Evaluating Code Reuse When Porting a Desktop Application to the Web

Zachary Forster, Isaac Schemm, David Spiegel, Matthew Wisby,
Joline Morrison and Mike Morrison
Department of Computer Science
University of Wisconsin-Eau Claire
Eau Claire, WI 54702
Corresponding author: morrisjp@uwec.edu

Abstract

This project explores measuring code reuse when porting an application from one platform to another. Previous accounts of code reuse focus on reusing code within the same application rather than porting an application's functionality and underlying code to an entirely different platform. In this project, we transported an existing desktop Java application to a web-based application. While doing so, we kept track of the amount of code from the old system that was able to be reused in the new application. We explored different ways to measure the amount of code that was reused, and present and discuss the results of these methods. We also identify the problems with measuring code reuse in this scenario.

1 Introduction

Many desktop applications have been written for tasks for which a Web application would be better suited, but no body of research exists exploring the issues involved while porting software from the desktop to the Web. Many programming languages commonly used to develop desktop applications have been augmented to support web programming as well, so it is often possible to re-use at least some of the existing code. This can reduce development time and costs. However, when porting a program from one interface (a desktop GUI) to another (a web site), the user interface must be changed, so it is first necessary to separate the main software code from the interface. The purpose of this research is to determine how much of a desktop application's code can be re-used when porting it to a Web application.

Our research project previously used a desktop application to support student peer reviews. This application is written as a desktop application in Java with a Swing GUI [1, 3]. During testing and use, two major issues arose: the user must always have the latest version of the software, and the user must be connected to the university network in order to reach the database server. We determined that a version of this software that runs as a web application would alleviate both of these issues: the software running on the web server would always be the latest version, and the web server (being on campus) would always be able to access the database server.

We chose Struts 2 as our method for creating the web application. We chose it mostly because we were familiar with it and found its method of controlling data flow easy to understand. However, we also wanted to use Struts because the model code is Java based and would be similar to the code for the old system. We thought that converting the system to another format that partially uses the same language would greatly increase the amount of code that could be reused.

In this paper, we first explore existing methods for measuring code reuse and analyze their applicability to our research. Next, we describe the methods we developed to measure code reuse, and present and discuss our results. Finally, we develop conclusions about measuring code reuse, and discuss future research possibilities.

2 Background

Previous researchers have identified quantitative metrics to measure code reuse. The most common is a simple comparison using lines of code: For example, Mascena et al. define "reuse percent" as measuring the ratio of reused lines of code to total lines of code in the system [4].

There are several papers in the literature about "code reuse" [e.g., 2, 4, 5, 6, 7] but most are not relevant to our research. For example, Waldo [7] uses the term "code reuse" to describe using code from a library. Devanbu [2] focuses on cost savings due to code reuse, and Sojer and Henkel [6] look at factors specific to open-source software that encourage reuse of code.Our specific case is relatively unrelated to all of these, because we are writing a new version of an existing application with a different front-end. The absence of

existing research was quite surprising to us, because Java is used in desktop applications, Web applications, and Android device applications. Due to the many common uses for the language, we would expect that porting a program from one environment to the other would be more common and would be addressed in previous research.

3 Measuring Code Reuse

When determining how to measure our code reuse, we initially had to choose between measuring the percentage of code from the old system that would bereused, or the percentage of the new system that would be made up of reused code.From a productivity standpoint, the percentage of code in any system that does not have to be written from scratch is of paramount importance, while the amount of code in an older system that is reusable is of dubious use. To illustrate this, imagine that an application is being ported from one platform to another, but that a great deal of extra functionality is being added in the process. The former code base would be quite small when compared to the latter. As a result, one might reuse 20 percent of the old code, while still being forced to write 95 percent of the new code from scratch. For the purposes of determining the amount of reused code present in our new system, we decided to compare two different metrics of code reuse: (1) the number of lines of code reused from the old system in the new system; and (2) and the number of blocks of reused code. Together, these measurements should lead to some interesting conclusions.

3.1 Blocks of Code

We loosely define a block of code as a class. We use this term loosely, because a large portion of our code for the web application was written in JSP, CSS, and JavaScript pages, none of which could be traced back to the original code. We decided that it was not acceptable to simply ignore these, because they demonstrated the true cost of rewriting an application with a different interface. Essentially, we considered each separate file to be a block, whether it was a Java class or a JavaScript file, and evaluated it accordingly.

The next step was deciding on how to identify the blocks. We decided to individually review each method within a class, and determine which methods (getters and setters included) were reused and which were not. If only a certain percentage of a method was reused, we simply recorded that percentage in decimal form, computed on a line-by-line basis. In the end, we were able to compute the total percentage of each class that was reused by dividing the number of reused methods by the total number of methods in the class. The non-Java pages caused no issues, because the percentage of reused code was always zero in these instances. This approach places less emphasis on the line-by-line accuracy of the final percentage, but instead provides valuable information about the types and functions of the large sections of code that are reused.

3.2 Lines of Code

While our method of counting blocks of code was somewhat subjective, we used a much more objective method for counting lines. We originally attempted to find a tool geared towards comparing similar lines in two different projects. We looked specifically at

CLOC (Count Lines of Code) [8] and UCC (Unified CodeCount) [9]. However, the available tools seemed to be directed at the comparisonof two versions of the same project. Since our two projects were so radically different, these tools were unable to correctly compare them.

Instead, we manually counted each line of code that was similar between the two projects. We ignored lines comprised only of whitespace or opening/closing brackets, so as not to skew anyof the results. Then, we proceeded to divide this number by the total number of lines of code in the new project. This measurement was obtained using a Perl script that looked for files with certain extensions (.java, .xml, .js, .jsp, .htm, .html and .css) and counted their lines. Two regular expressions were included to remove Java comments, while a third was used to exclude lines with only whitespace or brackets.

4 Results

4.1 Blocks of Code

When using the block method of counting code reuse, we found that of all the total blocks, only 6% were reused. When we isolated our count to only include the model code (not counting JSPs, CSSs, and JavaScript code) we discovered that 17% of blocks were reused. This is probably a better estimate of the total code reuse, since we never expected the view to be reused anyway.

4.2 Lines of Code

With the lines of code method, 10.56% percent reuse was found when including all of the "classes." When just the model was taken into consideration, 21.70% of the code was reused.

4.3 Results Compared

When comparing the two methods, it was found that the line method results in a higher percentage of reused code than the block method (see Figure 1). In particular, when all classes were taken into account, the percentage was almost doubled from block method to line method. This is not surprising considering that the Java Server Pages, which had no reused code, were only counted as one block using the block method. In contrast, the jsp emphasis was much stronger in the line method, since these pages were fairly lengthy in terms of lines.

In Figure 2, each Java class reuse percentage was compared side by side. The results from line method and block method were fairly consistent, except for a few cases. It is worthy to note that in general there was either a large or a very small percentage of code reused.

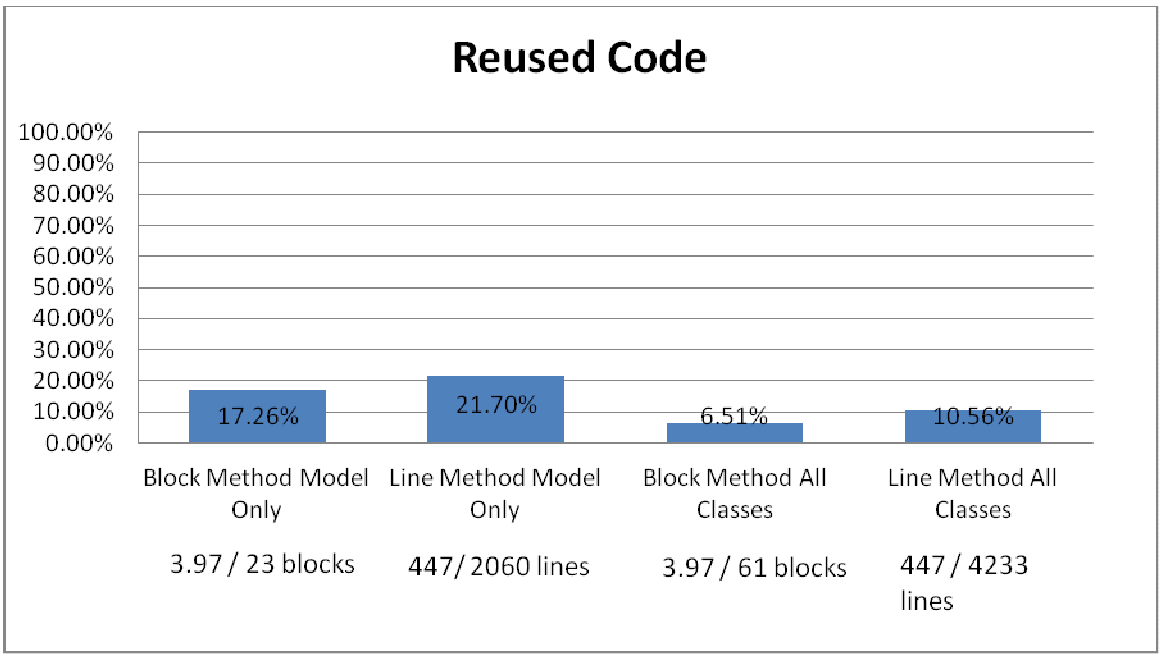

Figure 1: The percentage of reused code by blocks and lines, among .java source files (Model Only) and among all files (All Classes).

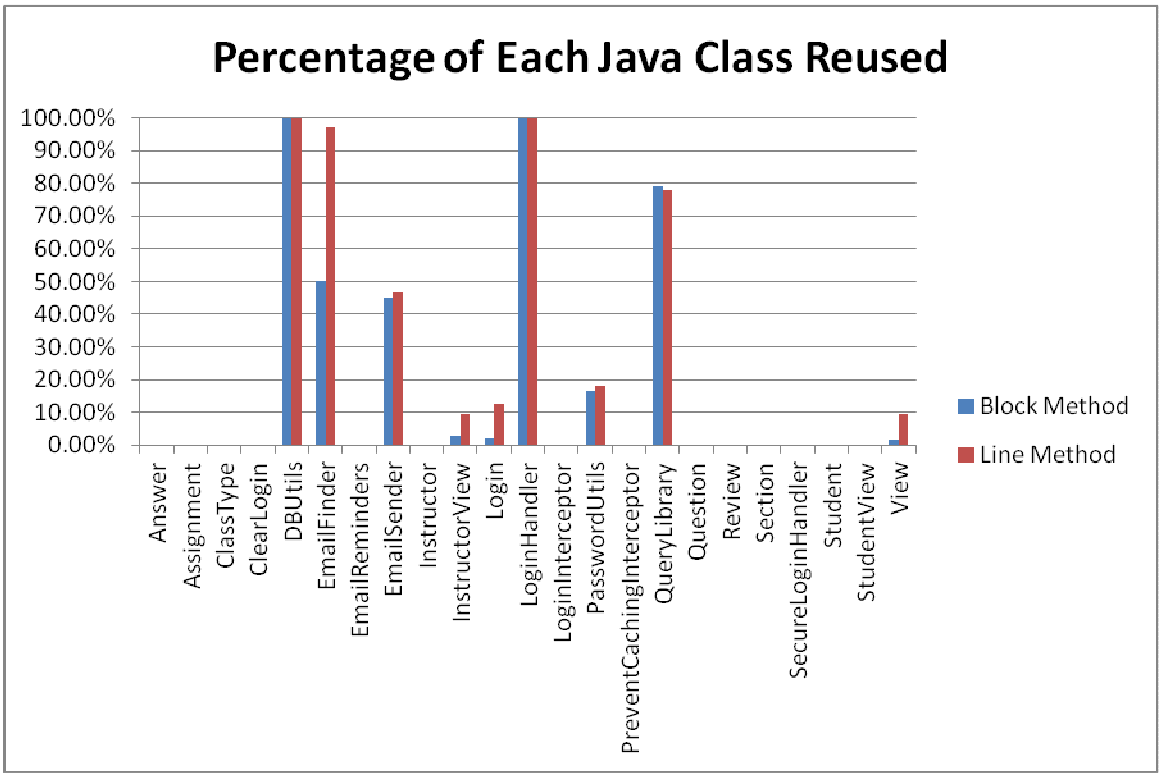

Figure 2: The percentage of each Java class that was reused, by methods and by lines.

4.4 Discussion

Our current Web application has 32 Java class files. Of these, nine classes contain at least some code reused from the old system. These classes can be subjectively divided into six categories, based on their function:

- **Isolated from Struts**
    - **Database** (DBUtils, QueryLibrary): The DBUtils class contained static methods for getting certain information from the database. Most of these methods were removed in our new system, but its method for creating a new database connection is still in use. QueryLibrary simply defines SQL queries in static strings, and many of these (170 lines worth) were reused.
    - **Authentication** (LoginHandler, SecureLoginHandler) – The LoginHandler interface existed in the old system. We used the interface unmodified, but rewrote the authentication code to let us store passwords in our database more securely. We could have used the old login code with the new system; conversely, we could also back port the new login code to the old system with no changes.
    - **Specialized functions** (EmailFinder, EmailSender, PasswordUtils) – These classes implement extra functionality that doesn't depend on the Struts framework. PasswordUtils contains static functions to generate and change passwords; 14 of the 86 lines of code were reused in this class. EmailFinder and EmailSender provide a way for instructors to email reminders to students about upcoming assignments. Most of the code for the email subsystem was reused (except for a new function that retrieves email server information from the database).
    - **Data classes** (Answer, Assignment, ClassType, Instructor, Question, Review, Section, Student) – These classes exist to store data in an object-oriented way within the program. All eight of these classes were created with the web application; none of them existed in the old system.
- **Dependent on Struts**
    - **Session handling** (ClearLogin, Login, LoginInterceptor, PreventCachingInterceptor) – These classes use the Struts session variable to ensure that users can't access the system without being logged in. Only 11 lines were reused – the lines that interface with the LoginHandler interface.
    - **Controllers** (DownloadUtils, EmailReminders, InstructorView, StudentView, View) – These classes contain functions that are used for Struts actions; therefore, they contain most of the program code. (View is

the parent class of both StudentView and Instructorview). This section consists of 1,114 lines of code. 74 lines were re-used (between View and InstructorView); the rest of the code was newly written for the web application.

5 Conclusions

Our findings indicate that porting a desktop application written in Java to the Web using Struts is not particularly efficient in terms of opportunities for code reuse. This stems from the heavy integration of the Swing code from the original system into the program logic. For this reason, we conclude that any attempt to port a Java desktop application to the Web will require heavy modification of the original code, and a good deal of code written from scratch. We think this would be true for other Java web platforms, including JSF and pure JSP files, because the need to replace Swing code with new front-end code is true for all of them. These findings are significant because of the new perspective that they provide with regard to code reuse.

As an increasing number of applications are developed as or ported to Web applications, research on the topic of code reuse when switching platforms will become increasingly relevant in the business world. Based on the results of this project, some interesting related projects could include more research into desktop to Web porting of applications written in languages other than Java, and porting using other available Web technologies. In addition, our findings have opened only a small window into the efficiency and productivity factors involved with porting applications to the Web. It would also be interesting to investigate advantages of having an existing code base to leverage from other than potential code reuse.

References

[1] Brinkerhoff, D., Komiskey, L., Miller, H., Morrison, J. and Morrison, M., "A Peer Review System to Enhance Collaborative Learning: Testing and Preliminary Evaluation," presented at MICS 2010 (Midwest Instruction and Computing Symposium), April 16-17, 2010, Eau Claire, Wisconsin (http://www.micsymposium.org/).

[2] Devanbu, Prem. "Analytical and empirical evaluation of software reuse metrics." *Proceedings of the 18th International Conference on Software Engineering* (1996): 189-199. doi:10.1109/ICSE.1996.493415

[3] Holt, B., Komiskey, L., Morrison, J. and Morrison, M, "A Peer Review System to Enhance Collaborative Learning," presented at MICS 2009 (Midwestern Instruction and Computing Symposium), April 2009, South Dakota School of Mines & Technology, Rapid City, South Dakota (http://www.micsymposium.org/)

[4] Mascena, J. C. C. P., de Almeida, E. S., & de LemosMeira, S. R. (2005). A comparative study on software reuse metrics and economic models from a traceability perspective. Proceedings of the IEEE International Conference on Information Reuse and Integration (2005), 72-77.

[5] Rothenberger, Marcus A., et al. “Strategies for software reuse: a principal component analysis of reuse practices.” *IEEE Transactions on Software Engineering* 29.9 (2003): 825-837. doi:10.1109/TSE.2003.1232287

[6] Sojer, Manuel, and Joachim Henkel. “Code Reuse in Open Source Software Development: Quantitative Evidence, Drivers, and Impediments.” *Journal of the Association for Information Systems* 11.12 (2010): 868-901. <http://papers.ssrn.com/sol3/papers.cfm?abstract_id=1489789>

[7] Waldo, Jim. “Code reuse, distributed systems, and language-centric design.” Fifth International Conference on Software Reuse, Proceedings (1998): 17-23. doi:10.1109/ICSR.1998.685726

[8] CLOC – Count Lines of Code. Northrop Grumman Corporation et al. Accessed March 14, 2013. <http://cloc.sourceforge.net>

[9] Unified CodeCount. USC Center for Systems and Software Engineering. Accessed March 14, 2013. <http://sunset.usc.edu/research/CODECOUNT>