Who Reuses What?
C++ Libraries and Programmer's Traits

Mark Lattanzi
Dr. Sallie Henry

Virginia Tech
Computer Science Department
Blacksburg, Virginia 24060
lattanzi@cs.vt.edu

Abstract. Software reuse is one key to increasing productivity within the software development process [1, 2, 3]. By reusing existing software, time and effort are saved in the coding, testing, and maintenance phases of a software product. The goals of this research involve examining the process of software reuse and attempting to characterize it. A set of experiments have been run to determine the effects of software reuse in the object oriented paradigm. This paper focuses on the traits of reusers, what type of reuse they perform, and the common characteristics of the classes being reused. In particular, the traits of programmers who reuse often are identified as well as the traits of those who reuse infrequently. Also, do programmers perform black box reuse (reusing library classes without modification) or do they perform white box reuse (using inheritance to modify a library class to the desired functionality) or do they reuse at all? Lastly, a first attempt at characterizing the reusability of a C++ class is presented. Several traits are identified as making a class more reusable to programmers.

1 Introduction

This paper focuses on the concept of reusing object oriented software (C++) classes. Previous research has been done at Virginia Tech which compares the procedural paradigm to the object oriented paradigm [4] and shows the benefits of reusing software in the object oriented paradigm [5]. A new series of experiments has been run to further explain and characterize the effects of software reuse in the object oriented paradigm. This paper focuses on three aspects of software reusability: the type of reuse that is occurring when reusing classes from C++ libraries, the traits of the programmers who are reusing classes, and the characteristics of a reusable class (a class that gets reused often or easily).

Regarding the type of reuse occurring, three types of reuse are identified: *black box reuse* - reusing a library class without modification, *white box reuse* - using inheritance to modify a library class to obtain the desired functionality, and *no reuse* - designing a new class from scratch. The merits of these three types of reuse are not yet fully understood. Clearly, black box reuse is the easiest from the programmer's point of view, but seems to require that the library classes are very specific. Thus, class libraries tend to grow large and classes become difficult to find. White box reuse allows for taking general classes and tailoring them to the problem at hand; however, this requires that the class implementation be, at least partially, understood. Programmers may be reluctant to put forth the effort required to use inheritance.

The traits of programmers who reuse often are also examined. An experiment using 30 Computer Science seniors was conducted and data was collected about all of the subjects including their academic traits and their programming aptitude and experience. Each subject implemented portions of object oriented designs and was free to use code from a library of C++ classes. Correlations have been found between the traits of the programmers and the amount of reuse they performed.

Lastly, this research focuses on the traits of a reusable classes. Others [6, 7, 8] have attempted to characterize a reusable class. This research takes a subjective approach to identifying the traits of a reusable class. The subjects of the experiments as well as some industry experts were interviewed about various classes in some C++ libraries. Their opinions on what characterizes a reusable class have been compiled into several broad factors.

2 Experiment Description

The research experiment was performed in a senior level Computer Science course at Virginia Tech [9]. The students were required to design and implement an object oriented system. Data for this research comes from the term projects developed during the course. The experiment described below occurs over the course of an fifteen week class meeting five times a week for approximately one hour each meeting.

Thirty students are enrolled in the course. They are divided into ten groups of three students each. The first part of the experiment is to teach to students object oriented analysis and design. After that, they are presented with some C++ libraries and educated about the available reusable components (classes) in them. They are encouraged to find and reuse code from other sources as well. Most of the classes that the students needed to develop were either interface type classes or data structure classes. Therefore, the libraries provided to the students concentrated on these two areas. There were very few instances where the class being developed was unrelated to a class in any of the libraries. In other words, the potential for reuse was present for most of the classes developed by the programmers.

Each three-person team is required to design a unique medium size software system (3000-5000 lines of code). All the systems have to use some complex data structures and also have an extensive graphical user interface. All the teams are practicing object oriented design and coding in C++. Several libraries of reusable objects and routines are at the teams disposal, but the teams are not forced to use any objects that they do not wish to use. Each team must design their own project and then "hire" other students to do the actual coding for them. Doing this allowed each of the subjects many opportunities to reuse classes in the software libraries. Every student in the course is required to code for at least three other projects. Each project team has to hire five or more other students as programmers. After all the coding is completed, the design teams then integrate the various components into a working system. At this point, the design teams are allowed to write any connective code that is necessary. All of the code that each subject wrote is saved and analyzed to determine if and how that particular programmer is reusing.

3 Results

Figure 1 shows the tabulated results from the experiment. The rows of the table represent *amount of reuse* performed. The first two columns further define the amount of reuse. Row one is the averaged traits of the six programmers (column two) who reused between zero and two classes (column one). The level of reuse is divided into four broad categories from *little/no reuse* (0-2 classes reused) to *very high reuse* (10+ classes reused). Columns three through five are the dependent variables for this study. These columns are the new classes developed, the library classes used (black box reuse), and the library classes used as parents for new classes (white box reuse) respectively. Column six is the average total reused classes for each group (sum of columns four and five). Column seven is the average total classes written or reused by the group (the sum of columns three and six). Column eight represents the percent reuse for each group (column six / column seven * 100).

	1	2	3	4	5	6	7	8		9	10	11	12	13	14	15	16	17
Level of Reuse	**Number of Classes**	**Number of Programmers**	**Original Classes**	**Black Box Classes Reused**	**White Box Classes Reused**	**Total Classes Reused**	**Total Classes**	**Reused / Total Classes (%)**		**UNIX**	**Inheritance**	**Encapsulation**	**Design Docs**	**Bottom Up Design**	**Procedural Reuse**	**Modifying Other'sCode**	**Reusing Other's Code**	**Software Libraries**
No / Little Reuse	0-2	6	15	1	0	1	16	8		543	2	97	47	73	42	121	127	42
Moderate Reuse	3-6	12	11	4	1	4	15	28		661	5	227	154	194	322	161	140	67
High Reuse	7-9	8	9	5	3	8	17	47		797	14	271	212	226	581	240	239	79
Very High Reuse	10+	3	7	9	3	12	19	63		871	30	560	615	740	939	1067	1067	275

Figure 1: Averaged Programmer Reuse Divided into Four Levels

The independent variables for the study occupy the rest of the table. For each of these columns, the number represents a weighted aptitude for the given concept or skill. Each programmer self-rated themselves on a set of skills (how long they've had the skill, how often they use it, and how proficient they currently are). The nine skills listed showed correlations to the reuse performed by each programmer.

4 Analysis

From the preliminary analysis of the data shown in figure one, three significant results have been found.

Programmers show a prevalence toward black box reuse.

All of the subjects' code is analyzed by a new tool developed at Virginia Tech: the C++ Metrics Analyzer. A T-test on the mean number of classes developed by each programmer shows that the mean number of classes for black-box reuse is significantly greater than the white box reuse mean ($p < .001$). The T-test was performed using the unaveraged programmer data (n=30). Columns four and five of figure one show the averages for the four reuse levels which also illustrates this tendency. This result is expected because black box reuse is easier to do than white-box reuse. Black box reuse requires no understanding of the internals of the class; white box reuse does. Furthermore, if the desired class does not exist in a class library, it seems that programmers are more inclined to write a new class than to figure out which existing library class would make a good parent for the desired class. In order to use inheritance, the parent class's implementation must be understood. Programmers seem reluctant to make this investment. So, more black box reuse and original class development are encountered over white box reuse.

Reusers tend to have the following traits.

The thirty programmers in the experiment are sub-divided into four groups by the number of classes they reused while writing code. No distinction is made between black-box or white-box reuse. The four groups are: no/little reuse, some reuse, moderate reuse, and high reuse. The average characteristics of the programmers in each of these groups are then compared. The most obvious correlation is to reuse experience in the procedural paradigm. More experience reusing in the procedural paradigm seems to cause the programmers to reuse more in the object oriented paradigm. Other programmer traits along this line include experience with software libraries, experience writing high level design documents, and experience with the concepts of encapsulation and inheritance. Programmers who frequently modified other people's code also tend to reuse more than programmers who work mostly on their own code. These trends can be seen in figure 1. The last nine columns are the

concepts / skills that correlated with the amount of reuse that a programmer performed.

What makes a class reusable? (subjective)

This result comes from the subjects' opinions about the class libraries as well as from experts in the field of C++ systems development. Other research has been done in this area as well. Korson and McGregor [7] identify a set of 23 attributes of reusable software ranging from inheritance structure to completeness. Some of the attributes can be measured objectively and some are subjective and need to be further divided into their measurable aspects. A few of these attributes can only be measured subjectively (by experts). Korson's research is based in the object oriented paradigm as well. Johnson and Foote [6] published a similar paper entitled *Designing Reusable Classes,* which enumerates 13 rules to follow to develop classes with a high reuse potential. Johnson's and Foote's rules are based on experience and subjective data. This research intends to start with Korson's attributes and Johnson's rules as a foundation for developing a suite of objective and subjective reusability metrics.

After this experiment is completed, the subjects are interviewed about their experience working in the object oriented paradigm and with C++ class libraries. Several traits of a reusable C++ class are identified. The most common response deals with the external documentation of the class as well as the readability of the code of the class. Both industry experts and a third of the experiment subjects mention these characteristics. Other frequently mentioned traits include how easy the class library is to use, the simplicity (ease of understanding) of the implementation of the code, the reliability of the class, and the completeness of the class implementation (functionality).

Most of these traits are slanted toward the class's usability. Clearly, the easier a class implementation is to understand, the more likely that it will be reused. A further area to research is to measure the complexity of the reused classes and look for a correlation between it and the frequency of the class's use by the subjects of the experiments.

5 Conclusions and Future Work

The above results are the beginning of the analysis of a series of experiments performed at Virginia Tech. This data will be used to further characterize the effects of software reuse and the object oriented paradigm. Other questions that this research will address include:

What is the complexity of frequently reused classes?
What are the traits of a "good" (highly reusable) class library?
What types of classes belong in class libraries?

What are the costs and benefits (development time, complexity, testing time, productivity) involved in reusing components?
When do programmers choose to reuse versus to write from scratch?

Generally, this research effort is attempting to create some reusability metrics for the object oriented paradigm to aid in the construction of reusable class libraries. The first step in this process is to determine how the C++ classes are being reused and what programmers want to reuse. These first results show that perhaps, well documented, simple, yet complete classes are more reusable than large, cryptic classes or classes that are hard to understand and use. Also, the more programmers reuse successfully, the more likely they are to reuse in the future.

References

1. Biggerstaff, T. and A. J. Perlis, Software Reusability Volume I, ACM Press Frontier Series, 1989.

2. Biggerstaff, T. and A. J. Perlis, Software Reusability Volume II, ACM Press Frontier Series, 1991.

3. Tracz, W., Tutorial: Software Reuse: Emerging Technology, The Computer Society Press of the IEEE, Washington D.C., 1988.

4. Henry, S. and M. Humphrey, "Comparison of an Object Oriented Programming Language to a Procedural Programming Language for Effectiveness in Program Maintenance," *Journal of Object Oriented Programming*, November, 1993.

5. Lewis, J. and S. Henry and D. Kafura and R. Schulman, "On the Relationship Between the Object Oriented paradigm and Software Reuse: An Empirical Investigation," *Journal of Object Oriented Programming*, August 1992.

6. Johnson, R. and B. Foote, "Designing Reusable Classes," *Journal of Object Oriented Programming*, June/July 1988.

7. Korson, T. and J. McGregor, "Technical Criteria for the Specification and Evaluation of Object Oriented Libraries," *Software Engineering Journal*, March 1992.

8. Meyer, B., "Reusability: The Case for Object Oriented Design," *IEEE Software*, March 1987, pp 50-64.

9. Henry, S., "A Project Oriented Course on Software Engineering," *Proceedings of the 1983 ACM SIGCSE Symposium on Computer Science Education*," February 1983, pp. 57-61.