Abstract:

It is believed that there are significant knowledge spillovers in Open Source Software (OSS). If such spillovers exist, it is likely they occur via two channels: In the first channel, programmers take knowledge know-how, and experience gained from one OSS project they work on and employ it in another OSS project they work on. In the second, programmers reuse software code by taking code from one OSS project and employ it in another OSS project.

In previous work, we found knowledge spillovers via the first channel. Focusing on the second channel, in this paper we develop a methodology to measure software reuse at the micro-micro level in a large OSS network. We then examine whether there are (spillover) benefits from software reuse. Finally, we examine which factors explain software reuse. Key findings involving software include the following:

- Controlling for other factors that explain success, projects that reuse code from a greater number of projects have higher success.
- Controlling for other factors that explain whether a project reuses code from other projects, younger projects are more likely to reuse code than older projects, while older projects are more likely to have their code reused.

Keywords: Reuse of Software Code, Knowledge Spillovers, Social Network, Open Source

References:

1 We gratefully acknowledge the financial support of the Israel Science Foundation (grant No. 1287/12 and 1069/15.) Any opinions expressed are those of the authors. We are grateful to Andrea Pozzi and Sarit Weisburd, and seminar participants at EIEF for helpful comments and suggestions.
1. Introduction

Product development in community-based organizations is becoming an increasingly important setting in which individuals create and disseminate knowledge in joint efforts to develop products. In such work environments, knowledge spillovers enable fellow software programmers, researchers and firms to benefit from innovations of others. Software programming is a vocation in which knowledge spillovers are likely to be important for product development given the rapid advancements in technologies, development methodologies, changing product-market preferences, and increasing competitive pressures.

Open Source Software (OSS) projects, like virtual teams, are semi-structured groups of skilled programmers working on interdependent tasks using informal, non-hierarchical, and decentralized communication with the common goal of creating a valuable product (Lipnack & Stamps, 1997). Virtual development teams, as opposed to traditional work teams that enjoy the benefits of face-to-face communication may also encounter challenges to form personal relationships (Beyerlein et al. 2001), to communicate (Pinto and Pinto 1990), and perform (Jehn and Shah 1997). Consequently, the resulting lack of strong connections and social support may have negative effects on productivity through reduced commitment, trust and leadership as well as willingness to share knowledge (Cascio 2000, Townsend et al. 1998; Whiting and Reardon 1998; Wong and Burton 2000). Accordingly, by the nature of its organizational design and structure, members of dispersed virtual development teams are restricted in their exposure to knowledge and know-how.

On the other hand, there are numerous advantages to the open source “team” model of innovation. In the case of OSS, the contribution of each individual programmer is known and measurable, since each addition or modification to the software is associated with a particular
Following the Code:

programmer. Hence, moral hazard problems that arise from joint output produced by teams (Holmstrom, 1982) are less likely to arise in OSS settings than in proprietary “cooperative” settings like research-based joint ventures. Additionally, OSS development teams make the underlying project knowledge accessible to the general population under a variety of OSS licenses (Laurent, 2004). Such licenses typically grant the rights to use the entire work, to create a derivative work, or to share or market such a work (Bonaccorsi, Rossi, & Giannangeli, 2006; Von Hippel & Von Krogh, 2003; Lerner & Tirole, 2002). Hence, intellectual property barriers in the form of patent thickets are less likely to adversely affect innovation in open source settings.

Indeed, one of the key benefits of OSS development is the ability to share and absorb knowledge that has been created outside of a distinct OSS project. Such spillovers facilitate the transfer of knowledge and ideas among individual programmers and across OSS projects. In particular, Open Source Software (OSS) can facilitate spillovers in R&D because the underlying software code is freely available and because programmers work on multiple projects. Knowledge spillovers in R&D in the case of OSS likely occur via two channels:

I. Spillovers from Software reuse: Programmers take software code from one project and employ it in another project.

II. Spillovers from Common Programmers: Programmers take knowledge, know-how, and experience from one or more OSS project they work on and employ that knowledge on another OSS project they work on.

The first channel includes (i) reuse from one project that a programmer is working on to another project he/she is working on as well as (ii) reuse from a project he/she is not working on to a project he/she is working on. The second channel includes knowledge, know-how, and experience, other than software reuse. A key question is whether these spillovers exist, i.e., whether the transfer of knowledge enhances other projects.
In previous work (Fershtman and Gandal 2011 and Gandal and Stettner 2016), we examined how connections among software projects via common programmers affected the success of OSS projects. We found evidence of positive spillovers, but (since we could not measure reuse) these spillovers included knowledge, know-how, experience, and reuse from another product the programmer is working to the relevant project. By directly measuring software reuse as well as network connections we can separately measure the importance of the two channels.

Until recently, however, measuring software reuse on a large scale remained beyond the abilities of the most powerful servers and most adept programmers. In this project we develop a methodology to measure software reuse at the micro-micro level in an open source network and to examine whether spillovers from software reuse affected the success of OSS projects. In section 2.2, we will describe in detail the methodology used to measure software reuse on a large scale. (Given the difficulty of the task and the huge number of files that needed to be compared, this process itself took nearly two years.) Key findings involving software reuse include the following:

- Controlling for other factors that explain success, projects that reuse code from a greater number of projects have higher success.
- Controlling for other factors that explain success, projects that supply code to more projects are no more successful than other projects. This makes sense, since the projects do not compete with each other.
- Controlling for other factors that explain whether a project reuses code from other projects, younger projects are more likely to reuse code than older projects, while older projects are more likely to have their code reused.

Prior research has examined the relationship between network structure and performance (Ahuja, 2000; Calvó-Armengol, Patacchini, & Zenou, 2009; Claussen, Falck, & Grohsjean, 2012.)
Our paper builds on Fershtman and Gandal, 2011 and Gandal and Stettner 2016).2 Using cross-sectional data, Fershtman and Gandal (2011) find that the structure of the product network is associated with the project’s success, which under the assumptions of the model, provides support for knowledge spillovers. Using panel data, Gandal and Stettner (2016) find that there are knowledge spillovers and that the number of additions and modifications are positively associated with project success.

In this paper, we first develop novel tools to quantify software reuse. We then calculate reuse measures for all projects in our data set, and examine whether (controlling for other factors) reuse of software is associated with project success.

Despite the perceived benefits from software reuse, there is no empirical work on reuse of software, the prevalence of reuse, and the potential benefits from reuse. The only empirical work on reuse of software code uses survey data. Based on a survey with OSS developers that yielded 686 responses, Sojer and Henkel (2010) find that developers with larger personal networks within the OSS community and those who have experience in a greater number of OSS projects reuse software more frequently.3

---

2 Other recent studies have examined the relationship between network structure and behavior (e.g., Ballester, Calvó-Armengol, & Zenou, 2006; Calvo-Armengol & Jackson, 2004; Goyal, van der Leij and Moraga-Gonzalez (2006); Jackson & Yariv, 2007; Karlan, Mobius, Rosenblat, & Szeidl, 2009).

3 There are also case studies about the benefits of software reuse. I. ACM Transactions on Software Engineering and Methodology for a case study about benefits in a telecom setting. II. “Orbotech, as part of the Israeli Software Reuse Industrial Consortium (ISWRIC), explored the possibilities of software reuse in a three-year project, supported by the Israel Ministry of Trade and Commerce. The positive results of the project made software reuse a common practice at Orbotech.” Quote taken from https://www.computer.org/csdl/proceedings/swste/2005/2335/00/23350110-abs.html
2. Measuring Knowledge Spillovers in OSS projects

2.1 Spillovers from Common Programmers

Direct knowledge spillovers occur when two projects have a common programmer who transfers knowledge, know-how and experience embedded in the code from one project to another. In contrast, indirect project spillovers occur when knowledge is transferred from one project to another when the two projects are not directly linked through a common programmer. For example, suppose that programmer "A" works on projects I and II, while programmer "B" works on projects II and III. Programmer A could take innovative code from project I and use it in project II. Programmer B might find that code useful – and port it from project II to project III. In such a case, knowledge is transferred from one project to another by programmers who work on more than one project. There is a direct spillover from project I to project II, and an indirect spillover from project I to project III, since projects I and III are not directly connected.

2.2 Spillovers from Software Reuse

Software reuse is often cited as one of the key benefits from open source software. Benefits include (i) improved software quality overall, (ii) improved functionality, (iii) consistency and interoperability across products, and (iv) the leveraging of technical skills and knowledge. To the best of our knowledge, we are the first to empirically “follow the code” from project to project. We do so by employing publicly available data from SourceForge, a platform that hosts tens of thousands of OSS projects and their programmers. “Following the Code” was a very difficult and time-consuming process and involved comparing a huge number of file pairs for

Following the Code:

similarity. In this paper, we chose to focus on one software language, JAVA. We focus on one language for two reasons: First, the scope of the language processing is tremendous requiring Trillions of “file-pair” comparisons for JAVA-based projects alone in our data set. Second, direct migration of code between different programming languages is not realistic. Despite the restriction to JAVA, we still had approximately 9,000,000 JAVA files and hence, we had to compare file similarity across approximately 81 trillion file pairs. The comparison is conducted at the level of the file.

Using custom software and large-scale text/data mining applications, we extracted and analyzed the log file for each JAVA-based project to determine the evolution of the software. Given the enormous size of the data set, all actions described below were executed by a set of custom programs written in Python and JAVA. The download and parse code were based on open source project code, which we took and modified to fit our setting.

We traced the progression of the source code across projects to determine whether a software file that was initially developed in project X was copied, in whole or part, and utilized in project Y. Our program traversed all project directories and extracted the text from text-based files including regular txt files, code files, word, and pdf. Some repositories contained zip compressed files that the program extracted and parsed as well. Along with the text, we extracted information on the last change made to the software file in the calendar year, the location of the file within the project’s file structure and other relevant parameters. We then saved the data in an XML structured file for easier post processing.

5 Comparing document A to B usually creates a different score than comparing B to A.
6 We constructed and analyzed the network itself (i.e., calculated network centrality measures using MySQL and Python.
7 Apache Solr™, Apache Lucene™
Following the Code:

For each project, we downloaded and parsed the files that had been added or changed within a calendar year.\(^8\) We started sampling from 31/12/2005, getting the latest version of all the files in the project up to that time. For each year after 2005, we employed the latest modified version of each file for each relevant calendar year. (Our data on downloads is from 2005 to 2008, so we have a four-year panel.)

We measure code similarity between two files based on the text of the code by examining function names, variable names, code fragments and comments within the code. To make this comparison, we used Apache Solr™, an Open Source distributed natural language processing engine based on Lucene™. These software applications are able to index very large numbers of text files, and provide searching capabilities over the text, similarity functionality and many other natural language functionalities.

Apache Solr™ employs a natural language processing methodology for searching for similarity between documents that is based on their vector space representation. Accordingly, every word in each document is assigned two scores: (1) Term Frequency (TF), which measures the number of times the word appears within the same document compared to all other words in the document and (2) Inverted Document Frequency (IDF), which measures the number of documents in the entire text universe (i.e., all files) in which the particular word appears. Thus, the importance of a word in a document is proportional to its TF score and inversely proportional to its IDF score. For example, in the context of this paper, the word “source” is important because it appears many times and signifies part of the topic of the article. On the other hand, the word “the”

---

\(^8\) Large files are files that are usually non-code files and those few large files that are code files were often created by automatic software that creates standard code. Thus, to make the processing of the enormous dataset possible, and contain the download time, network bandwidth and storage space on the servers we only kept the files below 200KB. There was a relatively small number of these files.
is less important, because it is common in the English language and appears in many other documents. Using a “standard” combined TF-IDF score of each word within a document (file,) we construct a representation vector of size K, where K is the number of distinct words. Each entry in K is the number of times each word appears in the document. We then calculate the cosine distance between the vector scores of all pairs of files across projects to determine the similarity between the documents.

Next, two trained software developers employed a binary search across pairs of randomly selected files (across projects) to identify the lower threshold for meaningful similarity scores. More specifically, approximately 30 randomly selected file-pairs with comparable (i.e., nearly identical) similarity scores were manually reviewed. If one or more of these file-pairs included a “false positive,” i.e., the software files did not involve reused software, the similarity score threshold was deemed inaccurate. We then randomly selected another set of 30 file-pairs from the population of files with higher (and comparable) “similarity scores.” The experts then reviewed these files. The process continued until there were no “false positives.” “This manual and tedious process involved cross consultation and careful deliberation; in this way, we established a conservative cut-off. We are extremely confident that all similarity scores above the cut-off involve reused software. In Appendix C, we provide an example of a file-pair that is above the cut-off.

It was important to us that the cutoff was determined by experts who examined the file pairs one by one. This way the cutoff is meaningful – and gives us ability to say something about the prevalence of software reuse in a large database of open source projects.

Armed with the similarity/reuse cutoff, we went over all 81 trillion pairs of files to denote those with similarity scores above the cutoff as “reused software.” To construct the “software reuse
Following the Code:

A flow network,” we arranged all files in chronological order. Thus, if two files are similar enough, so that the second one is defined as reused software, the one created first is considered the original file and the other the destination file. Thus, in this network there is a directed connection from file A.java to file B.java if (i) A.java was created before B.java, (ii) the pair had a similarity score above the cutoff (i.e., they were defined as reused software.), and (iii) and no other file C.Java functions as the origin file of B.Java.

We then constructed a “reuse” connection network between the projects where project X has a directed connection to project Y if there is at least one pair of similarity files belonging to these projects in that year such that the original file belongs to X and the destination file belongs to Y. Finally, we then add up all of the connections and define the variables reuse_in and reuse_out for each project. “Reuse_in” is the number of other projects from which that project reused software. “Reuse_out,” is the number of projects to which the project contributed software code.9, 10 Since we have data over time, an example helps make the dynamic definitions clear:

9 We can also define the weight of the connection between two projects as the number of similar files pairs created by project X and reused by project Y.
10 We used the NetworkX package to measure outgoing degrees and incoming degrees of each node in the “software reuse flow network.”
Following the Code:

Example: Construction of software reuse network variables (reuse_in and reuse_out)

Assume the following:

File X in project A was created in 2006
File Y in project B is a copy of X and was created in 2007
File Z in project C is a copy of X and was created in 2008

Then for project A:

2006 - reuse_in=0 ; reuse_out=0
2007 - reuse_in=0 ; reuse_out=1 (one other project copied from it through year 2007)
2008 - reuse_in=0 ; reuse_out=2 (two projects copied from it through 2008)

For project B:

2006 - reuse_in=0 ; reuse_out=0
2007 - reuse_in=1 ; reuse_out=0 (copied from one project through 2007)
2008 - reuse_in=1 ; reuse_out=0 (copied from one project through 2008)

For project C:

2006 - reuse_in=0 ; reuse_out=0
2007 - reuse_in=0 ; reuse_out=0
2008 - reuse_in=1 ; reuse_out=0 (copied from one project through 2008)

Note that if another project (say D) created file "W" in 2006 and that code was copied and used in project B in 2008, then for project B, for 2008, reuse_in=2. This is because project B reused software from two projects. If, however, “W” was created by project X and was copied and used in project B in 2008, reuse_in=1 for project B in 2008, since it copied software from just one project.

The whole process described in this section took nearly two years. Once we completed it, we proceeded with the analysis, which we describe in section 3.
3. Research Setting and Data

This paper uses data from Sourceforge.net, a free and accessible online platform for managing software development projects, facilitating developer collaboration and communication. Sourceforge.net is the largest repository of registered OSS development projects during the period of our study hosting tens of thousands of projects and their programmers. Each project links to a standardized “Project page” that lists descriptive information on a particular project, including a statement of purpose, software categories, intended audience, the license, and the operating system for which the application is designed. The most popular categories are “Internet Software”, “Development Software”, “System” and “Communications Software”. Other software categories are “Games/Entertainment” and “Scientific/Engineering” Software.11 Similarly, a standardized “Statistics page” shows various project activity measures, including the number of project page views and downloads registered for the project. Moreover, each OSS project contains a list of registered members who contribute their time and knowledge to the advancement of the project. Each project links to a standardized “programmer page” that contains meta-information on a particular programmer, including the unique user name, the date the programmer joined the project, the programmer’s functional description (e.g., administrator, programmer) and his or her geographic location.

3.1 Dependent Variable

Consistent with prior research, we measure project performance or success (denoted S) by examining the number of times a project has been downloaded. We focus on downloads of the

11 See Appendix D for details regarding products by software categories.
Following the Code:

executable, compiled product because end-users do not typically download the code. In the case of software, downloading code and getting it to work takes time and effort; engineers and computer scientists consider downloads to be an excellent proxy for success and the perceived quality of the product. Previous research (Grewal et al. 2006, Fershtman and Gandal 2011, and Gandal and Stettner 2016) has employed this measure. Although some data are available for other periods, statistics on downloads are available only for the 2005–2008 period. Therefore, we deploy yearly panel data from 2005–2008 in our analysis.

3.2 The Project and Programmer Networks

We constructed two distinct two-mode networks: (i) the project network and (ii) the programmer network. In the case of the project network in 2008, we find that 84.3% percent of the projects have either one or two programmers, 9.2% have three to four programmers and 6.5% have five or more programmers (see Table 1). With regard to the programmer network in 2008, 91.3% of the programmers worked on one or two projects, 6.5% of the programmers worked on three to four projects, and 2.1% of the programmers worked on five or more projects.

12 Recall that in the project network, the nodes are the OSS projects, and two projects are linked when there are common contributors who work on both. In the contributor network, the nodes of the contributor network are the contributors, and two contributors are linked if they participated in at least one OSS project together.

13 Percentages were virtually identical in other years as well.
Following the Code:

Table 1
Distribution of components in project networks—2008

<table>
<thead>
<tr>
<th>Project Network</th>
<th>Programmer Network</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Programmers</strong></td>
<td><strong>Percent of</strong></td>
</tr>
<tr>
<td>Per project</td>
<td>total projects</td>
</tr>
<tr>
<td>1</td>
<td>69.9</td>
</tr>
<tr>
<td>2</td>
<td>14.4</td>
</tr>
<tr>
<td>3-4</td>
<td>9.2</td>
</tr>
<tr>
<td>5-9</td>
<td>4.8</td>
</tr>
<tr>
<td>10 or more</td>
<td>1.7</td>
</tr>
</tbody>
</table>

Many empirical networks (including ours) consist of multiple distinct components: one very large component and several very small components. Indeed, our project network has one extremely large component (here forth "giant component") consisting of more than 14,000 connected projects in 2008 while the next largest component consists of less than 30 connected projects.

Whereas we focus on the project network, our analysis also includes a key feature of the programmer network: programmers who work on five or more projects. In the giant component, approximately 50 percent of the projects have one programmer who works on five or more projects. Indeed, we have defined this variable (five or more projects) so that no project has more than one programmer who works on five or more projects.

### 3.3 Degree and Closeness

While we do not directly observe spillovers, we adopt a simple model from Fershtman and Gandal (2011) allowing us to proxy spillovers by two network centrality measures: (i) a project’s degree, which is the number of projects with which the focal project has a direct link or common programmers, and (ii) a project’s closeness centrality, which is the inverse of the sum of all distances between a focal project and all other projects multiplied by the number of other projects.
Intuitively, closeness centrality measures how far each project is from all the other projects in a network and is calculated as:\(^{14}\)

\[
C_i = \frac{(N - 1)}{\sum_{j \in N} d(i, j)},
\]

where \(N\) is the number of projects and \(d(i, j)\) is the distance between project \(i\) and \(j\). For two projects that are directly connected, \(d(i, j) = 1\). For two projects that are indirectly linked via a third project, \(d(i, j) = 2\). In the case of a network with a single project that is connected to all other projects, the closeness centrality of that project equals 1, which is the maximum value for closeness centrality. Projects that indirectly link other projects have a higher closeness centrality measure than projects at the edge of a network.\(^{15}\) Since closeness is only defined for connected projects, we focus our analysis on projects in the giant component.

### 3.4 Model Specification

Having defined degree and closeness centrality as our proxies for spillovers we continue by assuming that the expected success level of each project “\(i\)” without any spillovers is given by

\[
S_{it} = \alpha_i + X_{it} \omega + \varepsilon_{it}.
\]

where the variable \(S_{it}\) is the success of project \(i\) at time \(t\), \(\alpha_i = \alpha + A_i' \delta\), where \(\alpha\) is a constant, \(A_i\) is a vector of unobserved time-invariant project factors, \(X_{it}\) is a vector of observable time-varying factors, and \(\varepsilon_{it}\) is an error term. There are likely many important unobserved time-invariant project factors (in the vector \(A\)) including project management structure, conditions potential programmers have to meet in order to join the project, and rules about who can make

---


\(^{15}\) Closeness centrality lies in the range \([0,1]\). In the case of a Star network with a single project in the middle that is connected to all other projects, the closeness centrality of the project in the center is one.
Following the Code:

edits and changes to the code. Given these important unobserved time-invariant project factors, equation (2) should be estimated using a fixed effects model in which $\alpha_i = \alpha + A_i'\delta$ is a parameter to be estimated. As Angrist and Pischke (2009) note, treating $\alpha_i$ as a parameter to be estimated is equivalent to estimating in deviations from means.16

Having a panel rather than cross-sectional data is advantageous, since a cross-section cannot control for time-invariant project effects; they are included in the error term in cross-sectional analysis. If these unobserved effects are correlated with the right-hand-side variables, the estimates from the cross-sectional analysis will be biased; however, we eliminate this problem by using fixed effect models. Further, as we show below, panel data enables us to develop a novel test for reverse causality. We believe that performing such a test is important when working with network data.

We adopt Fershtman & Gandal’s (2011) assumptions that (a) each project may receive a positive spillover denoted $\beta$ from all “connected” projects, and (b) that a project may enjoy positive spillovers from projects that are indirectly connected, but (c) that these spillovers are subject to decay that increases linearly as the distance between the projects in the projects network increases. When the distance between project $i$ and $j$ is $d(i,j)$, this spillover is $\gamma/\sum_j d(i,j)$. Under these assumptions, the success level of each project $i$ at time $t$ can be written

$$
S_{it} = \alpha_i + X_{it}\omega + \beta D_{it} + \gamma/\sum_j d(i,j) + \varepsilon_{it}.
$$

where $D_{it}$ is the degree of project $i$ in the network at time $t$, and $\beta$ and $\gamma$ are greater than or equal to zero. Using (1), the expression for closeness centrality, project $i$’s success at time $t$ can be rewritten as

$$
S_{it} = \alpha_i + X_{it}\omega + \beta D_{it} + \gamma C_{it}/(N-1) + \varepsilon_{it}.
$$

This spillover specification is simple but quite general. When $\beta$ and $\gamma$ equal zero, there are no spillovers at all. When $\beta>0$ and $\gamma=0$, there are only direct spillovers. When $\beta=0$ and $\gamma>0$, there

---

16 Fixed effects are also equivalent to estimating in differences if there are only two periods of data.
are both direct and indirect spillovers which are exclusively measured by the projects’ closeness centrality. When $\beta>0$ and $\gamma>0$, there are additional spillovers from directly connected projects above and beyond those captured by its closeness measure: the spillovers have a “hyperbolic” structure. Since we control for software reuse, the spillovers captured here are knowledge transfers other than software reuse from programmers in common across projects.

3.5 Functional Form

"Success" as measured by the number of downloads is skewed in our data, with a few projects having great success, and many others having less success. For this reason, we follow prior research in the network literature and use the natural log of downloads as the dependent variable (e.g., Clausen et. al, 2012; Fershtman and Gandal, 2011).

In our setting, several variables (modifications, additions, and software reuse) can take on zero values. In such a setting, it makes sense to use a log/linear model in which the independent variables enter linearly because variables with zero values are easy to handle.\(^{17}\) We denote the dependent variable as $\log$downloads $\equiv \ln(\text{downloads})$, where "ln" means the natural logarithm. All projects have at least one download in every year, so every project is included in the analysis.

3.6 Independent Variables

Project and Contributor Network Variables

For the empirical analysis, we use the following project network variables:

\(^{17}\) In the case of a log/log model, zero values on independent variables are more difficult to handle, since normalizations such as $\ln(x+1)$ are not innocuous for variables with very low means. We employ the natural log of years since the project began so that we can include controls for different years.
Following the Code:

- degree = degree of the project.
- closeness = closeness of the project

where project degree and project closeness were defined earlier.

From the contributor network, we include the following variable:

- The dummy variable "Many_Projects" takes on the value one if the contributor was a member of five or more projects. This variable stems from the programmer network rather than the project network. Clearly, having such a programmer join a project bestows that project with additional connections to other projects. An interesting question is whether adding such a programmer to the team of programmers has an effect on the success of a project beyond the effect it has on connectivity (i.e., network structure). Recall that no project has more than one such programmer.

Additions and Modifications

We also account for investment and effort in the project. Hence, we compute the number of modifications and additions made to the code for each project over the period between 2005 and 2008. A modification is defined as a change made by a programmer to existing code within a distinct file, while an addition occurs when a programmer adds a new file that contains a block of code that was not previously part of a focal OSS project. Thus, a modification captures an activity that affects a particular set of code with the desire to, for example, make the code more efficient or stable. Accordingly, modifications are a good proxy for incremental innovation that, for example, improve how the software product works via the refinement, reutilization, and elaboration of established ideas and technologies. Additions are a proxy for new knowledge and technologies that provide additional functionality (Lewin, Long, & Carroll, 1999). To better
Following the Code:

illustrate the process, we include an example of a modification in Appendix B. In that appendix, we also describe in greater detail how we classified modifications and additions.

For each project in each year, we count the number of modifications and additions. Hence, in 2008, the total number of modifications (additions) for each project is the sum of the modifications (additions) made during the 2005-2008 period.

- We define “Mods” as the number of modifications on the project.
- We define “Adds” as the number of additions to the project.

Software Reuse Variables

As noted in Section 2, we define the variable “Reuse_in” to be the number of other projects that “contributed” at least one software file to the particular project in a given year. The variable “Reuse_out” is defined to be the number of other projects that employ at least one software file from the particular project in a given year. A clear example of how these variables were constructed was included in the introduction.

We can also measure the weight of the software reuse link among projects. We define the variable “Reuse_in_2” to be the number of other projects that contributed at least two files of “reused” software to the particular project in a given year. Similarly, the variable “Reuse_out_2” is the number of other projects that employ at least two software files from the particular project in a given year.

Control Variables

In addition to the variables of interest, we have data for a group of control variables:
Following the Code:

- The variable years_since is defined as the number of years that have elapsed since the project was first launched on Sourceforge:
- The variable cpp is defined as the number of contributors that participated in the project:
- The data from Sourceforge.net include information on the six possible (formal) stages of development for each product. The stages are: 1 – Planning, 2 - Pre-Alpha, 3 – Alpha, 4 – Beta, 5 – Production/Stable, 6 – Mature. The variable stage takes on values between one and six.\(^{18}\)

Complete descriptive statistics and correlations among the variables” are in Appendix A.\(^{19}\)

4. Empirical Analysis

4.1 Informal Examination of the Data

Before we estimate any models, it is important to examine the data informally. In this project, we include projects written in JAVA. The variables “degree” and “closeness,” however, are calculated using the full network, since we also want to examine the spillovers that come from projects that have a programmer in common. In 2008, there are 3,276 JAVA projects in the giant component and 5,726 JAVA projects outside of the giant component with complete data. All the information below is from 2008.

Projects in the giant component have on average many more downloads than projects outside of the giant component (151,928 vs. 10,092). Further, projects in the giant component have on average (i) more contributors (4.84 vs. 1.89), (ii) a larger degree (7.06 vs. 1.35), and a great

\(^{18}\) A few of the projects have multiple stages listed. We exclude these projects from the analysis. Including these projects and taking the average stage as the stage of the project has no effect on the results.

\(^{19}\) Descriptive Statistics are in Table A1. Correlations the variables are in Table A2.
Following the Code:

number of contributors who work on five or more projects (0.52 vs 0.09). Additionally, projects in the giant component receive on average 1,396 modifications compared to 353 for projects outside of the giant component. Similarly, projects in the giant component receive on average 799 additions compared to 225 for projects outside of the giant component.

We now report the number of downloads conditional on various characteristics of the projects for projects in the giant component. We again do this for 2008.

In the giant component, the median number of downloads for projects with a contributor who worked on at least five projects was 2739, while the median number of downloads for projects without such a contributor was 1749. In the case of single contributor project, the median number of downloads was 952. In the case of projects with two contributors, the median number of downloads was 1,634, while the median number of downloads for projects with more than two contributors was 4,885.

The median number of downloads for projects with values of degree above the median was 3,458, while the median number of downloads for projects with values of degree below the median was 1,446. Similarly, the median number of downloads for projects with values of closeness above the median was 3,499, while the median number of downloads for projects with values of closeness below the median was 1,543.

Of those projects that had a least one addition, the median number of downloads for those with additions above the median was 6,802 while the median number of downloads for those projects with additions below the median was 2,600. Similarly, for those projects that had a least

---

Note that the mean number of downloads for projects with a contributor who worked on at least five projects was 267,545, while the median number of downloads for projects without such a contributor was 24,281. This illustrates how skewed the variable downloads is. For this reason, in this exploratory discussion, when we discuss downloads, we report medians, rather than means.
Following the Code:

one modification, the median number of downloads for those with modifications above the median was 9,006 while the median number of downloads for those projects with modifications below the median was 1904.

4.2 Informal Examination of Code Reuse (for 2008)

In the giant component, 17% of the projects reused code from other projects. Outside of the giant component, only 7% of the projects reused code from other projects. The percentages are essentially the same for reuse_out. These data are the first rigorous measures of the prevalence of software reuse in OSS projects.

For projects in the giant component that did not reuse software, the median number of downloads was 1,772 while for projects that reused software, the median number of downloads was 8,423. In the case of projects outside of the giant component, the median number of downloads for projects did not reuse software was 714 while for projects that reused software, the median number of downloads was 2,553.

The descriptive data from section 4.1 and 4.2 suggest that reuse_in, degree, “many_projects”, additions and modifications are positively correlated with success. Appendix A shows that is indeed the case. Table A2 shows that, not surprisingly, the correlations between these variables and success are higher for projects in the giant component. Table A2 shows that with the exception of cpp and degree (correlation 0.60,) correlations among the independent variables are relatively low.

5. Analysis
As discussed above, a "log/linear" model is appropriate for our analysis. Thus, we use the following equation and estimate it using a fixed effects model:

\[
\text{ldownloads} = \alpha_i + \beta_0 + \beta_1 \text{cpp} + \beta_2 \text{degree} + \beta_3 \text{closeness} + \beta_4 \text{Many\_Projects} + \beta_5 \text{Stage} + \beta_6 \text{lyears\_since} + \beta_7 \text{mods} + \beta_8 \text{adds} + \beta_9 \text{reuse\_in} + \beta_{10} \text{reuse\_out} + \beta_{11} \text{single} + \beta_{12} \text{YEAR2006} + \beta_{13} \text{YEAR2007} + \beta_{14} \text{YEAR2008} + \epsilon, \tag{5}
\]

where the variable single is a dummy variable that takes on the value 1 if the project only has a single contributor and zero otherwise and YEAR2006 is a dummy variable that takes on the value one if the observation is from 2006 and otherwise. YEAR2007 and YEAR2008 are similarly defined.

We report summary statistics and correlations among the independent variables in Appendix A.

### 5.1 Addressing Possible Endogeneity from Reverse Causality

Although our analysis focuses on how the network structure affects success, the reverse may be true as well: contributors may want to join popular/successful projects. Developers may want to be associated with very successful projects, thereby making the number of contributors and degree endogenous. The interpretation would be that developers may want to be associated with more successful projects. This "joining popular projects" effect would make the number of contributors and degree endogenous. Since our network is fairly thin, and since there are many

\[21\text{ We suppress the time subscript (t) and, except for the fixed effects, we suppress the project subscript (i) for ease of presentation in (5.) As noted above, since we employ year dummies, we use the natural log of project age.}\]
Following the Code:

projects and relatively few developers per project, it is likely that the "joining popular projects" is not an important phenomenon in our setting. Nevertheless, we would like to examine this issue.

The panel data set enables us to employ a novel test developed in Gandal and Stettner (2016) to investigate potential endogeneities. We do so by restricting the analysis to those projects that had no changes in the number of contributors from one year to another. In such a case, reverse causality (i.e., the effect that describes the tendency to join popular projects) is absent.\textsuperscript{22} The key point is that the degree can change for projects that have no changes in the number of their contributors. The mechanism by which this change can occur is that the degree centrality of the original project also increases when a contributor on a particular project joins another project.\textsuperscript{23}

While this test is intuitive, we lose one year of data when we employ it. Hence for the giant component, we report results both for the full data set (model #1 in Table 2) and for the case when we restrict attention to projects for which there is no change in the number of contributors (model #2 in Table 2.)

We estimate (5) for 2005-2008, the period for which we have data on downloads. The main results are shown in Table 2. In model #1 in Table 2, we include all observations in the giant component. In order to control for possible endogeneities, we also conduct the analysis only for projects for which $\Delta$cpp=0; in such a case, we lose the data for 2005 (model #2 in Table 2.)\textsuperscript{24}

\textsuperscript{22} Of course, it is possible that some contributors joined and some left with a net change of zero during the year, but the vast majority of projects had no changes or small changes in personnel from year to year. This is because, as noted, the number of contributors per project is quite small.

\textsuperscript{23} Similar to degree, the project closeness and the variable “Many_Projects” can also change even when the number of contributors on the project does not change.

\textsuperscript{24} In this case, we cannot include the dummy variable YEAR2006.
Following the Code:

Although we focus on the giant component, we also report results for projects outside of the giant component in Model #3 in Table 2. In the case of projects outside of the giant component, we report the results for the case when Δcpp=0.

5.2 Results

Spillovers from Software Reuse

While we discuss all three regressions, our main results are in Model #2 in Table 2. We find that the greater reuse of code from other OSS projects hosted at SourceForge, the more successful the project is. The result is highly significant for the giant component, both in models #1 and in model #2 in Table 2. This result is important and, to the best of our knowledge, provides the first econometric evidence of the prevalence of and the benefits from software reuse.

We also find that, controlling for other factors, greater reuse of a project’s code by other projects has no effect on the success of the project. This makes sense, since these projects do not compete with each other: they are all small open source projects hosted at SourceForge.

We would, of course, expect that reuse_out would be positively correlated with downloads, since people will want to copy from successful projects - and indeed the variables are positively correlated in the raw data. Hence, it is nice that when we control for other factors that lead to success, reuse_out is not significant in explaining success. If there was systematic unexplained variance in the regression, reuse_out would have been significant in the regression with success as the dependent variable. This gives us confidence that there is little or no systematic variance in the error term.

25 In the case of projects outside of the giant component, the estimated coefficient is positive, but not statistically significant.
Knowledge Spillovers across Connected Projects

All three models in Table 2 show that degree centrality is positively associated with the number of downloads and that this association is statistically significant. The estimated coefficient on degree suggests that there are direct project spillovers from projects connected by having a programmer in common. This result holds even after taking into account spillovers from software reuse.\textsuperscript{26} The positive coefficients on “reuse\_in” and degree suggest that both spillover channels discussed in the introduction provide benefits.

Other Variables:

Older projects and projects at a more advanced stage are associated with more downloads, both for projects in the giant component and projects outside of the giant component. In the case of the giant component, additional contributors are positively associated with success, i.e., the number of downloads. In the case of model #2, the number of modifications is positively associated with the success of the project, while the number of additions is negatively associated with success. The latter results may be due to the fact that major changes in the software are the result of problems with the project.

\textsuperscript{26} The insignificant estimated coefficients on closeness suggests that, controlling for other factors, indirect spillovers are not important.
Following the Code:

### Table 2: Results Explaining Success of OSS Projects

Dependent Variable: ldownloads

<table>
<thead>
<tr>
<th></th>
<th>Model 1</th>
<th>Model 2</th>
<th>Model 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Giant Component</td>
<td>Giant Component</td>
<td>Outside Giant</td>
</tr>
<tr>
<td></td>
<td>All Estimates (T-stats)</td>
<td>ΔCpp=0 Estimates (T-stats)</td>
<td>ΔCpp=0 Estimates (T-stats)</td>
</tr>
<tr>
<td>Single</td>
<td>-0.19 (-2.72*** )</td>
<td>-0.15 (-1.77')</td>
<td>-0.14 (-1.69')</td>
</tr>
<tr>
<td>Lyears_since</td>
<td>1.02 (23.74*** )</td>
<td>1.02 (28.46*** )</td>
<td>0.92 (38.22*** )</td>
</tr>
<tr>
<td>Degree</td>
<td>0.0055 (2.03*** )</td>
<td>0.0064 (2.73*** )</td>
<td>0.013 (2.99*** )</td>
</tr>
<tr>
<td>Closeness</td>
<td>0.27 (0.40)</td>
<td>-0.034 (-0.08)</td>
<td></td>
</tr>
<tr>
<td>Cpp</td>
<td>0.0083 (1.84*)</td>
<td>0.0090 (1.71*)</td>
<td>-0.0056 (-0.21)</td>
</tr>
<tr>
<td>Many_projects</td>
<td>0.043 (1.93*)</td>
<td>0.020 (1.35)</td>
<td>0.013 (0.68)</td>
</tr>
<tr>
<td>Stage</td>
<td>0.24 (4.81*** )</td>
<td>0.074 (1.72* )</td>
<td>0.15 (3.83*** )</td>
</tr>
<tr>
<td>Adds</td>
<td>9.4e-06 (1.03)</td>
<td>3.3 e-05 (-2.22*** )</td>
<td>-1.9e-05 (-1.63)</td>
</tr>
<tr>
<td>Mods</td>
<td>9.6e-06 (1.52)</td>
<td>7.8 e-05 (4.58*** )</td>
<td>1.2 e-04 (5.83*** )</td>
</tr>
<tr>
<td>Reuse_out</td>
<td>0.027 (2.22*** )</td>
<td>0.20 (2.97*** )</td>
<td>0.021 (1.25)</td>
</tr>
<tr>
<td>Year2006</td>
<td>0.0011 (0.32)</td>
<td>-0.00023 (-0.01)</td>
<td>0.0037 (0.45)</td>
</tr>
<tr>
<td>Year2007</td>
<td>0.080 (5.56** )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Year2008</td>
<td>0.11 (4.26*** )</td>
<td>0.13 (1.38)</td>
<td>0.054 (7.56*** )</td>
</tr>
<tr>
<td>Constant</td>
<td>4.92 (22.24)</td>
<td>5.45 (27.79)</td>
<td>4.71 (26.82)</td>
</tr>
<tr>
<td>Observations</td>
<td>12,790</td>
<td>8,300</td>
<td>16,600</td>
</tr>
</tbody>
</table>

* p < 0.05, ** p < 0.01, *** p < 0.001
We employ robust standard errors (without clustering)

6. Further Analysis: What Factors Determine Software Reuse?

An important question to address is what factors determine whether (I) A project reuses code and what factors determine (II) whether a project has its code reused. We do this for both the giant and non-giant component for 2008. In order to do so, we define the following dummy variables:

Reuse_out_dummy = 1 if the project had its code reused in 2008.

Reuse_in_dummy = 1 if the project reused code from other software projects in 2008.

We then estimate probit models using these code reuse variables as dependent variables.

The results in Table 3 are for 2008, the final year for which we have data:
Following the Code:

### Table 3: Results Explaining Software Reuse (Probit Analysis for 2008)

<table>
<thead>
<tr>
<th>Independent Variables</th>
<th>Model #1</th>
<th>Model #2</th>
<th>Model #3</th>
<th>Model #4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Giant Component</td>
<td>Outside of Giant</td>
<td>Giant Component</td>
<td>Outside of Giant</td>
</tr>
<tr>
<td></td>
<td>Reuse_out_dummy</td>
<td>Reuse_in_dummy</td>
<td>Reuse_out_dummy</td>
<td>Reuse_in_dummy</td>
</tr>
<tr>
<td>Single</td>
<td>-0.74 (-5.37***)</td>
<td>-0.88 (-6.14***)</td>
<td>-0.91 (-6.81***)</td>
<td>-0.81 (-6.30****)</td>
</tr>
<tr>
<td>Years Since</td>
<td>0.084 (2.59****)</td>
<td>-0.15 (-4.37****)</td>
<td>0.070 (1.83*)</td>
<td>-0.11 (-2.88****)</td>
</tr>
<tr>
<td>Degree</td>
<td>0.010 (1.28)</td>
<td>0.0071 (0.93)</td>
<td>0.060 (1.90*)</td>
<td>0.015 (0.44)</td>
</tr>
<tr>
<td>Closeness</td>
<td>7.67 (2.85****)</td>
<td>3.78 (1.38)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cpp</td>
<td>0.020 (2.34**)</td>
<td>0.021 (2.52**)</td>
<td>0.020 (0.81)</td>
<td>0.062 (2.72****)</td>
</tr>
<tr>
<td>Many Projects</td>
<td>0.093 (0.81)</td>
<td>0.11 (0.89)</td>
<td>0.070 (0.26)</td>
<td>0.18 (0.72)</td>
</tr>
<tr>
<td>Stage</td>
<td>0.16 (3.47****)</td>
<td>0.16 (3.29****)</td>
<td>0.13 (2.47****)</td>
<td>0.096 (1.98**)</td>
</tr>
<tr>
<td>Adds</td>
<td>5.97e-05 (2.35**)</td>
<td>2.64e-04 (8.26**)</td>
<td>1.23e-04 (3.86****)</td>
<td>3.19e-04 (6.97****)</td>
</tr>
<tr>
<td>Mods</td>
<td>4.89e-05 (3.20****)</td>
<td>-1.86e-05 (-2.57**)</td>
<td>7.50e-05 (2.34**)</td>
<td>-4.83e-05 (-5.59****)</td>
</tr>
<tr>
<td>Constant</td>
<td>-4.01 (-9.24)</td>
<td>-2.16 (-4.99)</td>
<td>-3.37 (-10.49)</td>
<td>-2.26 (-7.44)</td>
</tr>
<tr>
<td>Observations</td>
<td>3,276</td>
<td>3,276</td>
<td>5,732</td>
<td>5,732</td>
</tr>
</tbody>
</table>

The main results are as follows:

- Older projects are more likely to have their code reused. This holds for both projects in the giant component and projects outside of the giant component when the dependent variable is whether a project has its code reused in 2008 (Models #1 and #3 in Table 3.)

- The older the project, the less it reuses software from other projects. This result holds for projects in the giant component (Model #2 Table 3) and projects outside of the giant component (Model #4, Table 3.) In these models, the dependent variable is whether or not the project reuses code from other projects.

- In the case of the giant component, the greater the number of contributors, the more the project reuses software from other projects and the more its software is used by other projects.

- Position in the network (degree and/or closeness) is significantly associated with code reuse by others. This effect obtains for closeness in the case of the giant component and degree in the case of projects outside of the giant component. This effect obtains after controlling for the number of contributors.
Following the Code:

- The more mods a software project has, the less likely it is to reuse code from others. This could be measuring a substitution effect. We also find that the more mods a code has, the more likely it is to have its code reused. These results hold for projects in the giant component, as well as projects outside of the giant component.

We believe that these results are interesting and intuitive – and they give us a sense of the dynamics involved in software reuse.

7. Brief Conclusions and Future Directions for Research

In this paper, we examined the effect of the reuse of software on success in open source software projects. We found that, controlling for other factors that explain success, projects that reuse code from a greater number of projects have higher success. We also found that controlling for software reuse by a project, the degree of the project is significantly associated with project success. This suggests that projects receive additional (i.e., non-code) knowledge spillovers from connected projects. Thus, both channels (reuse of code and other knowledge spillovers from connected projects) yield spillover benefits.

We also examined what factors explain software reuse itself. We found that, controlling for other factors that explain whether a project reuses code from other projects, younger projects are more likely to reuse code than older projects, while older projects are more likely to have their code reused.

In future work, we hope to combine detailed data we are now collecting (from supplementary sources) on contributors (age, education, experience) to OSS projects with the data we employed in this project. This will provide insights into the knowledge flow itself. In particular, we can ask, how do people decide what to copy? Who does the copying and who copies from who? We leave these questions for future research.
References


Following the Code:


Following the Code:

Appendix A. Descriptive Statistics and Correlations

Table A1
Inside of Giant Component (Year = 2008)

<table>
<thead>
<tr>
<th>Variables</th>
<th>Observations</th>
<th>Mean</th>
<th>Std. Dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>downloads</td>
<td>3,276</td>
<td>151,928</td>
<td>5,420,344</td>
<td>6</td>
<td>308,000,000</td>
</tr>
<tr>
<td>years_since</td>
<td>3,276</td>
<td>5.87</td>
<td>1.56</td>
<td>2.98</td>
<td>9.13</td>
</tr>
<tr>
<td>degree</td>
<td>3,276</td>
<td>7.06</td>
<td>8.44</td>
<td>1</td>
<td>127</td>
</tr>
<tr>
<td>closeness</td>
<td>3,276</td>
<td>0.14</td>
<td>0.02</td>
<td>0.07</td>
<td>0.21</td>
</tr>
<tr>
<td>cpp</td>
<td>3,276</td>
<td>4.84</td>
<td>8.53</td>
<td>1</td>
<td>258</td>
</tr>
<tr>
<td>many_projects</td>
<td>3,276</td>
<td>0.52</td>
<td>0.50</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>stage</td>
<td>3,276</td>
<td>3.95</td>
<td>1.13</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>Adds</td>
<td>3,276</td>
<td>799</td>
<td>3,682</td>
<td>0</td>
<td>114,222</td>
</tr>
<tr>
<td>Mods</td>
<td>3,276</td>
<td>1,396</td>
<td>7,977</td>
<td>0</td>
<td>337,974</td>
</tr>
<tr>
<td>reuse_in</td>
<td>3,276</td>
<td>0.45</td>
<td>1.57</td>
<td>0</td>
<td>23</td>
</tr>
<tr>
<td>reuse_out</td>
<td>3,276</td>
<td>0.87</td>
<td>3.69</td>
<td>0</td>
<td>104</td>
</tr>
</tbody>
</table>

Outside of Giant Component (Year = 2008)

<table>
<thead>
<tr>
<th>Variables</th>
<th>Observations</th>
<th>Mean</th>
<th>Std. Dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>downloads</td>
<td>5,726</td>
<td>10,091.75</td>
<td>174,149.70</td>
<td>1</td>
<td>11,600,000.00</td>
</tr>
<tr>
<td>years_since</td>
<td>5,726</td>
<td>5.39</td>
<td>1.47</td>
<td>2.97</td>
<td>9.11</td>
</tr>
<tr>
<td>degree</td>
<td>5,726</td>
<td>1.35</td>
<td>2.12</td>
<td>0</td>
<td>24</td>
</tr>
<tr>
<td>cpp</td>
<td>5,726</td>
<td>1.89</td>
<td>1.96</td>
<td>0</td>
<td>31</td>
</tr>
<tr>
<td>many_projects</td>
<td>5,726</td>
<td>0.09</td>
<td>0.28</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>stage</td>
<td>5,726</td>
<td>3.75</td>
<td>1.17</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>Adds</td>
<td>5,726</td>
<td>225</td>
<td>1,607.27</td>
<td>0</td>
<td>73,651</td>
</tr>
<tr>
<td>Mods</td>
<td>5,726</td>
<td>353</td>
<td>6,466.83</td>
<td>0</td>
<td>475,308</td>
</tr>
<tr>
<td>reuse_in</td>
<td>5,726</td>
<td>0.13</td>
<td>0.76</td>
<td>0</td>
<td>27.00</td>
</tr>
<tr>
<td>reuse_out</td>
<td>5,726</td>
<td>0.23</td>
<td>1.75</td>
<td>0</td>
<td>48.00</td>
</tr>
</tbody>
</table>
**Following the Code:**

**Table A2**
Correlation among variables: giant component (Year=2008; N=3,276)

<table>
<thead>
<tr>
<th>Variables</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Downloads (1)</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Degree (2)</td>
<td>0.06</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Closeness (3)</td>
<td>0.05</td>
<td>0.42</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Cpp (4)</td>
<td>0.07</td>
<td>0.60</td>
<td>0.26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>many_projects (5)</td>
<td>0.02</td>
<td>0.49</td>
<td>0.29</td>
<td>0.15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Stage (6)</td>
<td>0.03</td>
<td>0.12</td>
<td>0.10</td>
<td>0.09</td>
<td>0.08</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Adds (7)</td>
<td>0.05</td>
<td>0.24</td>
<td>0.10</td>
<td>0.39</td>
<td>0.06</td>
<td>0.09</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Mods (8)</td>
<td>0.09</td>
<td>0.26</td>
<td>0.13</td>
<td>0.36</td>
<td>0.07</td>
<td>0.09</td>
<td>0.53</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>reuse_in (9)</td>
<td>0.15</td>
<td>0.17</td>
<td>0.08</td>
<td>0.24</td>
<td>0.07</td>
<td>0.07</td>
<td>0.39</td>
<td>0.23</td>
<td>1</td>
</tr>
<tr>
<td>reuse_out (10)</td>
<td>0.17</td>
<td>0.13</td>
<td>0.11</td>
<td>0.17</td>
<td>0.03</td>
<td>0.06</td>
<td>0.14</td>
<td>0.16</td>
<td>0.36</td>
</tr>
</tbody>
</table>

Correlation among variables: outside of giant component (Year=2008; N=5,726)

<table>
<thead>
<tr>
<th>Variables</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>downloads (1)</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Degree (2)</td>
<td>0.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Cpp (3)</td>
<td>0.06</td>
<td>0.15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>many_projects (4)</td>
<td>0.01</td>
<td>0.73</td>
<td>0.05</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Stage (5)</td>
<td>0.05</td>
<td>0.12</td>
<td>0.06</td>
<td>0.08</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Adds (6)</td>
<td>0.07</td>
<td>0.06</td>
<td>0.20</td>
<td>0.015</td>
<td>0.05</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Mods (7)</td>
<td>0.02</td>
<td>0.02</td>
<td>0.14</td>
<td>0.01</td>
<td>0.03</td>
<td>0.68</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>reuse_in (8)</td>
<td>0.02</td>
<td>0.07</td>
<td>0.15</td>
<td>0.04</td>
<td>0.03</td>
<td>0.33</td>
<td>0.04</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>reuse_out (9)</td>
<td>0.02</td>
<td>0.03</td>
<td>0.07</td>
<td>0.01</td>
<td>0.00</td>
<td>0.09</td>
<td>0.02</td>
<td>0.40</td>
<td></td>
</tr>
</tbody>
</table>
Following the Code:

Appendix B. Modifications and Additions

Source Code encapsulates a collection of computer instructions written in a human-readable computer language such as C++ or Java. Generally, these source code files are stored in a database of a source-code version control systems (VCS). Individual software programmers can add files containing source code to the VCS. Alternatively, programmers can retrieve existing files from the system and return modified version to the VCS that correct errors (i.e., bug fixes), make the code more efficient (i.e., require fewer processing power), make the code more stable to avoid crashes (e.g., Windows’s infamous Blue Screen of Death) or introduce enhancements. In fact, moderately complex software often requires the compilation of hundreds of different source code files each of which may have undergone dozens of modifications over time by different software programmers. In this study, we have gained access to the VCS of all software projects enabling us to track each addition and modification to the code by project.

27 The actions to be performed are generally transformed by a compiler program into low-level machine code (i.e., executable file) for execution at a later time. Most software applications, and in particular closed, proprietary software products, are distributed in a form that includes executable files, but not their source code.
Following the Code:

Example of a modification in Project aMSN

Project aMSN is an MSN compatible messenger application. Accordingly, on May 28, 2008 a user with username *square87* made a modification to file *guicontactlist.tcl*. This revision with unique identifier [r9986] is described in a comment by *square87* as “A minor code improvement.” The modification covers the deletion of some lines of code (indicated in red) and the addition of new lines of code (indicated in green).

For improved readability, empty lines and some comments have been removed from the source code. Note that an earlier modification to the file *guicontactlist.tcl* with unique identifier [r9911] is referenced in the text. It was made on May 22, 2008 by a different user whose username we have shortened to preserve privacy.

---

<table>
<thead>
<tr>
<th>Line #</th>
<th>a/trunk/amsn/guicontactlist.tcl</th>
<th>Line #</th>
<th>b/trunk/amsn/guicontactlist.tcl</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td># * change cursor while dragging (should we ?)</td>
<td>7</td>
<td># * change cursor while dragging (should we ?)</td>
</tr>
<tr>
<td>10</td>
<td># * ... cfr. &quot;TODO: &quot; msgs in code</td>
<td>10</td>
<td># * ... cfr. &quot;TODO: &quot; msgs in code</td>
</tr>
<tr>
<td>14</td>
<td>namespace eval ::guiContactList {</td>
<td>14</td>
<td>namespace eval ::guiContactList {</td>
</tr>
<tr>
<td>15</td>
<td>namespace export drawCL</td>
<td>15</td>
<td>namespace export drawCL</td>
</tr>
<tr>
<td>1050</td>
<td>if { [lindex $unit 1] == &quot;reset&quot; } {</td>
<td>1050</td>
<td>if { [lindex $unit 1] == &quot;reset&quot; } {</td>
</tr>
<tr>
<td>1051</td>
<td>set font_attr [font configure $defaultfont]</td>
<td>1051</td>
<td>set font_attr [font configure $defaultfont]</td>
</tr>
<tr>
<td>1052</td>
<td>} else {</td>
<td>1052</td>
<td>} else {</td>
</tr>
<tr>
<td>1053</td>
<td>set font_attr [font configure [lindex $unit 1]] }</td>
<td>1053</td>
<td>set font_attr [font configure [lindex $unit 1]] }</td>
</tr>
<tr>
<td>1055</td>
<td>array set current_format $font_attr</td>
<td>1055</td>
<td>array set current_format $font_attr</td>
</tr>
<tr>
<td>1056</td>
<td>} else {</td>
<td>1056</td>
<td>} else {</td>
</tr>
<tr>
<td>1057</td>
<td>array set current_format $font_attr</td>
<td>1057</td>
<td>array set current_format $font_attr</td>
</tr>
<tr>
<td>1058</td>
<td>array set modifications [lindex $unit 1]</td>
<td>1058</td>
<td>array set modifications [lindex $unit 1]</td>
</tr>
<tr>
<td>1059</td>
<td>foreach key [array names modifications] {</td>
<td>1059</td>
<td>foreach key [array names modifications] {</td>
</tr>
<tr>
<td>1060</td>
<td>set current_format($key) [set modifications($key)]</td>
<td>1060</td>
<td>set current_format($key) [set modifications($key)]</td>
</tr>
<tr>
<td>1409</td>
<td># Function that draws a contact</td>
<td>1408</td>
<td># Function that draws a contact</td>
</tr>
<tr>
<td>1410</td>
<td>proc drawContact { canvas element groupID } {</td>
<td>1410</td>
<td>proc drawContact { canvas element groupID } {</td>
</tr>
<tr>
<td>1411</td>
<td>} if { ${::guiContactList::external_lock}</td>
<td></td>
<td>!$::contactlist_loaded } { return }</td>
</tr>
<tr>
<td>1413</td>
<td># We are gonna store the height of the nicknames</td>
<td>1413</td>
<td># We are gonna store the height of the nicknames</td>
</tr>
<tr>
<td>1414</td>
<td>variable nickheightArray</td>
<td>1414</td>
<td>variable nickheightArray</td>
</tr>
<tr>
<td>1416</td>
<td>#Xbegin is the padding between the beginning of the contact and the left edge of the CL</td>
<td>1416</td>
<td>#Xbegin is the padding between the beginning of the contact and the left edge of the CL</td>
</tr>
<tr>
<td>1417</td>
<td>variable Xbegin</td>
<td>1417</td>
<td>variable Xbegin</td>
</tr>
<tr>
<td>1420</td>
<td>if { ${::guiContactList::external_lock}</td>
<td></td>
<td>!$::contactlist_loaded } { return }</td>
</tr>
</tbody>
</table>
Following the Code:

```tcl
set stylestring [list ]

if {$grId == "mobile"} {
  set nickstatespacing 5
  set statetext "([trans mobile])"
}

if {[$grId $groupID] == "1"} {
  set email [lindex $element 1]
  set grId $groupID
  set force_colour 1
  set psm [::abook::getpsmmedia $email 1]
  if {$force_colour && ![::MSN::userIsNotIM $email]} {
    set img [::skin::loadPixmap nonim]
  } elseif {[$grId $groupID] == "1" && ![::abook::getContactData $email MOB] == "Y" && $state_code == "FLN"} {
    set img [::skin::loadPixmap blocked_off]
  } else {
    set img [::skin::loadPixmap blocked]
  }
  if {$grId == "mobile"} {
    set update_img [::skin::loadPixmap space_update]
  } elseif {[$grId $groupID] == "1"} {
    set noupdate_img [::skin::loadPixmap space_no_update]
    #this is when there is an update and we should show a star
    set space_update [::abook::getVolatileData Semail space_updated 0]
    set space_shown [::abook::getVolatileData Semail SpaceShowed 0]
  }
```
Following the Code:

```tcl
# Check if we need an icon to show an updated space/blog; and draw one if we do
# We must create the icon and hide after else, the status icon will stick the border\n# it's surely due to anchor parameter
if { ![::MSNSPACES::hasSpace $email] } {
    lappend stylestring [list "tag" "$space_icon"]
    lappend stylestring [list "text" $statetext]
    lappend stylestring [list "colour" "reset"]
    if {$force_colour} {
        lappend stylestring [list "colour" "ignore"]
    }
}
else {
    # TODO : uncomment this line to get back the space needed for the support of MSN spaces.
    lappend stylestring [list "space" [image width $noupdate_img]] }
}

###Status icon###

# This is a technology demo, the default is not unchangeable
# values for this variable can be "inline", "ccard" or "disabled"
if {$space_shown && [::config::getKey spacesinfo "inline"] eq "both"} {
    lappend stylestring [list "newline" "n"]
    if {::con
```
## Following the Code:

### Appendix C: Example of File-Pair with Similarity Score above Cutoff

**Origin File:** Project: easim; File Size: 69.15kb; Filename: WorkStation.java; Creation date: 2005-12-20

**Destination File:** Project: desmoj; File Size: 71.96kb; Filename: WorkStation.java; Creation date: 2010-12-29

**Similarity Score:** 300 (maximum score is 1000)

**Important terms:** simprocess, partslist, capacity, workstation, slavequeues, queuebased, getquotedname, numofparts, enqueued, time, method, allsuitslaves, simtime, queuebased.fifo

**Overall Findings:** 1832 identical lines (omitted below), 23 new lines in destination (only in red), most are comments; 64 small semantic differences in other lines (red and black). (Functionality identical.)

### (Black color = in original and destination file, red color = only in destination file)

<table>
<thead>
<tr>
<th>Origin File</th>
<th>Destination File</th>
<th>File Contents (empty lines removed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>import desmoj.core.simulator.QueueListStandardFifo;</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>import desmoj.core.simulator.TimeInstant;</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>import desmoj.core.simulator.TimeSpan;</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>* @author Soenke Claassen</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>* @version DESMO-J, Ver. 2.0.1 copyright (c) 2005 licensed under GNU LGPL</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>* @version DESMO-J, Ver. 2.2.1 beta copyright (c) 2010</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>* @author Soenke Claassen</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>* Licensed under the Apache License, Version 2.0 (the &quot;License&quot;);</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>* you may not use this file except in compliance with the License. You</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>* may obtain a copy of the License at</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>* <a href="http://www.apache.org/licenses/LICENSE-2.0">http://www.apache.org/licenses/LICENSE-2.0</a></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>* Unless required by applicable law or agreed to in writing, software</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>* distributed under the License is distributed on an &quot;AS IS&quot;</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>* BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>* or implied. See the License for the specific language governing</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>* permissions and limitations under the License.</td>
<td></td>
</tr>
<tr>
<td>154</td>
<td>masterQueue = new QueueListStandardFifo(); // better than nothing</td>
<td></td>
</tr>
<tr>
<td>178</td>
<td>masterQueue = new QueueListStandardFifo(); // better than nothing</td>
<td></td>
</tr>
<tr>
<td>181</td>
<td>masterQueue = new QueueListFifo(); // better than nothing</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>masterQueue = new QueueListFifo(); // better than nothing</td>
<td></td>
</tr>
<tr>
<td>337</td>
<td>masterQueue = new QueueListFifo();</td>
<td></td>
</tr>
<tr>
<td>353</td>
<td>masterQueue = new QueueListStandardFifo();</td>
<td></td>
</tr>
<tr>
<td>405</td>
<td>process.setBlocked(false); // the process is not blocked anymore</td>
<td></td>
</tr>
<tr>
<td>900</td>
<td>* @return SimTime : Average waiting time of all processes since last reset</td>
<td></td>
</tr>
<tr>
<td>903</td>
<td>public SimTime mAverageWaitTime()</td>
<td></td>
</tr>
<tr>
<td>906</td>
<td>public TimeSpan mAverageWaitTime()</td>
<td></td>
</tr>
<tr>
<td>955</td>
<td>* @return desmoj.SimTime : Longest waiting time of a process in the master</td>
<td></td>
</tr>
<tr>
<td>958</td>
<td>public SimTime mMaxWaitTime()</td>
<td></td>
</tr>
<tr>
<td>967</td>
<td>* @return desmoj.SimTime : The point of simulation time when the process</td>
<td></td>
</tr>
<tr>
<td>984</td>
<td>* @return desmoj.core.TimeInstant : The point of simulation time when the process</td>
<td></td>
</tr>
</tbody>
</table>
Following the Code:

```
970  public SimTime mMaxWaitTimeAt() {
971  
987  public TimeInstant mMaxWaitTimeAt() {
999  * @return desmoj.SimTime : Point of time with minimum master queue length
1011  * @return double : The standard deviation for the master queue's processes
1023  * @return desmoj.core.TimeInstant : Point of time with minimum master queue length
1045  * @return desmoj.core.TimeSpan : The standard deviation for the master queue's processes
1066  if (currentlySendDebugNotes())
1082  if (currentlySendTraceNotes())
1090  if (traceIsOn()) {
1106  if (traceIsOn()) {
1123  if (!checkProcess(slave, where)) // if the slave process is not
1139  if (!checkProcess(slave, where)) // if the slave process is not
1155  * @return desmoj.SimTime : Average waiting time of all processes since last reset
1171  * @return desmoj.SimTime : The point of simulation time when the indicated
1183  * @return desmoj.SimTime : Point of simulation time when the indicated
1199  * @return desmoj.SimTime : Longest waiting time of a process in the slave
1215  * @return desmoj.core.TimeSpan : Longest waiting time of a process in the slave
1231  * @return desmoj.core.TimeSpan : The point of simulation time when the process
1247  * @return desmoj.core.TimeInstant : The point of simulation time when the process
1263  * @return desmoj.core.TimeInstant : Point of time with minimum slave queue length
1280  * @return desmoj.core.TimeInstant : Point of time with minimum slave queue length
1296  * @return desmoj.SimTime : Average waiting time of all processes since last reset
1312  * @return desmoj.SimTime : Longest waiting time of a process in the slave
1328  * @return desmoj.core.TimeSpan : Longest waiting time of a process in the slave
1344  * @return desmoj.SimTime : The point of simulation time when the process
1360  * @return desmoj.core.TimeInstant : The point of simulation time when the process
1376  * @return desmoj.SimTime : Point of time with minimum slave queue length
1392  * @return desmoj.core.TimeInstant : Point of time with minimum slave queue length
1408  * @return double : The standard deviation for the slave queue's processes
1424  * @return desmoj.core.TimeSpan : The standard deviation for the slave queue's processes
1441  * @return double : The standard deviation for the slave queue's processes
1457  * @return desmoj.core.TimeSpan : The standard deviation for the slave queue's processes
1473  if (traceIsOn()) // tell in the trace where the slave is waiting
1489  if (traceIsOn()) // tell in the trace where the slave is waiting
1505  * @return TimeSpan : Average waiting time of all processes since last reset
1521  * @return TimeSpan : Average waiting time of all processes since last reset
1537  * @return TimeSpan : Average waiting time of all processes since last reset
1553  * @return TimeSpan : Average waiting time of all processes since last reset
1569  * @return TimeSpan : Average waiting time of all processes since last reset
1585  * @return TimeSpan : Average waiting time of all processes since last reset
1601  * @return TimeSpan : Average waiting time of all processes since last reset
1617  * @return TimeSpan : Average waiting time of all processes since last reset
1633  * @return TimeSpan : Average waiting time of all processes since last reset
1649  * @return TimeSpan : Average waiting time of all processes since last reset
1665  * @return TimeSpan : Average waiting time of all processes since last reset
1681  * @return TimeSpan : Average waiting time of all processes since last reset
1697  * @return TimeSpan : Average waiting time of all processes since last reset
1713  * @return TimeSpan : Average waiting time of all processes since last reset
1729  * @return TimeSpan : Average waiting time of all processes since last reset
1745  * @return TimeSpan : Average waiting time of all processes since last reset
1761  * @return TimeSpan : Average waiting time of all processes since last reset
1777  * @return TimeSpan : Average waiting time of all processes since last reset
1793  * @return TimeSpan : Average waiting time of all processes since last reset
1809  * @return TimeSpan : Average waiting time of all processes since last reset
1825  * @return TimeSpan : Average waiting time of all processes since last reset
1841  * @return TimeSpan : Average waiting time of all processes since last reset
1857  * @return TimeSpan : Average waiting time of all processes since last reset
1873  * @return TimeSpan : Average waiting time of all processes since last reset
1889  * @return TimeSpan : Average waiting time of all processes since last reset
1905  * @return TimeSpan : Average waiting time of all processes since last reset
1921  * @return TimeSpan : Average waiting time of all processes since last reset
```

### Appendix D: Distribution of Software by Topics

<table>
<thead>
<tr>
<th>Topic</th>
<th>Relative frequency in single-programmer projects</th>
<th>Relative Frequency in multi-programmer projects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internet</td>
<td>16%</td>
<td>17%</td>
</tr>
<tr>
<td>Software Development</td>
<td>14%</td>
<td>14%</td>
</tr>
<tr>
<td>System</td>
<td>11%</td>
<td>13%</td>
</tr>
<tr>
<td>Communications</td>
<td>11%</td>
<td>10%</td>
</tr>
<tr>
<td>Games/Entertainment</td>
<td>11%</td>
<td>8%</td>
</tr>
<tr>
<td>Scientific/Engineering</td>
<td>7%</td>
<td>7%</td>
</tr>
<tr>
<td>Multimedia</td>
<td>8%</td>
<td>8%</td>
</tr>
<tr>
<td>Office/Business</td>
<td>5%</td>
<td>5%</td>
</tr>
<tr>
<td>Database</td>
<td>5%</td>
<td>5%</td>
</tr>
<tr>
<td>Other</td>
<td>13%</td>
<td>14%</td>
</tr>
</tbody>
</table>