A Study of reusability, complexity, and reuse design principles

Conference Paper · September 2012

DOI: 10.1145/2372251.2372280

CITATIONS
24

READS
2,143

2 authors:

Reghu Anguswamy
Virginia Tech
13 PUBLICATIONS **51** CITATIONS
SEE PROFILE

William B Frakes
IEEE TCSE committee on software reuse
169 PUBLICATIONS **5,705** CITATIONS
SEE PROFILE

A Study of Reusability, Complexity, and Reuse Design Principles

Reghu Anguswamy
Software Reuse Lab, Virginia Tech.
7054 Haycock Road
Falls Church, VA, USA - 22043
Ph: +1-703.538.8371
email: reghu@vt.edu

William B. Frakes
Software Reuse Lab, Virginia Tech.
7054 Haycock Road
Falls Church, VA, USA - 22043
Ph: +1-703.538.8371
email: wfrakes@vt.edu

ABSTRACT
A study is reported on the relationship of complexity and reuse design principles with the reusability of code 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 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 is 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. The reuse design principles: *well-defined interface,* and *clarity and understandability* significantly increase the ease of reuse, while *documentation* does not have a significant impact on the ease of reuse

Categories and Subject Descriptors
D.2.13 [**Reusable Software**]: Component-based software engineering – *reusing components*

Keywords
Software reuse, reuse design, reusability, empirical study

1. INTRODUCTION
Studies from the software industry have analyzed benefits from software reuse such as improved software quality, productivity, and reliability [1-6]. Component based development for software reuse was presented as early as 1969 by McIlroy [7] suggesting that interchangeable pieces called software components should form the basis for software systems. Our study is based on analyzing the reusability of code components in an application. Reusability 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 [8]. There have been no or little empirical studies similar to ours.

In a previous study [9], subjects built one-use stemming components [10]. 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 reusable components were found to be significantly larger (the median was 80% higher) in size than their equivalent on-use components. The three most commonly used reuse design principles identified in the study were *well-defined interface, documentation,* and *clarity and understandability.* We will now define briefly each of these principles.

Well-defined interface: 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 [11].
Documentation: It is essential for any future use or modification and critical for maintainability. Documentation should be self-contained, adaptable and extensible [11]. Stand-alone external documentation, interface documentation lilke Javadocs, or in-line code documentation are all considered *documentation*.
Clarity and Understandability: This is the degree to which a component is easily understood regarding functionality. Matsumoto [12] discusses definiteness as a 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.
The common belief is that the larger the component the harder to reuse. Even in popular cost estimation models such as COCOMO II [13] which consider software reuse and reusing components, the cost is estimated higher for larger reusable components. Our study analyzes the relationship between the size of components and the ease of reuse. We also analyze the relationship between the reuse design principles and the ease of reuse.

2. HYPOTHESES
Hypothesis 1: The smaller the component the easier it is to reuse. Size is measured in SLOC (source lines of code).

Hypothesis 2: A component designed and built with a given reuse design principle will be easier to reuse than a component which is not built based on that reuse design principle. For this hypothesis, we analyzed individually the three most used reuse design principles [9] - *well-defined interface, documentation,* and *clarity and understandability.*

3. METHOD
Based on the faceted classification of types of software reuse by Frakes and Terry [14], the reuse design in our 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.

3.1 Subject 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.

The subjects were asked to mention their roles in their respective organizations. They could choose multiple roles and 8 had at least 2 roles. Almost two-thirds (64.7%) were involved in development and programming. About one-fifth (20.6%) of the subjects were system architects. Five subjects (14.7%) were both system architects and software developers/programmers. 17.6% of the subjects were systems engineers. Two of them were system architects as well. 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.

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.

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. Almost one-third (35.3%) had no experience in the field of software reuse and less than one-tenth (8.8%) had considerable experience (>8 years). One-fifth of the subjects had very little experience (0-2years) in software reuse.

3.2 Allocating and Reusing Components

In a previous study [9], 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 [10]. 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. All the components compiled and executed successfully, and passed a suite of test cases. The s-stemming algorithm implemented was specified by 3 rules as given below (only the first applicable rule was 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

Twenty-five components from the sample of 101 components from study [9] 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 [9]. Each component was reused at least 5 times but no more than 8 times.

In this study, 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. 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 to choose 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 hosted on surveymonkey (http://www.surveymonkey.com/s/37CVND8). The results are discussed in section 4 (Results and Analysis).

3.3 Reuse design principles

In study [9], the subjects were given training on designing and building reusable components. Nineteen reuse design principles were taught to the subjects via class lectures. The subjects had reported the reuse design principles they used and were validated by the course grader. That study identified three most frequently used reuse design principles as – *well-defined interface*, *documentation*, and *clarity and understandability*. The distribution of the reuse design principles in the 25 components selected for this study is shown in Table 1. For example, 13 of the 25 components in our study were designed and built using a well-defined interface.

Table 1. Distribution of the reuse design principles

Well defined interface	13
Documentation	15
Clarity and understandability	13

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/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). The distribution of the reusability scores is given in figure 1. 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).

4.1 Complexity vs. reusability

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.[15], many cost estimation models were based directly on size measured in SLOC. COCOMO [16], COCOMO II [13], SLIM [17], SEER [18] are some of them.

Complexity of software components has been measured based on SLOC in many empirical studies [19-22]. In this study, the smallest component had 37 SLOC and the largest component had 361 SLOC. Half of the components had SLOC between 77 (25^{th} percentile) and 136 (75^{th} percentile). Twenty of the components were between 50 and 150 SLOC.

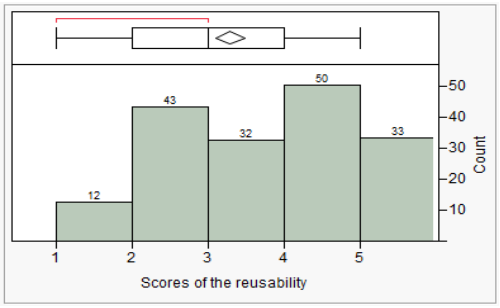

Figure 1. Distribution of the reusability scores

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. 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 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 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 other 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.

Regression Analysis: A bivariate plot with a linear fit of the SLOC vs. the average reusability scores of the components is shown in figure 2. 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.

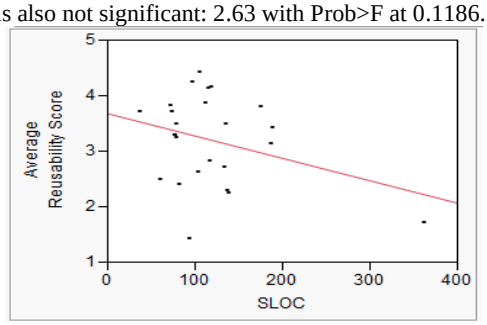

Figure 2. Bivariate fit of SLOC vs, the average reusability scores

4.2 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. 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 with a confidence level of 95% [23]. 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 indicating that they were easy to reuse. Figure 3 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.

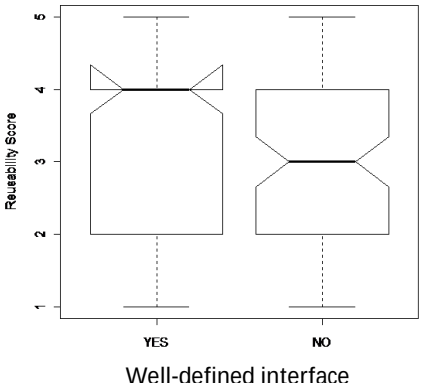

Figure 3. Box-plot comparison of the reusability scores of components with and without a well-defined interface

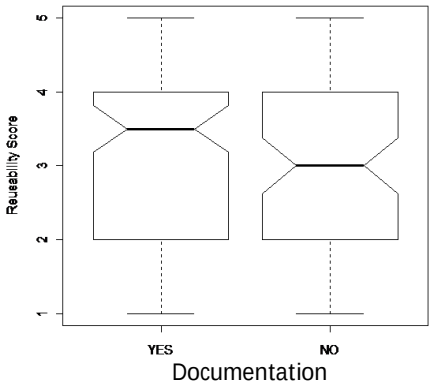

Figure 4. Box-plot comparison of the reusability scores of components with and without documentation

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 4 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 5 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.

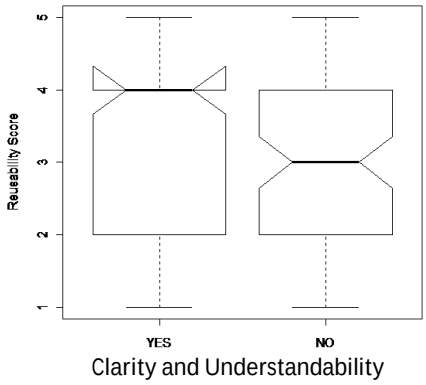

Figure 5. Box-plot comparison of the reusability scores of components with and without clarity and understandability

4.3 Threats to validity

The components used are only in Java. So, the results may not be valid for components in other languages. The components are also small in size. Future studies may involve larger components and in other languages as well. The subjects had varied degrees of experience from none to very high (>8 years). This may a reflection of 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. The reuse design principles that the subjects attributed to the components claimed to exhibit was based on the evaluation by the course grader.

5. CONCLUSIONS AND FUTURE WORK

In this study the relationship of reusability with complexity and reuse design principles was reported. 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 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 relationship was not statistically significant. An analysis of the relationship between a set of reuse design principles and the ease of reuse was also reported. Two of the three reuse design principles: *well-defined interface,* and *clarity and understandability* significantly increased the ease of reuse while *documentation* does not have a significant impact on the ease of reuse.

The impact of the development environment, OS and programming language on the ease of reusing code components may also be studied in future. Realizing that the components in our study are smaller in size and only in Java, similar studies may also be done for larger reusable components and in other languages as well. The subjects had also answered open ended questions giving feedback on cases when they could not reuse the components. A content analysis may be done on the feedback. Complexity and reuse design principles are considered as independent factors in this study. These combined with more commonly used reuse design principles and subject demographics, such as experience in programming and software reuse, may be considered for a combined analysis to identify the combination of factors that affect reusability.

6. References

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

[2] P. Mohagheghi and R. Conradi, "Quality, productivity and economic benefits of software reuse: a review of industrial studies," *Empirical Software Engineering,* vol. 12, pp. 471-516, 2007.

[3] 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.

[4] 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.

[5] M. Morisio*, et al.*, "Success and Failure Factors in Software Reuse," *IEEE Transactions on Software Engineering,* vol. 28, pp. 340-357, 2002.

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

[7] M. D. McIlroy*, et al.*, "Mass produced software components," *Software Engineering Concepts and Techniques,* pp. 88–98, 1969.

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

[9] R. Anguswamy and W. B. Frakes, "An Exploratory Study of One-Use and Reusable Software Components," *24th International Conference of Software Engineering and Knowledge Engineering, SEKE'12*, San Francisco, USA, 2012. (ACCEPTED)

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

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

[12] 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.

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

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

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

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

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

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

[19] 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.

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

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

[22] 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.

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