Following the Code: Spillovers and Knowledge Transfer[1]

Neil Gandal
Berglas School of Economics, Tel Aviv University and CEPR,
gandal@post.tau.ac.il

Peter Naftaliev
Berglas School of Economics, Tel Aviv University, peternaftaliev@gmail.com

Uriel Stettner
Coller School of Management, Tel Aviv University, urielste@tau.ac.il

April 2018

[1] We gratefully acknowledge the financial support of the Israel Science Foundation (grants #1287/12 and #1069/15.) Any opinions expressed are those of the authors. We are grateful to Lukasz Ggrzybowski, the editor, and to an anonymous referee for comments and suggestions that significantly improved the paper. We also thank Andrea Pozzi and Sarit Weisburd, and seminar participants at EIEF, NYU, MIT, University of Warwick, and Tel Aviv University for helpful comments and suggestions.

Abstract:

Knowledge spillovers in Open Source Software (OSS) can occur via two channels: In the first channel, programmers take knowledge and experience gained from one OSS project they work on and employ it in another OSS project they work on. In the second channel, programmers reuse software code by taking code from an OSS project and employing it in another. We develop a methodology to measure software reuse in a large OSS network at the micro level and show that projects that reuse code from other projects have higher success. We also demonstrate knowledge spillovers from projects connected via common programmers.

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, it is believed that knowledge spillovers enable fellow software programmers and other projects to benefit from innovation in a particular project (Haefliger, Krogh, and Spaeth, 2007). 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).

The OSS model of innovation has accelerated over the last few years, as technology continuously makes it easier for programmers to work remotely. 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 knowledge spillovers in open source settings.

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 (Haefliger, Krogh, and Spaeth, 2008). Open Source Software (OSS) can facilitate knowledge spillovers in R&D because the underlying software code is freely available and because programmers work on multiple projects. In the case of OSS development, knowledge spillovers (if they exist) 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 incudes (i) reuse from one project that a programmer is working on to another project he or she is working on as well as (ii) reuse from a project that has no common programmers with the relevant project.

The second channel includes knowledge, know-how, and experience, other than software reuse. A key question is whether these spillovers exist in a large OSS network, and if they do, whether knowledge transfer enhances the performance of the projects involved.

In previous work we examined how connections among software projects via common programmers affected the success of OSS projects (Fershtman and Gandal 2011; Gandal and Stettner 2016), We found evidence of positive spillovers, but since we could not measure reuse on a large scale, these spillovers include knowledge, know-how, experience, and reuse from other projects the programmer is working on. By directly measuring software reuse as well as network connections we can separately measure the importance of the two channels.

Until recently, measuring software reuse on a large scale remained beyond the abilities of the most powerful servers and most adept programmers. In this project, we first develop a methodology for measuring software reuse at the micro level in a large open source network. In section 2.2, we describe in detail the methodology used to measure software reuse on a large scale[2]. We then calculate reuse measures for all projects in our data set, and examine whether reuse of

[2] Given the difficulty of the task and the huge number of files that needed to be compared, this process itself took nearly two years to complete

software is associated with project success (controlling for other factors). Our key findings are as follows:

- I. Projects that reuse code from a greater number of projects have more success. We then delineate reuse into two categories: (A) software reuse from connected projects, i.e., reuse from a project with a contributor in common with the relevant project and (B) software reuse from unconnected projects, i.e., reuse from projects without a contributor in common with the relevant project.

  An interesting stylized fact from our data is that most code reuse is not from neighboring projects, but rather from other projects in the network. We find that reuse from connected projects is not statistically significant in explaining success, while reuse from unconnected projects is statistically significant.

- II. We define two projects to be connected if they have a programmer in common. After controlling for software re-use effects, we find that projects with more direct connections with other projects have greater success. Since we have controlled for reuse of software by all projects (and connected projects in particular,) this means that there are knowledge spillovers via common programmers among projects that arise from factors other than code reuse.

  We control for the quality of the programmers on projects so this effect is not due to quality of programmers. Additionally, the open source projects in our data set are not well-known. They are hosted at SourceForge and do not have their own independent websites. Hence programmers on these

projects are not very visible – and projects do not get more downloads because of popularity or visibility. Thus, we are confident that this effect (like "I." above) is a knowledge spillover.

Overall, our results suggest that knowledge spillovers from neighboring products are primarily due to knowledge other than copying code, while "reuse" spillovers come from the general community of open source software projects. These results provide the first empirical support for knowledge spillovers via reused code in large open source software networks.

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).[3] 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.

Despite the perceived benefits from software reuse, there is virtually no empirical work on reuse of software, the prevalence of reuse, and the potential benefits from reuse. Employing a survey of 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. In a case study of six open

[3] 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).

source software projects, Haefliger, Von Krogh, and Spaeth (2008) find evidence of code reuse.[4] While these two studies provide some evidence of code reuse, they do not "follow the code" from project to project over time in a large open source in order to quantify knowledge spillovers from code reuse.

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 knowledge from project I and use it in project II. Programmer B might find that knowledge useful and take 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

[4] There are also a few "industry" case studies about the benefits of software reuse. Because of its participation in the Israeli Software Reuse Industrial Consortium (ISWRIC), Orbotech concluded that there were benefits from software reuse and software reuse became a common practice at Orbotech. See https://www.computer.org/csdl/proceedings/swste/2005/2335/00/23350110-abs.html

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, and (iii) interoperability across products.[5]

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 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.[6][7]

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

---

[5] See the Lombard Group, http://lombardhill.com/what_reuse.htm.

[6] Comparing document A to B usually creates a slightly different score than comparing B to A. There are approximately 9000 projects in our data and we have data for four years. Thus, on average, open source projects in our data have approximately 250 files.

[7] We excluded software libraries (which consist of pre-written software) from the analysis. Large files are files 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 large files.

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 A was copied, in whole or part, and utilized in project B. 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.

For each project, we downloaded and parsed the files that had been added or changed within a calendar year. 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 file is assigned two scores: (1) Term Frequency (TF), which measures the number of times the word appears

within the same file compared to all other words in the file and (2) Inverted Document Frequency (IDF), which measures the number of files in the entire text corpus (i.e., all files) in which the particular word appears. Thus, the importance of a word in a file 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 in this file and does not likely appear in many other economic papers. On the other hand, the word "the" is less important, because it is common in the English language and appears in many other files. 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 file. We then calculate the cosine distance between the vector scores of all pairs of files across projects to determine the similarity between the files.

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 nearly identical similarity scores (at the initial proposed threshold) 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 again reviewed thirty randomly selected file-pairs with comparable similarity scores. 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 A, 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 in detail. This 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 and denoted those with similarity scores above the cutoff as "reused software." To construct the "*software reuse 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 X.java to file Y.java if (i) X.java was created before Y.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 functions as the origin file of Y.Java.

We then constructed a "reuse" connection network between the projects where project A has a directed connection to project B if there is at least one pair of similarity files belonging to these projects such that the original file belongs to A and the destination file belongs to B. Note that if Project B copied from Project A, and Project C copied the same file from project B, project A gets credit as the source in both cases. In this case, project B is just a facilitator and does not get credit as the source.

Finally, we then added up all of the connections and defined the variables reuse_in and reuse_out for each project. "Reuse_in" is the number of other projects from which that project reused at least one software file. "Reuse_out," is the number of projects to which the project "contributed" at least one software file.

We also define variables that measure the number of files created by (say) project A and reused by project B.[8] These are important variables because they allow us to distinguish between reuse of software files and cloning (or forking) of

[8] We used the NetworkX package to measure outgoing degrees and incoming degrees of each node in the "software reuse flow network."

projects. For example, the variable Reuse_in_10 measures the number of other projects for which the relevant project uses 10 or more files. In the robustness analysis, we show that our results obtain even when we eliminate projects that reuse a large number of files from one other project.

An example helps make the definitions of these variables clear:

---

*Exa*mple: Construction of code 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 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 A 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. Since project B copied two software files (W and X) from project A, the variable “reuse_in_2” takes on the value 1 for project B in 2008.

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 the 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. Similarly, a standardized "Statistics page" shows various project activity measures, including the number of downloads 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 "programmer page" that contains meta-information on a particular programmer, including the unique user name, and the date the programmer joined the project.

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 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 (Fershtman and Gandal 2011; Gandal and Stettner 2016; Grewal et al. 2006) 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 employ yearly panel data from 2005–2008 in our analysis.

3.2 The Project and Programmer Networks

We constructed two distinct networks: (i) the project network and (ii) the programmer network. 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 are the contributors, and two contributors are linked if they participated in at least one OSS project together.

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.[9]

Table 1

Distribution of components in project networks—2008

Project Network		**Programmer Network**	
Programmers Per project	*Percent of total projects*	*Projects per programmer*	*Percent of total Programmers*
1	69.9	1	77.2
2	14.4	2	14.1
3-4	9.2	3-4	6.5
5-9	4.8	5-9	1.9
10 or more	1.7	10 or more	0.2

[9] Percentages were virtually identical in other years as well.

Many empirical networks (including ours) consist of multiple distinct components with one very large component and many very small components. Our project network has one extremely large component (here forth "giant component") consisting of more than 10,000 connected projects in 2008 while the next largest component consists of less than 100 connected projects.

In order to give a sense of what the connected network looks like, in Figure 1 in Appendix B, we show the largest "thickly" connected component, where, two projects are "thickly" connected if they have more than one contributor in common. It is feasible to show the largest thick component in a graph, since it has 276 projects, of which 77 are written using JAVA.[10] In the figure, projects written in JAVA are labeled. The figure shows that many of the JAVA projects are connected to each other, but other JAVA projects are connected to projects written in a different code. Spillovers among projects written in different programming languages will likely involve the transfer of knowledge rather than code reuse. In appendix B, we also briefly discuss four projects (highlighted in Figure 1) to demonstrate the variety of open source software projects hosted at SourceForge.

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. 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

[10] The largest thickly connected component is, of course, part of the giant component.

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 defined as:[11]

$$(1) \qquad C_i \equiv \frac{(N-1)}{\sum_{j \in N} d(i,j)},$$

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 than projects at the edge of a network.[12] Closeness is only defined for connected components.

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

$$(2) \qquad S_{it} = \alpha_i + X_{it}\omega + \varepsilon_{it}.$$

---

[11] See Freeman (1979), pp. 225-226.

[12] 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.

where the variable S_{it} is the success of project *i* at time *t*, $\alpha_i \equiv \alpha + A_i'\delta$, where α 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 ε_{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 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 $\alpha_i \equiv \alpha + A_i'\delta$ is a parameter to be estimated. As Angrist and Pischke (2009) note, treating α_i as a parameter to be estimated is equivalent to estimating in deviations from means.[13]

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 employ 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 β 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

[13] Fixed effects are also equivalent to estimating in differences if there are only two periods of data.

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

$$(3) \qquad S_{it} = \alpha_i + X_{it}\omega + \beta D_{it} + \gamma/\sum_j d(i,j) + \varepsilon_{it}.$$

where D_{it} is the *degree* of project *i* in the network at time t, and β and γ 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

$$(4) \qquad S_{it} = \alpha_i + X_{it}\omega + \beta D_{it} + \gamma C_{it}/(N-1)_t + \varepsilon_{it}.$$

This spillover specification is simple but quite general. When β and γ equal zero, there are no spillovers at all. When β>0 and γ=0, there are only direct spillovers. When β=0 and γ>0, there are both direct and indirect spillovers which are exclusively measured by the projects' closeness centrality. When β>0 and γ>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.

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; Gandal and Stettner 2016).

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.[14] We denote the dependent variable as ldownloads ≡ ln(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:

- 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 for the relevant project if one of its 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

[14] 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.

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 that may provide additional functionality (Lewin, Long, & Carroll, 1999). To better illustrate the process, we include an example of a modification in Appendix C. 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. An 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.

Similarly, the variable "Reuse_out_2" is the number of other projects that employ at least two software files from the particular project.

Control Variables

In addition to the variables of interest, we have data for control variables:

- 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 the number of project contributors:
- 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.[15]
- We have data on the type of license used. Two popular open source licenses (gpl and lgpl) are used by more than 70% of all projects. These licenses are "so-called" viral licenses, since software created by a project using either of these licenses has to remain in the public domain when used by other projects. Other popular open source licenses (like BSD) allow the user to decide whether the reused software will remain in the public domain. The dummy variable "viral" takes on the value one if the project uses a GPL or LGPL license and zero otherwise.
- Topic, Intended Audience and Operating System. These variables are employed as controls. While we include them in the regressions, we do not report estimated coefficients.

[15] 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.

Complete descriptive statistics and correlations among the variables" are in Appendix D1 and D2, respectively.

4. Empirical Analysis

4.1 Informal Examination of the Data

As discussed, in this analysis, we only 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 were 3,276 JAVA projects in the giant component and 5,726 JAVA projects outside of the giant component with complete data.

In 2008, projects in the *giant component* had on average many more downloads than projects outside the *giant* (151,928 vs. 10,092). Further, projects in the *giant component* had on average (i) more contributors (4.84 vs. 1.89), (ii) a larger *degree* (7.06 vs. 1.35), and a great number of contributors who work on five or more projects (0.52 vs 0.09). Additionally, projects in the *giant component* received on average 1,396 modifications compared to 353 for projects outside of the *giant component*. Similarly, projects in the *giant component* received on average 799 additions compared to 225 for projects outside of the *giant component*.

4.2 Informal Examination of Code Reuse

In the giant component, 17% of the projects reused code from other projects during the 2005-2008 period. 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 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.

4.3 Summary

As noted, we report summary statistics for the key variables used in the analysis in Appendix D (Table D1.) 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 D shows that this is indeed the case. Table D2 shows that the correlations between these variables and success are higher for projects in the giant component. With the exception of cpp and degree (correlation 0.60,) correlations among the independent variables are relatively low.

5. Analysis

Ideally, we would like to include both the giant and non-giant component projects in the same regressions. However, since closeness is only defined for connected components, we first ran several regressions with the giant component separately and included closeness as an explanatory variable. The estimated coefficient on closeness was not statistically significant in any of these regressions.[16]

Hence, in the paper, our regressions include both projects in the giant component and projects outside of the giant component together. Our main results on knowledge spillovers are robust to running separate regressions for the giant component and the other components. We report these regressions in Appendix E.

---

[16] The insignificant estimated coefficients on closeness suggests that, controlling for other factors, indirect spillovers are not important.

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:

$$(5)\ \text{ldownloads} = \alpha_i + \beta_0 + \beta_1\,\text{cpp} + \beta_2\,\text{degree} + \beta_3\,\text{Many\_Projects} + \beta_4\,\text{Stage} + B_5\,\text{lyears\_since} + \beta_6\,\text{mods} + \beta_7\,\text{adds} + \beta_8\,\text{reuse\_in} + \beta_9\,\text{single} + \beta_{10}\text{YEAR2006} + \beta_{11}\text{YEAR2007} + \beta_{12}\text{YEAR2008} + \beta_{13}\,\text{Giant} + \beta_{14}\,\text{Viral} + \varepsilon,$$ [17]

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. The dummy variable "giant" takes on the value one if the project is in the giant component and zero otherwise. We employ the natural log of years since the project began (denoted lyears_since in (5)) so that we can include dummy variables for different years.

In Regression #1 in Table 2, we estimate (5) for 2005-2008, the period for which we have data on downloads.

5.1 Robustness: 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,

[17] 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 years_since (project age.) We also include dummy variables for topic, intended audience, and operating system. See Appendix F for details regarding products by topic (i.e., software category.)

thereby making the number of contributors and degree endogenous. This "joining popular projects" effect would make the number of contributors and degree endogenous. Since our network is “thin,” and since there are many 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. In the robustness analysis, we restrict 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.[18] 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.[19]

Thus in order to control for possible endogeneities from reverse causality, we also conducted the analysis only for projects for which $\Delta cpp=0$, that is, for projects with no change in the number of contributors (regression #2 in Table 2.)[20]

5.2 Robustness: Forking or Cloning of Projects + Reuse from Non-Neighbors

An additional concern is that some software projects are “clones” or “forks” of other projects. In such a case, software reuse could be misleading. For this

---

[18] 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.
[19] Similar to degree, the variable “Many_Projects” can also change even when the number of contributors on the project does not change.
[20] In this case, we lose one year of data and cannot include the dummy variable YEAR2006.

reason, in regression #3 in Table 2, we exclude projects that "reuse" ten or more files from a single project. This is perhaps "overdoing it," since projects typically have on average more than two hundreds files. Nevertheless, in the robustness analysis, we want to be sure not to include projects that copied many files from a single project.

Reuse from Connected Projects vs. Reuse from Unconnected Projects

We then delineated the "reuse_in" variable into two categories: (I) software reuse from connected projects, i.e., reuse from a project from which the project has a contributor in common and (II) software reuse from unconnected projects, i.e., from projects from which the project does not have a contributor in common. In the giant component 16% of the projects reused code from unconnected projects, while 3% reused code from connected projects. In projects outside of the giant component, 6% of the projects reused code from unconnected projects, while less than 1% reused code from connected projects. The correlation between the two categories of reuse_in is 0.16.

5.3 Results

Spillovers from Software Reuse

We find that the greater the reuse of code from other OSS projects hosted at SourceForge, the more successful the project is. The result is highly significant in all three regressions 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. In regressions #1 and #2, we included all reuse in the variable "reuse_in."

In regression #3, we restrict reuse_in to reuse from non-connected projects and find that reuse from unconnected projects is statistically significant in explaining project success (coefficient = 0.041, t=2.31).[21]

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 highly 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.

The positive coefficients on "reuse_in" and degree suggest that both spillover channels discussed above provide benefits. The results from regressions #2 and #3 in Table 2 gives us confidence that these results are very robust.

Other Results:

Regarding the other variables included in the analysis, we have the following results

- Additional contributors are associated with success and this effect is robust across all three models in Table 2.
- The variable "Many_Projects" is associated with success and this effect is statistically significant in Regression #1 in Table 2. The effect is not, however, robust, since the estimated coefficient on this variable is positive, but not statistically significant in Regressions #2 and #3 in Table 2.

[21] When we also include reuse from connected projects as a variable in the regression (this regression is not shown,) reuse from connected projects is not statistically significant in explaining success (coefficient =0.0083, t=0.18.) This result from neighboring products is not necessarily beneficial.

- The number of modifications is positively associated with the success of the project and this effect is statistically significant in all three regressions in Table 2. The number of additions is not significantly associated with success.[22]
- Older projects and more mature projects are associated with more downloads.

Table 2: Fixed Effects Regressions: Explaining Success of OSS Projects

Dependent Variable: ldownloads	Regression 1 All Projects Estimates (T-stats)	Regression 2 Projects for which ΔCpp=0 Estimates (T-stats)	Regression 3 Projects excluding possible clones/forks, ΔCpp=0, reuse_in from non-neighbors Estimates (T-stats)
Single	-0.13 (-3.49***)	-0.10 (-2.30***)	-0.10 (-2.25***)
Lyears_since	0.93 (43.36***)	0.95 (49.11***)	0.95 (48.93***)
Degree	0.0094 (3.54***)	0.0083 (4.07***)	0.0082 (4.02***)
Cpp	0.0090 (1.90*)	0.011 (1.74*)	0.010 (1.64*)
Many_projects	0.052 (2.99***)	0.012 (1.01)	0.011 (0.94)
Stage	0.27 (8.53***)	0.13 (4.44**)	0.13 (4.51***)
Adds	1.5e-05 (1.42)	-2.1 e-05 (-1.89*)	-1.9e-05 (-1.61)
Mods	3.9e-06 (1.02)	8.8 e-05 (56.01***)	8.9 e-05 (6.02***)
Reuse_in	0.040 (2.89***)	0.023 (2.56***)	
Reuse_in (non)			0.041 (2.41***)
Viral license	0.071 (0.96)	-0.086 (-1.52)	-0.073 (-1.24)
Year2006	0.14 (15.64**)		
Year2007	0.20 (13.36***)	0.043 (7.76***)	0.043 (7.69***)
Year2008	0.17 (8.50***)	-0.0018 (-0.19***)	-0.0023 (-0.24)
Giant	0.045 (2.07**)	0.0031 (0.27)	0.0032 (0.28)
Observations	36,362	24,900	24,679

* $p < 0.10$, ** $p < 0.05$, *** $p < 0.01$; We employ robust standard errors (without clustering)

[22] The latter result may be because major changes in the software are the result of problems with the project. Or it may suggest that incremental innovation (modifications) is more important than drastic innovation (additions.)

6. Exploring Software Reuse

We would expect that "reuse_out" would be positively correlated with downloads, since people will want to copy software from successful projects - and the variables are positively correlated in the raw data. (See Table D2 in appendix D.) Indeed, the correlation between reuse_out and success is higher than the correlation between success and other variables.

Hence, it is nice that when we control for other factors that lead to success and include resuse_out, it is not significant in explaining success in any of the regressions in Table 2. If there was systematic unexplained variance in the regression, reuse_out would have been significant in the regressions with success as the dependent variable. This gives us confidence that there is little or no systematic variance in the error term.

The variable "Reuse_out" is an intermediate measure of success, since only high quality software will be reused. Indeed, the relatively high correlation between reuse_out and success affirms this: high quality software is being recused.

Since reuse_out is an "outcome" variable, we can also examine what factors determine whether a project has its code reused for 2008. In order to do so, we define the following dummy variable: Reuse_out_D equals one if the project had its code reused in 2008, and zero otherwise. We then estimate a probit model using this variable as the dependent variable. The results in Table 3 are for 2008, the final

year for which we have data. (We include all the control variables that were included in the regressions in Table 2.)

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

Dependent Variable: Reuse_out_D	
Independent Variables	All Projects (2008)
	Reuse_out_dummy
Single	-0.43 (-9.50***)
Years Since	0.039 (2.73***)
Degree	0.0097 (2.37**)
Cpp	0.012 (2.55**)
Many Projects	0.085 (1.54)
Stage	0.074 (4.14***)
Adds	4.5e-05 (4.74***)
Mods	2.5e-05 (3.86***)
Viral License	-0.080 (-1.90*)
Giant	0.25 (5.34***)
Pseudo R_sqaured	0.15
Observations	9002

* $p < 0.10$, ** $p < 0.05$, *** $p < 0.01$; Z-statistics in parentheses

From Table 3, the greater the number of contributors to a project, the more the project's software is reused by other projects. The greater the degree of the project, the more likely code is reused by others. This effect obtains after controlling for the number of contributors. That degree is significantly associated with code reuse by others provides additional evidence that there are knowledge spillovers.

Further, the more modifications and additions, the more the project's software is reused. Additionally, older projects are more likely to have their code reused. Projects in the giant component are more likely to have their code reused. Finally, projects using viral licenses are less likely to have their code reused. This is intuitive, since viral licenses restrict what can be done with the code.

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. In order to measure reuse, we had to compare file similarity across approximately 81 trillion file pairs. This was a huge multiyear project in itself. However, it was worth the effort!

- By carefully following the code among projects hosted at SourceForge over several years, we were able to provide the first estimate of actual software reuse. We found that in (outside of) the giant component, 17% (7%) of the projects reused code from other projects during the 2005-2008 period. These are the first rigorous measures of the prevalence of software reuse in OSS projects. This is important because up until this point, estimates of code reuse were based on surveys (i.e., Sojer, Manuel and Henkel, Joachim. 2010). While surveys are important, it is essential to begin providing actual estimates of reuse.
- We also found that software reuse was typically from unconnected projects. Less than 20 percent of the reuse was from connected projects (via a common innovator.)
- Reuse itself is an intermediate outcome variable. We found that there is a high correlation between (i) projects whose software is reused by other projects and (ii) success of the project that had its software reused. The

relatively high correlation between “reuse_out” and success affirms that high quality software is being recused.

Thus, even before we conducted our econometric analysis, we provided important “stylized” facts about reuse in open source software projects.

When we conducted the econometric analysis, we found that, controlling for other factors that explain success, projects that reuse code from a greater number of other projects have higher success. This is an important result and it suggests that taking knowledge from several different sources increases 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.

What might these knowledge spillovers be? Tucker (2008) studied adoption of video messaging technology and found that people who occupy connecting or central positions in the social network are more influential and have a larger effect on adoption (of technology) decisions than those who are not. Banerjee et al. (2013) examined the effect of (eigenvector) centrality on the spread of information and subsequent adoption of microfinance finance by village dwellers in India

showing indeed that information sharing and technology adoption spread more widely if originated in central nodes. Knowledge spillovers from connected software projects are likely similar.

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

Ahuja, G. 2000. “Collaboration Networks, Structural Holes, and Innovation: A Longitudinal Study.” *Administrative Science Quarterly* 45 (3): 425–55.

Angrist, J., and J. Pischke, 2009. *Mostly harmless econometrics: an empiricist’s companion.*

Ballester, Coralio, Antoni Calvó-Armengol, and Yves Zenou. 2006. “Who’s Who in Networks. Wanted: The Key Player.” *Econometrica* 74 (5): 1403–1417.

Banerjee, Abhijit, et al. "The diffusion of microfinance." Science 341.6144 (2013): 1236498.

Beyerlein, Michael Martin, Douglas A. Johnson, and Susan T. Beyerlein. 2001. *Virtual Teams*. Jai.

Bonaccorsi, A., C. Rossi, and S. Giannangeli. 2006. “Adaptive Entry Strategies under Dominant Standards: Hybrid Business Models in the Open Source Software Industry.” *Management Science* 52 (7): 1085–1098.

Calvó-Armengol, A., M. O. Jackson. 2004. The effects of social networks on employment and inequality. *American Economic Review* 94(3) 426–454.

Calvó-Armengol, Antoni, Eleonora Patacchini, and Yves Zenou. 2009. “Peer Effects and Social Networks in Education.” *The Review of Economic Studies* 76 (4): 1239–1267.

Cascio, W. F. 2000. Managing a virtual workplace. *Academy of Management Executive* 14(3) 81–90.

Claussen, J., Falck, O., & Grohsjean, T. 2012. The strength of direct ties: Evidence from the electronic game industry. *International Journal of Industrial Organization*, 30(2), 223-230.

Fershtman, C., & Gandal, N. 2011. Direct and indirect knowledge spillovers: the “social network” of open-source projects. *The RAND Journal of Economics*, 42(1): 70–91.

Freeman, LC. 1979. “Centrality in Social Networks: Conceptual Clarification.” *Social Networks* 1 (3): 215–39.

Gandal, Neil, and Uriel Stettner. 2016. “Network Dynamics and Knowledge Transfer in Virtual Organisations.” *International Journal of Industrial Organization* 48 (September): 270–90.

Goyal, Sanjeev, Marco J. Van Der Leij, and José Luis Moraga-González. 2006. “Economics: An Emerging Small World.” *Journal of Political Economy* 114 (2): 403–412.

Grewal, Rajdeep, Gary L. Lilien, and Girish Mallapragada. 2006. “Location, Location, Location: How Network Embeddedness Affects Project Success in Open Source Systems.” *Management Science* 52 (7): 1043–1056.

Haefliger, S., Von Krogh, G., & Spaeth, S. 2008. Code reuse in open source software. Management Science, 54(1), 180-193.

Hippel, Eric von, and Georg von Krogh. 2003. “Open Source Software and the ‘private-Collective’ Innovation Model: Issues for Organization Science.” *Organization Science* 14 (2): 209–223.

Holmstrom, B., 1982, Moral Hazard in Teams, Bell Journal of Economics, 13(2) 324-340.

Jackson, Matthew O., and Leeat Yariv. 2007. “Diffusion of Behavior and Equilibrium Properties in Network Games.” *The American Economic Review* 97 (2): 92–98.

Jehn, Karen A., and Priti Pradhan Shah. 1997. “Interpersonal Relationships and Task Performance: An Examination of Mediation Processes in Friendship and Acquaintance Groups.” *Journal of Personality and Social Psychology* 72 (4): 775.

Karlan, Dean, Markus Mobius, Tanya Rosenblat, and Adam Szeidl. 2009. “Trust and Social Collateral.” *The Quarterly Journal of Economics* 124 (3): 1307–1361.

Laurent, A. M.S. 2004. *Understanding Open Source and Free Software Licensing*. O’Reilly Media, Inc.

Lerner, J., and J. Tirole. 2002. “Some Simple Economics of Open Source.” *The Journal of Industrial Economics* 50 (2): 197–234.

Lewin, Arie Y., Chris P. Long, and Timothy N. Carroll. 1999. “The Coevolution of New Organizational Forms.” *Organization Science* 10 (5): 535–50.

Lipnack, J., and J. Stamps. 1997. *Virtual Teams: Reaching across Space, Time, and Organizations with Technology*. John Wiley & Sons Inc.

Pinto, Mary Beth, and Jeffrey K. Pinto. 1990. “Project Team Communication and Cross-Functional Cooperation in New Program Development.” *Journal of Product Innovation Management* 7 (3): 200–212.

Sojer, Manuel and Henkel, Joachim. 2010 Code Reuse in Open Source Software Development: Quantitative Evidence, Drivers, and Impediments," *Journal of the Association for Information Systems* 11(12).

Townsend, Anthony M., Samuel M. DeMarie, and Anthony R. Hendrickson. 1998. “Virtual Teams: Technology and the Workplace of the Future.” *The Academy of Management Executive* 12 (3): 17–29.

Tucker, C., 2008, "Identifying formal and informal influence in technology adoption with network externalities." Management Science 54.12: 2024-2038.

Whiting, V. R., and K. K. Reardon. 1998. “Communicating from a Distance: Establishing Commitment in a Virtual Office Environment.” *Academy of Management*.

Wong, Sze-Sze, and Richard M. Burton. 2000. “Virtual Teams: What Are Their Characteristics, and Impact on Team Performance?” *Computational & Mathematical Organization Theory* 6 (4): 339–360.

For Online Publication

Appendix A: 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)

Origin File	Destination File	File Contents (empty lines removed)
	11	import desmoj.core.simulator.QueueListStandardFifo;
	14	import desmoj.core.simulator.TimeInstant;
	15	import desmoj.core.simulator.TimeSpan;
18		* @author Soenke Claassen
22		* @version DESMO-J, Ver. 2.0.1 copyright (c) 2005 licensed under GNU LGPL
	25	* @version DESMO-J, Ver. 2.2.1beta copyright (c) 2010
	26	* @author Soenke Claassen
	28	* Licensed under the Apache License, Version 2.0 (the "License");
	29	* you may not use this file except in compliance with the License. You
	30	* may obtain a copy of the License at
	31	* http://www.apache.org/licenses/LICENSE-2.0
	32	*
	33	* Unless required by applicable law or agreed to in writing, software
	34	* distributed under the License is distributed on an "AS IS"
	35	* BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express
	36	* or implied. See the License for the specific language governing
	37	* permissions and limitations under the License.
	154	masterQueue = new QueueListStandardFifo(); // better than nothing
162		masterQueue = new QueueListFifo(); // better than nothing
	178	masterQueue = new QueueListStandardFifo(); // better than nothing
181		masterQueue = new QueueListFifo(); // better than nothing

	197	masterQueue = new QueueListStandardFifo(); // better than nothing
200		masterQueue = new QueueListFifo(); // better than nothing
	216	masterQueue = new QueueListStandardFifo(); // better than nothing
337		masterQueue = new QueueListFifo();
	353	masterQueue = new QueueListStandardFifo();
405		process.setBlocked(false); // the process is not blocked anymore
900		* @return SimTime : Average waiting time of all processes since last reset
	917	* @return TimeSpan : Average waiting time of all processes since last reset
903		public SimTime mAverageWaitTime() {
	920	public TimeSpan mAverageWaitTime() {
946		public SimTime mMaxLengthAt() {
	963	public TimeInstant mMaxLengthAt() {
955		* @return desmoj.SimTime : Longest waiting time of a process in the master
	972	* @return desmoj.core.TimeSpan : Longest waiting time of a process in the master
958		public SimTime mMaxWaitTime() {
	975	public TimeSpan mMaxWaitTime() {
967		* @return desmoj.SimTime : The point of simulation time when the process
	984	* @return desmoj.core.TimeInstant : The point of simulation time when the process
970		public SimTime mMaxWaitTimeAt() {
	987	public TimeInstant mMaxWaitTimeAt() {
989		* @return desmoj.SimTime : Point of time with minimum master queue length
	1006	* @return desmoj.core.TimeInstant : Point of time with minimum master queue length
992		public SimTime mMinLengthAt() {
	1009	public TimeInstant mMinLengthAt() {
1011		* @return double : The standard deviation for the master queue's processes
	1028	* @return desmoj.core.TimeSpan : The standard deviation for the master queue's processes
1014		public SimTime mStdDevWaitTime() {
	1031	public TimeSpan mStdDevWaitTime() {
	1097	if (currentlySendDebugNotes())
	1102	if (currentlySendTraceNotes())
1103		if (traceIsOn()) {
	1122	if (currentlySendTraceNotes()) {
1143		if (!checkProcess(slave, where)) // if the slave process is not
	1163	// not
1292		String allConds = new String("");
	1342	if (currentlySendDebugNotes())
	1347	if (currentlySendTraceNotes())
1349		if (traceIsOn()) // tell in the trace where the master is waiting
	1371	if (currentlySendTraceNotes()) // tell in the trace where the master is waiting
1481		* @return SimTime : Average waiting time of all processes since last reset
	1503	* @return TimeSpan : Average waiting time of all processes since last reset
1489		public SimTime sAverageWaitTime(int index) {
	1511	public TimeSpan sAverageWaitTime(int index) {
1688		* @return desmoj.SimTime : Point of simulation time when the indicated
	1710	* @return TimeInstant : Point of simulation time when the indicated
1696		public SimTime sMaxLengthAt(int index) {
	1718	public TimeInstant sMaxLengthAt(int index) {
1708		* @return desmoj.SimTime : Longest waiting time of a process in the slave
	1730	* @return desmoj.core.TimeSpan : Longest waiting time of a process in the slave
1716		public SimTime sMaxWaitTime(int index) {
	1738	public TimeSpan sMaxWaitTime(int index) {

1725		* @return desmoj.SimTime : The point of simulation time when the process
	1747	* @return desmoj.core.TimeInstant : The point of simulation time when the process
1734		public SimTime sMaxWaitTimeAt(int index) {
	1756	public TimeInstant sMaxWaitTimeAt(int index) {
1758		* @return desmoj.SimTime : Point of time with minimum slave queue length
	1780	* @return desmoj.core.TimeInstant : Point of time with minimum slave queue length
1767		public SimTime sMinLengthAt(int index) {
	1789	public TimeInstant sMinLengthAt(int index) {
1791		* @return double : The standard deviation for the slave queue's processes
	1813	* @return TimeSpan : The standard deviation for the slave queue's processes
1799		public SimTime sStdDevWaitTime(int index) {
	1821	public TimeSpan sStdDevWaitTime(int index) {
1842		if (!checkProcess(slave, where)) // if the slave is not a valid process
	1865	// process
	1892	if (currentlySendDebugNotes())
	1896	if (currentlySendTraceNotes())
1888		if (traceIsOn()) // tell in the trace where the slave is waiting
	1913	if (currentlySendTraceNotes()) // tell in the trace where the slave is waiting

Appendix B: Largest thickly connected network and descriptions of four projects

In order to give a sense of the variety of projects, below, we discuss four projects in the largest thickly connected component. These projects are highlighted in green in the figure. The values of the variables discussed below are for 2008.[23]

The Spring Framework project was founded in 2004 with the aim of providing a comprehensive programming and configuration model for modern Java-based enterprises. Spring focuses on the "infrastructure" of enterprise applications so that developers can focus on application-level business logic, without unnecessary ties to specific deployment environments. Its mature codebase

[23] The information and data discussed below were extracted from https://www.openhub.net/explore/projects

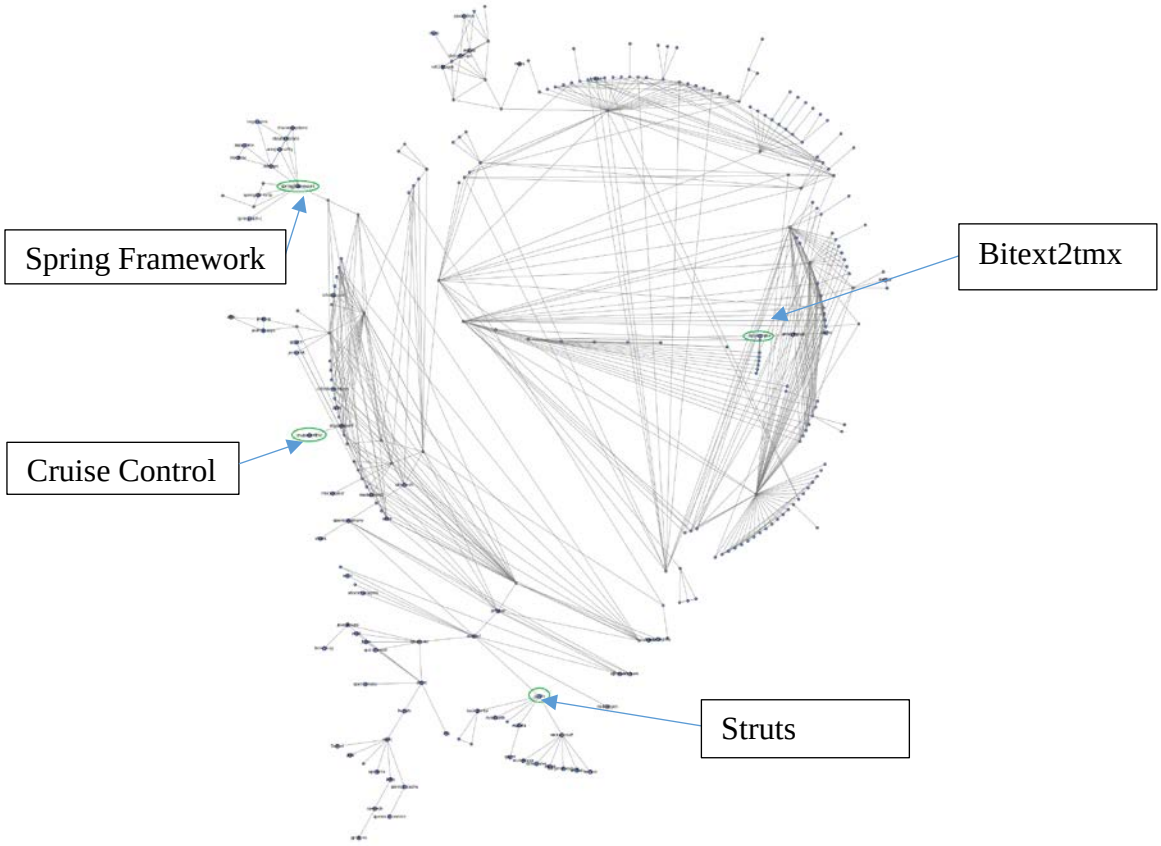
consists of 37,283 additions and modifications 1,239,948 lines of well-commented code. It had 416 contributors a degree of 54, and had received more than 3.25 Million downloads by 2008. By that year, eleven other OSS projects at SourceForge had reused its code. This project did not reuse code of other OSS projects at SourceForge.

Struts is a project for developing Java EE web applications that was started in 2005. The software itself is written in Java. By 2008, it had more than 6,000 additions and modifications from 89 contributors and a degree of 42. It includes almost 380,000 lines of well-commented code. By 2008, it had received more than 477,000 downloads and its software was reused by another open source project at SourceForge.

CruiseControl, which was conceived in 2001, is a framework written in Java for the automation of the process of compiling source code. It includes plugins for email notification and source control tools; it also provides a web-based user interface to view the details of the current and previous builds. By 2008, the software had more 4,000 additions and modifications and totaled about 177,000 lines of code of which about 30% of all lines in the code are comments. By 2008, it had 14 contributors and a degree of 11. By 2008, it had received more than 482,000 downloads. By that year, two other OSS projects had reused its code and it had reused code from two other OSS projects at SourceForge.

Bitext2tmx is a free computer-assisted translation (CAT) tool to align and converter bitext into translation memory format to be used in other CAT tools by translators and other language professionals. The software, conceived in 2002, had additions and modifications made by three contributors representing 5,674 lines of code. It is written in Java with a well commented and mature codebase. By 2008, it had received more than 6,500 downloads and its degree was 14. No other OSS project at SourceForge reused its code and it did not reuse code from other OSS projects at SourceForge.

Figure 1: Graph of largest thickly connected network


Spring Framework
Bitext2tmx
Cruise Control
Struts

Appendix C: Modifications and Additions

Source Code encapsulates a collection of computer instructions written in a human-readable computer language such as C++ or Java.[24] 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.

[24] 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.

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).[25]

MSN compatible messenger application
Commit [r9986]
A minor code improvement
Authored by: square87 2008-05-28
Changed: guicontactlist.tcl

Line #	**a/trunk/amsn/guicontactlist.tcl**	Line #	**b/trunk/amsn/guicontactlist.tcl**
7	# * change cursor while dragging (should we ?)	7	# * change cursor while dragging (should we ?)
10	# * ... cfr. "TODO:" msgs in code	10	# * ... cfr. "TODO:" msgs in code
12	::Version::setSubversionId {$Id: guicontactlist.tcl 9911 2008-05-22 21:51:56Z tom $}	12	::Version::setSubversionId {$Id: guicontactlist.tcl 9986 2008-05-28 07:43:18Z square87 $}
14	namespace eval ::guiContactList {	14	namespace eval ::guiContactList {
15	namespace export drawCL	15	namespace export drawCL
1050	if { [lindex $unit 1] == "reset" } {	1050	if { [lindex $unit 1] == "reset" } {
1051	set font_attr [font configure $defaultfont]	1051	set font_attr [font configure $defaultfont]
1052	} else {	1052	} else {
1053	set font_attr [font configure [lindex $unit 1]] }	1053	set font_attr [font configure [lindex $unit 1]] }
1055	array set current_format $font_attr		
1056	} else {	1055	} else {
1057	array set current_format $font_attr	1056	array set current_format $font_attr
1058	array set modifications [lindex $unit 1]	1057	array set modifications [lindex $unit 1]
1059	foreach key [array names modifications] {	1058	foreach key [array names modifications] {
1060	set current_format($key) [set modifications($key)]	1059	set current_format($key) [set modifications($key)]
1409	# Function that draws a contact	1408	# Function that draws a contact
1411	proc drawContact { canvas element groupID } {	1410	proc drawContact { canvas element groupID } {
		1411	i { ${::guiContactList::external_lock} \|\| !$::contactlist_loaded } { return }
1413	# We are gonna store the height of the nicknames	1413	# We are gonna store the height of the nicknames

[25] 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 whoes username we have shortened to preserve privacy.

1414	variable nickheightArray	1414	variable nickheightArray
1416	#Xbegin is the padding between the beginning of the contact and the left edge of the CL	1416	#Xbegin is the padding between the beginning of the contact and the left edge of the CL
1417	variable Xbegin	1417	variable Xbegin
1420	if { ${::guiContactList::external_lock} \|\| !$::contactlist_loaded } { return }		
1422	set stylestring [list]	1419	set stylestring [list]
1424	set email [lindex $element 1]	1421	set email [lindex $element 1]
1425	set grId $groupID	1422	set grId $groupID
1471	set force_colour 1	1468	set force_colour 1
1472	}	1469	}
1474	set psm [::abook::getpsmmedia $email 1]	1471	set psm [::abook::getpsmmedia $email 1]
		1473	if {[::MSN::userIsNotIM $email]} {
		1474	set img [::skin::loadPixmap nonim]
1476	if {[::config::getKey show_contactdps_in_cl] == "1" &&	1475	} elseif {[::config::getKey show_contactdps_in_cl] == "1" &&
1477	![::abook::getContactData $email MOB] == "Y" && $state_code == "FLN") &&	1476	![::abook::getContactData $email MOB] == "Y" && $state_code == "FLN")} {
1478	![::MSN::userIsNotIM $email]} {		
1479	set littlepic [::skin::getLittleDisplayPictureName $email]	1477	set littlepic [::skin::getLittleDisplayPictureName $email]
1480	set img ${littlepic}_cl	1478	set img ${littlepic}_cl
1482	catch { $img delete}	1480	catch { $img delete}
1483	image create photo $img	1481	image create photo $img
1504	$img copy [::skin::loadPixmap notnlist_emblem]	1502	$img copy [::skin::loadPixmap notinlist_emblem]
1505	}	1503	}
1507	} else {	1505	} else {
1509	if {[::MSN::userIsNotIM $email]} {		
1510	set img [::skin::loadPixmap nonim]		
1511	} elseif { [::MSN::userIsBlocked $email] } {	1507	if { [::MSN::userIsBlocked $email] } {
1512	if { $state_code == "FLN" } {	1508	if { $state_code == "FLN" } {
1513	set img [::skin::loadPixmap blocked_off]	1509	set img [::skin::loadPixmap blocked_off]
1514	} else {	1510	} else {
1515	set img [::skin::loadPixmap blocked]	1511	set img [::skin::loadPixmap blocked]
1516	}	1512	}
1542	if {$grId == "mobile"} {	1538	if {$grId == "mobile"} {
1543	set nickstatespacing 5	1539	set nickstatespacing 5
1544	set statetext "\([trans mobile]\)" } }	1540	set statetext "\([trans mobile]\)" } }
1548	set update_img [::skin::loadPixmap space_update]		
1549	set noupdate_img [::skin::loadPixmap space_noupdate]		
1551	#this is when there is an update and we should show a star		

1552	set space_update [::abook::getVolatileData $email space_updated 0]		
1554	#is the space shown or not ?		
1555	set space_shown [::abook::getVolatileData $email SpaceShowed 0]		
1557	set maxwidth [expr {[winfo width $canvas] - 2*$Xbegin - [::skin::getKey buddy_xpad] - 5}]	1544	set maxwidth [expr {[winfo width $canvas] - 2*$Xbegin - [::skin::getKey buddy_xpad] - 5}]
1560	# Beginning of the drawing	1547	# Beginning of the drawing
1577	lappend stylestring [list "tag" "icon"]	1564	lappend stylestring [list "tag" "icon"]
1579	###Space icon###	1566	###Space icon###
1582	if {[::config::getKey showspaces 1]} {	1569	set showspaces [::config::getKey showspaces 1]
		1570	if {$showspaces} {
		1572	#this is when there is an update and we should show a star
		1573	set space_update [::abook::getVolatileData $email space_updated 0]
		1575	#is the space shown or not ?
		1576	set space_shown [::abook::getVolatileData $email SpaceShowed 0]
		1578	set update_img [::skin::loadPixmap space_update]
		1579	set noupdate_img [::skin::loadPixmap space_noupdate]
1583	# Check if we need an icon to show an updated space/blog, and draw one if we do	1581	# Check if we need an icon to show an updated space/blog, and draw one if we do
1584	# We must create the icon and hide after else, the status icon will stick the border \	1582	# We must create the icon and hide after else, the status icon will stick the border \
1585	# it's surely due to anchor parameter	1583	# it's surely due to anchor parameter
1586	if { [::MSNSPACES::hasSpace $email] } {	1584	if { [::MSNSPACES::hasSpace $email] } {
1587	lappend stylestring [list "tag" "$space_icon"]	1585	lappend stylestring [list "tag" "$space_icon"]
1593	lappend stylestring [list "tag" "-$space_icon"]	1591	lappend stylestring [list "tag" "-$space_icon"]
1594	} else {	1592	} else {
1595	# TODO : uncomment this line to get back the space needed for the support of MSN spaces.	1593	# TODO : uncomment this line to get back the space needed for the support of MSN spaces.
1596	lappend stylestring [list "space" [image width $noupdate_img]] }	1594	lappend stylestring [list "space" [image width $noupdate_img]] }
1598	}	1596	
1599	incr marginx [image width $noupdate_img]	1597	incr marginx [image width $noupdate_img]
		1598	}
1602	###Status icon###	1601	###Status icon###
1700	lappend stylestring [list "tag" "state"]	1699	lappend stylestring [list "tag" "state"]
1701	lappend stylestring [list "text" $statetext]	1700	lappend stylestring [list "text" $statetext]
1702	lappend stylestring [list "tag" "-state"]	1701	lappend stylestring [list "tag" "-state"]
1704	lappend stylestring [list "colour" "reset"]	1703	# lappend stylestring [list "colour" "reset"]
1706	if {$force_colour} {	1705	if {$force_colour} {
1707	lappend stylestring [list "colour" "ignore"]	1706	lappend stylestring [list "colour" "ignore"]
1758	##Controversial inline spaces info##	1757	##Controversial inline spaces info##

```
1761     #This is a technology demo, the default is not unchangeable
1762     # values for this variable can be "inline", "ccard" or "disabled"
1763     if {$space_shown && \
1764         ([::config::getKey spacesinfo "inline"] == "inline" || \
1765         [::config::getKey spacesinfo "inline"] == "both") } {
1767         lappend stylestring [list "newline" "\n"]
1768         if {[::con
```

```
1760     #This is a technology demo, the default is not unchangeable
1761     # values for this variable can be "inline", "ccard" or "disabled"
1762     if {$showspaces && $space_shown && \
1763         ([::config::getKey spacesinfo "inline"] == "inline" || \
1764         [::config::getKey spacesinfo "inline"] == "both") } {
1766         lappend stylestring [list "newline" "\n"]
```

Appendix D. Descriptive Statistics and Correlations

Table D1

Inside of Giant Component (Year = 2008)

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

Outside of Giant Component (Year = 2008)

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

Table D2

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

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

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

Variables	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
downloads (1)								
Degree (2)	0.03							
Cpp (3)	0.06	0.15						
many_projects (4)	0.01	0.73	0.05					
Stage (5)	0.05	0.12	0.06	0.08				
Adds (6)	0.07	0.06	0.20	0.015	0.05			
Mods (7)	0.02	0.02	0.14	0.01	0.03	0.68		
reuse_in (8)	0.02	0.07	0.15	0.04	0.03	0.33	0.04	
reuse_out (9)	0.02	0.03	0.07	0.01	0.00	0.09	0.02	0.40

Appendix E: Fixed Effect Regressions for the Giant and Non-Giant Component Separately

Dependent Variable: ldownloads

	Model 1 Giant Component All Estimates (T-stats)	Model 2 Giant Component ΔCpp=0 Estimates (T-stats)	Model 3 Outside Giant ΔCpp=0 Estimates (T-stats)
Single	-0.19 (-2.72***)	-0.15 (-1.77*)	-0.14 (-1.69*)
Lyears_since	1.02 (23.74***)	1.02 (28.46***)	0.92 (38.22***)
Degree	0.0055 (2.03***)	0.0064 (2.73***)	0.013 (2.99***)
Closeness	0.27 (0.40)	-0.034 (-0.08)	
Cpp	0.0083 (1.84*)	0.0090 (1.71*)	-0.0056 (-0.21)
Many_projects	0.043 (1.93*)	0.020 (1.35)	0.013 (0.68)
Stage	0.24 (4.81***)	0.074 (1.72*)	0.15 (3.83***)
Adds	9.4e-06 (1.03)	3.3 e-05 (-2.22***)	-1.9e-05 (-1.63)
Mods	9.6e-06 (1.52)	7.8 e-05 (4.58***)	1.2 e-04 (5.83***)
Reuse_in	0.027 (2.22***)	0.020 (2.97***)	0.021 (1.25)
Reuse_out	0.0011 (0.32)	-0.000023 (-0.01)	0.0037 (0.45)
Year2006	0.080 (5.56**)		
Year2007	0.11 (4.26***)	0.13 (1.38)	0.054 (7.56***)
Year2008	0.061 (1.80*)	-0.045 (-2.83***)	0.014 (1.11)
Constant	4.92 (22.24)	5.45 (27.79)	4.71 (26.82)
Observations	12,790	8,300	16,600

* $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$, Robust standard errors (without clustering)

Appendix F: Distribution of Software by Topics

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