text{-}use})$	48.6	53.7	100	9.09	0.89
$(Effort_{Reuse} - Effort_{one\text{-}use})$	3.09	5.9	97	5.1	0.56
$(Parameters_{Reuse} - Parameters_{one\text{-}use})$	2.76	2.87	100	9.64	0.88
$(SLOC_{ReuseDiff/hr} - SLOC_{one\text{-}use/hr})$	-20.5	31.67	97	-6.4	-0.81

5.2 Empirical Study II (Design with Reuse)[3]

5.2.1 Method

Twenty-five components from the sample of 101 components from study I was randomly selected for this study. From the pool of the selected 25 components, each of the 34 subjects in this study was randomly allocated 5 components. 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 I. Each component was reused at least 5 times but no more than 8 times.

5.2.1.1 Reusing the Components

The subjects were given an assignment 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. 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 hosted on surveymonkey (http://www.surveymonkey.com/s/37CVND8). The results are discussed in section 4 (Results and Analysis).

5.2.2 Reuse Design Principles

The distribution of the reuse design principles in the 25 components selected for this study is shown in Table 3. For example, 13 of the 25 components in our study were designed and built using a well-defined interface.

Table 3. Distribution of the reuse design principles

Well defined interface	13
Documentation	15
Clarity and understandability	13

5.2.3 Initial Results and Analysis

5.2.3.1 Demographics

Thirty-four subjects who participated were students of a graduate level course: *Software Design and Quality*. All were enrolled either at Master's or Ph.D. level at Virginia Tech., USA. Nine subjects (27%) already had a master's degree and the rest (73%) had an undergraduate degree. The subjects completed an online questionnaire hosted on surveymonkey (http://www.surveymonkey.com/s/37DLNPM) answering questions on their demographics. The questionnaire was completed by the subjects before they were given the assignment of reusing the components.

Half of the subjects (50.0%) 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. 26.5% of the subjects had at least 2 years of experience in programming but less than 8 years. Since in the practical world, users with no experience in the software industry are also likely to reuse existing components, we have included such subjects in our study as well.

[3]Preliminary results accepted to ESEM'12: Anguswamy, R. and Frakes, W. B., A Study of Reusability, Complexity, and Reuse Design Principles, *6th IEEE/ACM International Symposium on Empirical Software Engineering and Measurement, ESEM'12,* Lund University, Sweden, 19-20 September 2012

5.2.3.2 *Complexity vs. reusability*

After completing the assignment on reusing the components, the subjects completed an online questionnaire hosted on surveymonkey (http://www.surveymonkey.com/s/37CVND8) giving feedback on the applications they built and on the components they had used. 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 the 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).

Each of the 25 selected components in our study was allocated to at least 5 subjects but no more than 8. The average reusability score for a component was calculated as the sum of all the reusability scores for that component divided by the number of reuses. 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. Two components had average reusability scores less than 2. One 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 and was not reused by 2 subjects. This might be an indication that the larger the component the more difficult it is to reuse.

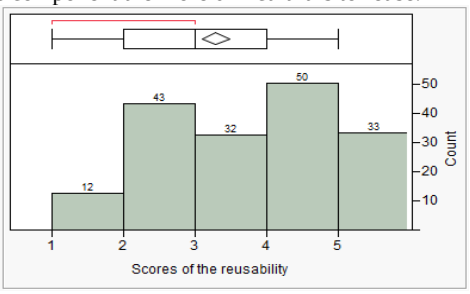

Figure 6. Distribution of the average reusability scores

Regression Analysis: A bivariate plot with a linear fit of the SLOC vs. the average reusability scores of the components is shown in Figure 7. 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. The RSquare value was however very low at 0.102 indicating that only about 10% of variability in the reusability scores is explained by the SLOC. The F-ratio is also not significant: 2.63 with Prob>F at 0.1186.

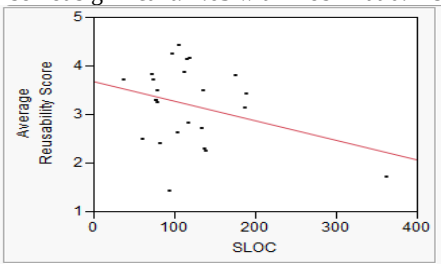

Figure 7. Bivariate fit of SLOC vs, the average reusability scores

5.2.3.3 *Reuse design principles vs. reusability*

The relationship between a reuse design principle and the ease of reuse was studied by comparing the boxplots of two groups of reusability scores. One group consists of the reuse cases where the component reused was built using a given reuse design principle and the other was not. The boxplots were generated using the statistical software `R 2.14.2` (http://cran.r-project.org/).

Of the 170 cases of reuse 88 of them had a well-defined interface, the other 82 were without a well-defined interface. More than half of the reuse cases with a well-defined interface (47) had either a score of 4 or 5. Figure 8 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.

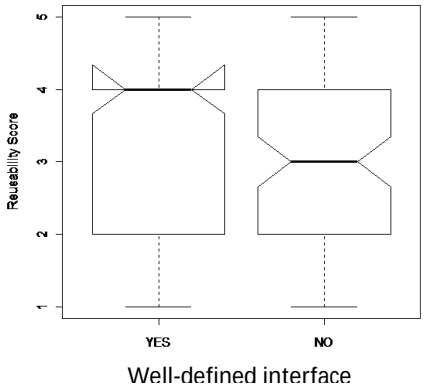

Figure 8. Box-plot comparison of the reusability scores of components with and without a well-defined interface

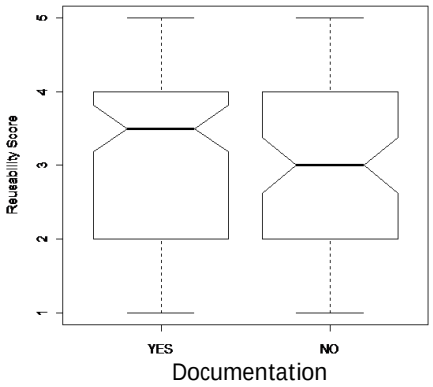

Figure 9. Box-plot comparison of the reusability scores of components with and without documentation

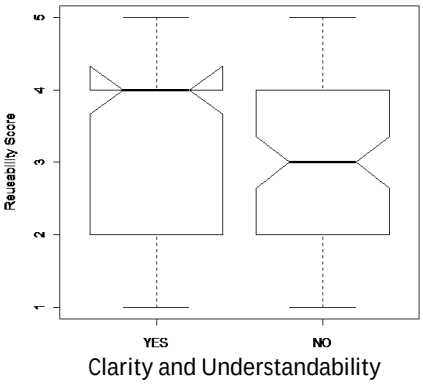

Figure 106. Box-plot comparison of the reusability scores of components with and without clarity and understandability

Of the 25 components used in this study, 15 of them had documentation and of the 170 cases of reuse 103 of them had documentation. Figure 9 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.

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 10 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 for components with clarity and understandability. This indicates that components with clarity and understandability have significantly higher reusability scores.

6. Threats to Validity

Here we identify the internal and the external threats to the two empirical studies.

6.1 Empirical Study I (Design *for* Reuse)

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. Also, the developers had to report why they chose the reuse design principles they used. This alleviated the threat to the validity of the reuse design principles used. 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 our study are small and only in Java, similar studies may be needed larger reusable components and in other languages as well.

6.2 Empirical Study II (Design *with* Reuse)

The second study also needs further studies with larger components and other languages as well. The subjects had varied degrees of experience from none to very high (>8 years). This may reflect the real world but is still a threat to external validity. 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. However, it was emphasized that the subjects would not be penalized for not being able to reuse any component, but they had to report the reasons why they could not reuse it. The subjects could have also discussed and provided the same response for the components. To alleviate this threat, no two subjects were given the same set of components. The reuse design principles that the subjects attributed to the components claimed to exhibit was based on the evaluation by the course grader. Reusability of a code component is always measured from the user's perspective. So, the reusability of a component is measured as the ease of reuse as perceived by the subjects reusing the component.

7. Issues and Questions

I would like the following feedback at the doctoral symposium:

- How can this work be followed up with studies for larger components and other languages?
- Are there any design principles that could be included in the proposed list?
- Are there any more ways to improve the data collection?
- Could more data analysis be done on the existing data?
- Are there any other questions relevant to reusable components that could be addressed in future studies?

8. 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 Comput. Surv.,* 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*, et al.*, "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 Softw.,* vol. 11, pp. 23-30, 1994.

[8] A. Paul*, et al.*, "Adaptive Reuse of Libre Software Systems for Supporting On-line Collaboration," ed, 2008.

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

[10] H. Nakano*, et al.*, "An Empirical Study on Software Reuse," in *Computer Science and Software Engineering, 2008 International Conference on*, 2008, pp. 509-512.

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

[12] D. Lucrédio*, et al.*, "Software reuse: The Brazilian industry scenario," *Journal of Systems and Software,* vol. 81, pp. 996-1013, 2008.

[13] 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.

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

[15] B. Weide*, et al.*, "Reusable software components," *Advances in computers,* vol. 33, pp. 1-65, 1991.

[16] M. Ezran*, et al.*, *Practical software reuse: the essential guide*. London: Springer Verlag, 2002.

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

[18] W. B. Frakes and M. Tortorella, "Foundational Issues in Software Reuse and Reliability," 2008.

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

[20] B. Boehm*, et al.*, "Cost estimation with COCOMO II," ed: Upper Saddle River, NJ: Prentice-Hall, 2000.

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

[22] 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.

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

[24] L. Latour*, et al.*, "Descriptive and predictive aspects of the 3Cs model: SETA1 working group summary," 1991, pp. 9-17.

[25] W. B. Frakes and R. Baeza-Yates, *Information retrieval: data structures and algorithms*, 2nd ed. vol. 77. Englewood Cliffs, NJ: Prentice-Hall. , 1998.

[26] B. Boehm*, et al.*, "Software development cost estimation approaches — A survey," *Annals of Software Engineering,* vol. 10, pp. 177-205, 2000.

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

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

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

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

[31] T. Tan*, et al.*, "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.

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] R. McGill*, et al.*, "Variations of box plots," *American Statistician,* pp. 12-16, 1978.

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