The Effects of Open Source License Choice on Software Reuse

John Brewer VIII

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

William B. Frakes, Chair
Gregory Kulczycki
Csaba Egyhazy

May 4, 2012
Falls Church, Virginia

Keywords: Open Source License, Software Reuse, Dependency Analysis, Package Repository

The Effects of Open Source License Choice on Software Reuse

John Brewer VIII

ABSTRACT

Previous research shows that software reuse can have a positive impact on software development economics, and that the adoption of a specific open source license can influence how a software product is received by users and programmers. This study attempts to bridge these two research areas by examining how the adoption of an open source license affects software reuse. Two reuse metrics were applied to 9,570 software packages contained in the Fedora Linux software repository. Each package was evaluated to determine how many external components it reuses, as well as how many times it is reused by other software packages. This data was divided into subsets according to license type and software category. The study found that, in general, (1) software released under a restrictive license reuse more external components than software released under a permissive license, and (2) that software released under a permissive license is more likely to be reused than software released under a restrictive license. However, there are exceptions to these conclusions, as the effect of license choice on reuse varies by software category.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The open source software movement was born from the idea that users and programmers should be free to modify existing software to meet their needs. In order for this to happen, source code must be freely accessible. This allows anyone to fix bugs or extend the software's functionality beyond that envisioned by its creators. It also promotes software reuse by allowing developers to use existing software rather than re-implementing a solution to a problem that has already been solved.

The history of the movement is filled with colorful characters debating a range of issues, including the definition of what it means to be open source. Software projects release their source code to the general public for a variety of reasons, such as to offer an alternative to proprietary applications, or to maximize the number of developers that can improve their product. As a result, the term *open source* can have several different meanings. There is plenty of rhetoric for and against these meanings, and many bold claims have been made regarding open source software's potential to strengthen individual liberties, reduce the cost of software development, or produce a superior product.

An individual's rights over a piece of software is described by the license under which the software is released. There are several open source licenses. What they all have in common is that they grant a user the right to access source code and make modifications. Where they differ is in the terms under which these changes may (or may not) be redistributed. These differences reflect the range of opinions by members of the open source community as to what it means to be open source. Aside from defining terms of use, an open source license is a project's way of establishing the underlying philosophy on software development.

To what extent do these ideological differences matter? Is there any way to measure the effectiveness of one software license type over another? Much has been written on the potential consequences of including certain rights over others in a license, but are such thoughts justified? Is there any characteristic of an open source license that has been

shown to be more effective at promoting the ideals of the movement, to the extent they can be defined, than others?

It is doubtful that there can ever be a definitive answer to any of these questions, particularly as they relate to abstract ideas of freedom. However, we can learn more about the efficacy of various licenses by measuring the indicators of success exhibited by projects using them. Many such studies have already been performed, and will be reviewed below. This paper is an attempt to examine the effect a project's choice of license has on the reusability of their product. Are any open source licenses better than others for increasing reuse?

1.1 Practical Reuse

Software reuse is the creation of new software from existing code, documentation, or other objects[1]. Reuse can be measured at many different levels of abstraction. Some studies of open source software reuse are performed at the source code level; that is, they seek to identify cases where the text in the source file of one project appears in another. By granting access to source code, open source licenses allow, and seem to encourage, this behavior. For example, suppose an application requires the ability to export data in a compressed format. A developer could locate an open source application that has such a feature, locate the compression algorithm it uses in the source code, and copy it into his own. However, such a practice occurs infrequently. A more likely approach would be to locate an open source component that provides the function and incorporate it into the application's design. Here the term *component* is general in scope and can refer to a variety of software objects, including a collection of library functions, a web service, or some other unit encapsulating specific functionality. The developer could use a component such as *zlib*[1] by calling it from within the application and achieve without ever having to access its code.

This approach offers many benefits. For one, it limits the amount of implementation-specific details the developer needs to learn in order to get his compression feature. It is usually simpler to learn and use a component's application programmer interface (API) than it is to trace through and fully understand its source. For another, it prevents unnecessary duplication. Instead of having two versions of a compression algorithm (the original and the one copied into the new program), there is only one. Any bug fixes or updates to the original need only be applied once.

Such component reuse is very common in open source development. It has been observed [2] that this approach appeals to the social ethics of open source programmers. A single solution developed by many programmers is likely to be more effective than a solution

[1] http://zlib.net/

made by an individual. The open source community as a whole benefits when developers decide to use and contribute changes to an existing component instead of creating a one-off version for themselves. This suggests that identifying how often a component is used in the creation of a new application is a good way to measure the practice of reuse within the open source development community.

1.2 Identifying Open Source Software

Over the past 15 years, a tremendous amount of open source software has been released. The success of products like Linux and the Apache web server has demonstrated the ability of open source development techniques to produce reliable business-class applications. Open source software has also introduced new economic models for generating income by giving away source code, as shown by the success of the Android operating system for cell phones. Access to source code, along with virtually free distribution over the Internet, have allowed an increasing number of contributors to use and maintain free software. In almost every application domain, there exist open source tools that provide a lower cost or higher quality (or both) alternative to proprietary tools produced using traditional development methods. This has obvious benefits to users, both personal and commercial. It is also beneficial to researchers of development methods. Open source projects typically do their work in full view of the public. Researchers can access communications between developers and users over mailing lists and message forums, view defect logs, study the history of the source code, read design documentation, and observe just about any part of the development process.

Unfortunately, the sheer number of projects has led to challenges in collecting data about open source software as a whole. The distributed nature of open source development means that there is no one source for all information about the various software projects. Complicating this issue further is that these projects reflect a wide range of quality and reliability. The low cost of entry for launching an open source project means that for each robust open source product like the Linux kernel, there are hundreds, if not thousands, of projects that never made it out of the alpha stage or have been abandoned. One of the challenges faced by researchers is identifying software to include in their studies. Depending on the questions being investigated, finding quality software that meets the requirements of the study can be the most difficult part of conducting a research project.

Many hosting services provide free or low cost homes to open source projects. These sites provide development tools such as code revision control systems and documentation tools (such as a wiki), and place to host files for download. In short, the give a project a home on the web. These sites are frequently referred to as “forges”, in refer-

ence to SourceForge[2], which at 300,000 projects[3] is the largest of such sites. Similar services are provided by Freecode (formerly FreshMeat)[4], GNU Savannah[5], Google Code[6], JavaForge[7], GitHub[8], and others. These sites offer researchers a collection of software projects cataloged in a similar way, and many research papers have used one or more of these sites to generate samples for analysis[3, 4, 5, 6, 7, 8]. However, using these sites as a data source has some drawbacks.

For one, a project is likely to be hosted on only one of these sites, making it difficult to identify all of the software projects in a given domain. In addition, not all projects make their home on one of these hosts. This is particularly true of large projects with commercial support, which tend to have their own dedicated homepage. Also, not all projects are equal; just because a project is hosted on a forge does not mean that it has an active development team, user base, or runs reliably. Extracting data from these sites can also be a challenge. While all of them let you examine projects individually, most do not provide any APIs or expose their underlying databases to the public. This makes it difficult to extract data about a large number of projects concurrently.

Software repositories maintained by Linux distributions may provide researchers with an alternative data source. The Linux kernel by itself is just a part of an operating system. A Linux distribution bundles the kernel along with the other utilities needed to create a complete system. Most distributions include a software management system that simplifies installing and removing software from a running instance of their system. This additional software is typically stored in freely accessible repositories that contain the components, bundled into packages, that will run on a computer using the Linux distribution. These packages are pre-compiled programs from open source projects. Repositories make it much easier to use open source software by providing ready-to-run executables and saving the user from having to compile a program from source. They greatly simplify the process of adding and removing programs.

This is primarily accomplished through dependency management. Package y is a dependency of package x if package y must be installed on a system in order for package x to function properly. Package y may have its own dependencies which will also have to be installed for package x to function. Dependencies are said to be resolved when the packages that provide them have been identified. Repositories simplify the task of resolving dependencies by tracking the installation requirements of all their member packages. The task of dependency management is carried out by tools developed to interact with the given repository structure. These tools allow users to search for and install packages

[2] http://sourceforge.net
[3] http://sourceforge.net/apps/trac/sourceforge/wiki/What%20is%20SourceForge.net?
[4] http://freecode.com/
[5] http://savannah.gnu.org/
[6] http://code.google.com/
[7] http://www.javaforge.com
[8] https://github.com/

stored in the repository. When asked to install a package, they automatically identify and install all of the required dependencies. The entire process makes it fairly trivial to install the most popular open source products. The tools can also be used to query the repository for information about its packages.

Repositories provide an additional, yet equally important function: they can be used to enforce a certain level of quality control over the system. For example, it is common practice for Linux distributions to maintain separate repositories for stable and development releases. This allows the user to easily choose between software that has been proven to run reliably and bleeding edge releases that have not been thoroughly tested. In addition, each repository owner typically places restrictions on which packages would be eligible for inclusion. The Debian[9] distribution, for instance, is committed to promoting free software, and therefore limit their repositories to include only packages that are released under a license that meets the open source definitions provided by the Free Software Foundation and the Open Source Initiative. Other distributions have less strict entrance requirements.

Linux package repositories provide a central location for thousands of projects and can be a great resource for researchers interested in studying open source development methods. To date, few open source research projects have taken advantage of them.

1.3 Problem Statement

The goal of this study is to explore the effects of license choice on open source software reuse. Is software released under a particular open source license more likely to be reused than software released under a different license? To what extent does license choice affect the amount of reuse that occurs within a software development project?

In attempting to answer these questions we use a definition and measurement of reuse that reflects the way software is reused in practice. Rather than search for code that is common between two or more projects, we identify the number of preexisting packages used to create a given application, as well as the number of times each package is reused by another project. It is hoped that the methods employed to analyze reuse of open source software used by this study will be used by practicing programmers to evaluate a software package's suitability for reuse within a project.

A software repository maintained by a Linux distribution is used as the primary data source for the study. As will be shown, Linux repositories provide a sample of open source software from multiple domains and license types. In short, they provide a good representation of the open source software that is in common use, robust, and actively supported.

[9] http://www.debian.org/

In the end, the results of the study may help to provide empirical evidence of a measurable difference between types of open source licenses in general, and inform the decision of which license type to use for a new software project in particular.

Chapter 2

Literature Review

2.1 Open Source Licenses

2.1.1 A Brief History of Open Source Software

The open source movement has a colorful history that can be traced back to the early days of computing. Many books, papers, and websites document this history much more thoroughly than will be done here. Nevertheless, a brief summary will help provide the context for the current state of open source research.

Before personal computers became affordable to the general public, the only way most programmers could access a computer was by using a mainframe system provided by their employer or university. Since these systems could only support a limited number of users at any given time, the first challenge lay in getting access to the machine. Computer systems were often custom built, and ran operating systems written in assembly language that ran on specific hardware. Software was custom built as well. The proprietary software industry was in its infancy and many applications were developed in house. During this time access to source code was common. System developers needed access to the source code in order to troubleshoot problems or update programs. Numerous sources[9, 10, 11] describe an environment where the source code to programming problem solutions were freely shared and redistributed.

Perhaps the best example of this is the development of the Unix operating system. Unix was created by a team of programmers working for Bell Labs during the early 1970s. It was written in the C programming language, which meant that it could be run on a variety of hardware platforms. Anti-trust laws prevented AT&T, which owned Bell Labs, from selling Unix as a commercial product. Instead, it was made available to universities and commercial entities under licensing terms that granted access to the source code. The proliferation of Unix among universities coincided with the development of the

ARPANET, which in turn helped facilitate the evolution of a community of programmers that would eventually create the Internet. The ability to share source code had a direct positive impact on the growth of this community and was a crucial part to its success.

As computers became more accessible, and it became possible for an individual to purchase one for their own use, access to source code began to become less common. The proprietary software industry, which began to grow as the number of personal computers increased, did not distribute source code along with their products in order to prevent unauthorized copying. This frustrated programmers who saw accessing and sharing source code as a fundamental part of software development. One of these programmers was Richard Stallman, a programmer who worked for the Massachusetts Institute of Technology Artificial Intelligence Lab during the late 1970s. In a widely repeated story, which may be apocryphal, Stallman, who was frustrated at the performance of a Xerox printer that had been donated to the lab, attempted to get access to the source code so he could modify the driver and improve its efficiency. He was refused on grounds that the printer driver was a proprietary product not intended for those outside of the company to modify. Stallman identifies this event as a turning point that ultimately led to his leaving the MIT AI Lab and forming the GNU Project in 1983[11].

The GNU[1] Project laid the foundation of today's open source movement. Its manifesto[12] establishes the ideological foundation of what it means for a program to be open source. In the manifesto, Stallman outlined his intention to build a Unix-compatible operating system and associated development tools that would be freely available to anybody who wished to use it. He also outlined his definitions of "free" by describing four freedoms: the freedom to run a program for any reason, the freedom to make modifications to that software, the freedom to distribute copies, and the freedom to distribute modifications made to the software. Stallman's ideas stem from a strong sense of obligation to his fellow programmers. He feels that restrictions to these freedoms, which are common in most commercial software licenses, are fundamentally unjust, and he refuses to use any program or computer system that he sees as restricting his freedom or the freedom of others. Not all open source supporters view the accessibility of code in these terms. Nevertheless, these principles were important in establishing the GNU Public License (GPL), the first software license that encouraged the distribution of source code. By the early 1990s, the GNU Project had produced many tools essential to the success of an operating system, including a compiler (gcc), text editor (emacs), and other software development utilities. It did not have a working kernel, and therefore did not provide a complete system.

In 1992 a computer science student at the University of Helsinki named Linus Torvalds, began working on a project to create a Unix-like kernel that would run on the i386 processor commonly found in PCs at the time. This kernel, named Linux, was developed by a group of volunteers spread across the world that communicated over the Internet.

[1]GNU is a recursive acronym that stands for "GNU's not Unix"; see http://www.gnu.org for more information.

Linux was released under the GPL, which meant that anybody interested in contributing to the project was able to do so. Within a few years Linux, combined with the utilities developed by the GNU Project, produced a stable Unix-compatible system that realized Stallman's original vision.

The success of Linux inspired a programmer and writer named Eric Raymond to write *The Cathedral and the Bazaar*[13], an influential paper that describes software development practices that allowed Linux to succeed. Raymond was very skeptical that a loose-knit group of developers working independently could produce a piece of software as complex as an operating system and have it be anything more than a toy with limited usability. His amazement when he found this not to be the case led him to investigate the software development methods used by the kernel team. In his paper he describes two contrasting methods of software development. The cathedral method, typical of most proprietary software projects, consists of a small group of developers working under tight leadership, that does not release software until it has been rigorously tested. In contrast, the bazaar method consists of a large number of developers, working individually under little or no supervision, frequently releasing lightly tested code. The Linux kernel was developed using the bazaar method. While trying to figure out how a project using the bazaar method could be successful, Raymond made a bunch of observations that underscored the practical advantages of allowing unrestricted access to source code.

Perhaps the most important observation was what Raymond dubbed Linus's law[13]: given enough eyeballs, all bugs are shallow. Increasing the number of users who have access to a product under development increases the number of bugs that can be discovered. By allowing any interested developer access to the source code, the chances that the fix for these bugs will be understood and implemented also increases. Releasing changes early and often can actually increase the quality of the product. None of this is possible if source code cannot be accessed and redistributed freely. Raymond's work stressed the strength of open source software licenses in creating robust and reliable products.

By the end of the 1990s and into the new millennium the open source movement created products capable of challenging their proprietary counterparts. Linux began to play an increasing role in the server market, with the Apache HTTP server becoming the most widely used web server.[2] The success of companies such as Red Hat have demonstrated that open source products can be commercially successful. Since their code is accessible and their development methods are transparent, open source products provide an excellent source for researchers. Numerous studies have been performed on a wide range of topics related to open source software, some of which are discussed further in this paper.

[2]See http://news.netcraft.com/archives/2011/01/12/january-2011-web-server-survey-4.html for more information.

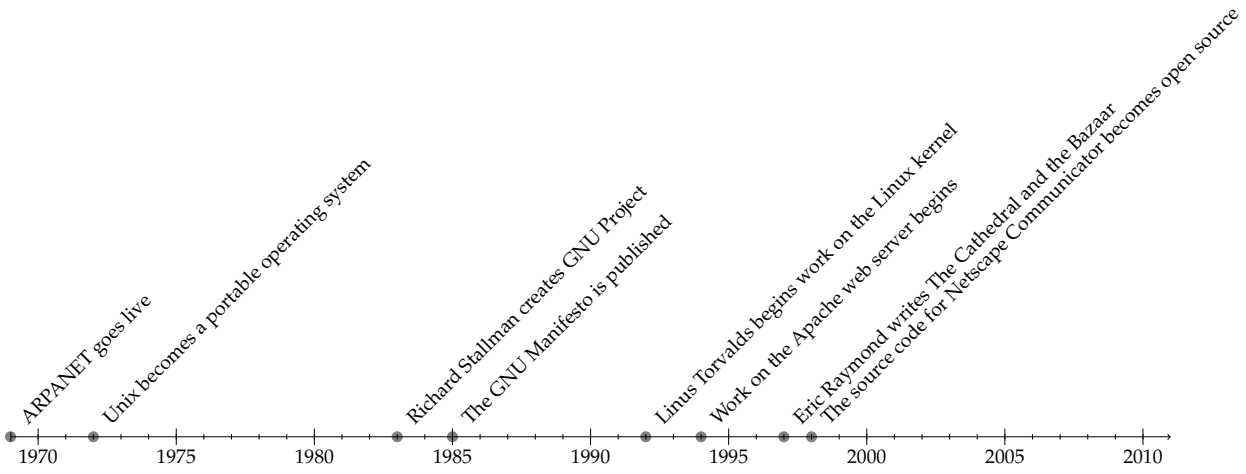

Figure 2.1: Some important dates in the history of open source software

2.1.2 Open Source License Types

In general, software can be considered open source if its source code has been made available to its users. Specifically, the license under which the software is released specifies the rights of licensees to access, modify, and redistribute its source code. The license distinguishes open source software from non-open source, or proprietary, software[6].

Computer software, like art, literature, music, and other published works, is copyright protected. The owner of a copyright retains the right to determine how their work is produced, distributed, and modified. Copyright law evolved with the evolution of the printing press, and was aimed at protecting a creator's right to profit from their work and to prevent the author's exploitation by others. These laws also apply to computer software. Proprietary software is usually produced by a team of developers working for a commercial entity. This entity retains the copyright of the produced software. The developers produce the software as a "work for hire", which means that in exchange for their salary and other economic benefits they void any claim of ownership on the code they produce. The commercial entity distributes the product for a price, and keeps any resulting profit[14].

Software licenses are intended to protect the copyright owner's exclusive rights to a software product by specifying the terms of use that a consumer must follow in order to use the product. Proprietary licenses vary but tend to have some significant points in common. Most importantly, they grant a user a license to use, and not own, the product. As a licensee, the consumer does not own the software; rather, they have purchased the right to use the software under the terms of the license. Typically these terms limit the number of systems that the software can be installed on, or the number of users who can use the software, under a given instance of the license. Proprietary licenses also prevent the user from making copies of the software and redistributing it to others. The source code of a

proprietary software product is not provided to the user because doing so would make it much easier for users to violate the terms of the license and reduce the effect of the entity's rights over their product.[15]

The open source software movement was born in part as a reaction to this model. Supporters of the movement view the enforcement of the rights of one entity at the expense of others as the fundamental problem of the proprietary software model. As a result, a number of open source licenses have been created that use the rights afforded by copyright protection to prevent one party from controlling the use and distribution of software. Rather than using license terms to restrict what a user can do with the software, an open source license grants a user the rights to make changes to the original software and redistribute these changes. A programmer who releases a program under an open source license does not give up copyright to their work. Instead, they use their rights as a copyright holder to grant modification and redistribution rights to others.

The open source movement consists of a collection of individual programmers and software projects, and the term "open source" can mean different things to different people. Two organizations, The Free Software Foundation (FSF)[3] and the Open Source Initiative (OSI)[4], propose formal definitions of what it means for a software product to be open source. These definitions are largely accepted by the open source community as a whole. Each organization also provides a list of those software licenses that comply with their definition. The FSF advocates "free" software, by which they mean the freedom of a user to access source code, modify it, and redistribute it without limiting the rights of others to do the same. The FSF sees free software as a solution to a social problem[12]. Their philosophy takes a political stance by emphasizing individual rights over corporate interests. The OSI, on the other hand, views open source development methods as a better approach to software engineering. They contend that the ability to use existing code in a new project produces higher quality code. They frame their open source advocacy in economic terms, and say that adopting an open source license will lead to greater innovation and market success than the use of a proprietary license[16].

Currently[5] the FSF list includes 86 different licenses, and the OSI list includes 69. However, despite the differences in philosophy between the two organizations, their lists mostly overlap. In addition, the vast majority of open source projects are released under a small subset of these licenses. All of these licenses have some important features in common. First, they grant access to a program's source code and allow the user to make modifications. Second, they allow a user to distribute copies of the program, in original or modified form. Open source licenses differ in the restrictions (or lack thereof) placed on how modifications can be redistributed. What follows is a brief overview of the most commonly used open source licenses.

[3] http://www.fsf.org
[4] http://www.opensource.org
[5] The license count was taken on November 27, 2011.

- *GNU General Public License*[17]- The GNU General Public License (GPL) is the most widely used open source license[8]. The original version of this license was written by Richard Stallman, and is intended to promote the ideas of freedom that led to the creation of the GNU project. It has evolved to become the standard license for software released by the FSF, including the GNU C Compiler, the Emacs text editor, and many others. The GPL allows a licensee to modify a program and distribute the changes, as well as to redistribute the original program. It also allows the licensee to charge a fee for their efforts provided that the distributed code is released under the same licensing terms (namely, the GPL) as the original and that any changes made to the original are identified. This is to ensure that the software remains freely accessible to anyone who wants to use it or modify it for their own purposes. Software released under the GPL cannot be turned into a proprietary product. The GPL is considered to be the most restrictive license, and an example of a copyleft license, due to this provision.

- *GNU Lesser General Public License*[18]- The GNU Lesser General Public License (LGPL) is a variant of the GPL with relaxed some of the redistribution restrictions. Code released under the GPL cannot be used as part of a program that is governed by another license. This can be a problem for code, such as a software library, that is intended to be integrated as part of a larger program. The LGPL was introduced to solve this problem. Software released under the LGPL is subject to the same terms as the GPL except for those that prevent the software from being included as part of a larger work covered by a different license. The software covered by the LGPL, and any modifications made to it, must be distributed under the LGPL. This allows an entity to produce a proprietary software product that uses libraries released under the LGPL to remain closed source, provided that the libraries, and any modifications to them, remain open source.

- *Apache Software License[19]*- The Apache Software License (ASL) was created for use by the Apache Software Foundation (ASF)[6], an organization that supports a variety of open source software projects. The ASL grants a licensee the right to distribute copies of a program, with or without modification, provided that any modifications are identified within the files that they occur and that any attribution notices from the original work remain. This allows recipients of the modified or redistributed code to be able to identify the contributors that have made changes to the code throughout the life of the product. As long as these terms are met, the ASL does not require that the modifications be released under the ASL. Section 4.4 of the licenses explicitly grants the licensee the ability to provide additional terms. The ASL does not give the licensee the right to use the trademarks or product names of the licensor. This means that a modified version of the software produced by a third party cannot be advertised as a product of the original author without the author's permission.

[6]http://www.apache.org

The ASL does not require that source code be included as part of a distribution, which means that software released under the ASL can be included as part of a proprietary product.

- *BSD License*[20]- Another permissive license is the Berkeley Software Distribution (BSD) license, named after the Unix-like operating system developed by the University of California, Berkeley that first used it. The current version of the license grants licensees the right to redistribute a program in source or binary form, with or without modification, provided that the existing copyright notice is retained. As with the ASL, code released under the BSD license can become part of a proprietary software package whose source is not released.

- *MIT License*[21]- The MIT license allows a recipient to copy, use, modify, and redistribute source code provided that the copyright/permission notice be included with any copies of the software. It is the most permissive open source license, as it places the least amount of restrictions on how modifications can be distributed. As with the ASL and BSD licenses, software released under the MIT license can be distributed under new licensing terms that limit access to source code.

2.1.3 Open Source License Categories

In the literature open source licenses tend to be categorized according to the restrictions they place on how changes to a program can be redistributed. Although researchers categorize licenses differently, most consider a license to fall into one of two categories, copyleft and permissive. Copyleft licenses require that any derivative works be released under the same licenses as the original work. Copyrights are typically used to ensure that the rights to a work remain in the control of one person or entity. Copyleft licenses use copyright protections to do the opposite. They are aimed at making sure that the work always remains freely available to everyone. Anyone is free to distribute derivative works provided that anyone else is free to do the same with the new work. In the software world, these licenses are specifically designed to prevent an entity from co-opting a program and releasing it under a proprietary license.

Permissive licenses do not require that derivative works be released under the same license. Software released under a permissive license can be changed and distributed under a different license, including a proprietary one, provided that the other license terms are met. In most cases the other terms deal with how to credit contributors and how trademark or patent rights are enforced under the license.

The copyleft and permissive categories neatly sum up the different opinions of what it means to be open source. Those who view open source in terms of individual freedom, such as the FSF, use copyleft licenses to ensure that nobody else's freedom to use or modify a program is taken by somebody else. Others see permissive licenses as the best way

Categorizing Open Source Licenses					
Researchers	GPL	LGPL	ASL	BSD	MIT
Lerner & Tirole[6]	Highly Restrictive	Restrictive	Unrestrictive	Unrestrictive	Unrestrictive
Stewart, Ammeter, & Maruping[10]	Restrictive	Unrestrictive	Unrestrictive	Unrestrictive	Unrestrictive
Colazo & Fang[3]	Copyleft	Copyleft	Noncopyleft	Noncopyleft	Noncopyleft
Lindman, Paajanen, & Rossi[1]	Restrictive	Restrictive	Permissive	Permissive	Permissive
Sen, Subramaniam, & Nelson[9]	Strong Copyleft	Weak Copyleft	NonCopyleft	NonCopyleft	NonCopyleft

Table 2.1: How different researchers categorize open source licenses

to increase commercial interest in open source software. They allow a university or research institution to produce working code as a proof of concept that can be adopted by the entire software industry, including entities that produce proprietary software.

Some researchers[22, 23] propose a third category, sometimes called "weak copyleft"[23], to include copyleft licenses that relax the restrictions of how a derivative work may be packaged with other software. A strongly copyleft license such as the GPL prevents modifications from being packaged with software covered under a different license. For example, if a programmer modifies a piece of software released under the GPL, they cannot release the unmodified code under the GPL and the new additions under a different license. The GPL requires that the entire derivative work be released under the GPL[15]. This is to ensure that anyone may have the same privileges to the modified software as the modifier had to the original[17]. A weak copyleft license, such as the LGPL, eases this restriction by allowing a protected work to be bundled with software covered under a different license provided that any changes to the original remain covered by the same license. A programmer can make changes to a library released under the LGPL and include their modifications as part of a larger work covered under a non-LGPL license provided that the library continues to be distributed according to the LGPL[15].

Not all studies of open source development consider different license categories, or even consider the differences between open source licenses at all. Others include clearly defined categories and which specific licenses belong to them. The creation of a license category and assignment of a particular license to that category varies according to the motives of the researcher and the software development characteristic being investigated. Table 2.1 shows how the studies included in this review that consider license categories to be important would treat the individual license types discussed earlier.

2.2 License Choice Matters

At first glance, the differences between open source license types may not appear to be significant. However, many studies have shown that the open source license chosen by

a software project can have a significant effect on the project's success. What follows is a summary of these studies. Each considers varying measures of success, and then examines how the license choice affects these metrics. These studies look at the choice of license from different perspectives, but they all agree that the choice of license is important.

2.2.1 License Choice Matters to Programmers

Open source developers tend to have strong feelings about open source licenses. These feelings vary according to the philosophical ideas expressed in the license, its economic potential, and the individual beliefs of the programmer. Early research of the open source movement focused on whether open source projects could produce software that could compete with proprietary software, and whether or not it is possible to make money by allowing redistribution of source code. As a result, many studies treat a programmer's license preference as a function of market factors; that is, a developer is likely to prefer the license that is most likely to lead to future economic reward.

In *The Scope of Open Source Licensing*, [8] Lerner and Tirole investigate an observation made in their previous work[24] that the proliferation of new open source licenses following the creation of OSI's open source definition suggests that license choice is important. Their follow-up study examines what factors determine the choice of license. Their research identifies peer recognition, possible future career opportunities, the joy of working on a challenging programming problem, and the ability to tailor code for a specific purpose as the main benefits received by a programmer that chooses to contribute to an open source project. Their model quantifies the noncommercial benefits of an open source project as the sum of three factors, one of which being the restrictiveness of the project's license. In this model, the noncommercial benefits of a project increase according to the restrictiveness of the license. This is intended to account for the increased possibility that a project under a less restrictive license will be hijacked by a commercial entity producing a closed source product. Such hijacking is seen as undesirable because the contributions made by programmer when the project was still open may become harder to identify when the source is closed. This reduces the potential of those contributions to increase peer recognition and to signal skills to future employers. It also prevents developers, along with the user community at large, from benefiting from the coding solutions introduced in subsequent versions. A highly restrictive license such as the GPL will guarantee the contributions are always visible, and all future modifications are accessible by everyone.

Mikko Mustonen also focuses on economic incentives for programmers in the model proposed in *Copyleft- the economics of Linux and Other Open Source Software*[14], which describes the conditions under which proprietary software can coexist with an open source equivalent. This paper assumes a conflict between the incentive to create new information (i.e., a piece of software) and the public's right to exchange this information freely. A programmer can choose to work for a proprietary software firm, or contribute to an

open source project. If the programmer chooses the first option, they will receive a fixed wage, but if they choose the second, they will have to rely on complementary income as they will not be able to receive direct compensation for the code they produce. Instead, the open source programmer will have the chance to develop a reputation that will lead to future financial returns, such as receiving investments for a future business or securing academic employment. Mustonen posits that a programmer will choose the option that provides the greatest net benefit, and proposes a model of how this choice is made. The model uses a programmer's productivity to identify which path to take. A commercial firm is likely to pay their programmers a similar wage regardless of individual productivity. This benefits a programmer that is not very productive, but not those with a high productivity. In an open source project, the benefit to the programmer is the expected complementary income times the programmer's productivity. There is productivity threshold above which a programmer can receive greater benefit by contributing to an open source project instead of working for a commercial firm. Note that this model does not take into account the non-financial benefits of contributing to an open source project. These will be discussed in subsequent paragraphs.

The studies that treat reputation benefits and peer recognition as economic indicators run the risk of over simplifying a significant explanation of the motivation of the open source programmer. In his essay "Homesteading the Noosphere"[2], Eric Raymond provides a sociological explanation for the open source movement. In this essay, as well as several others he has written about software engineering, he describes the open source movement in cultural terms; that is, he sees open source programmers, called hackers[7], as a collective group that share similar values, behaviors, and traditions, and whose motives cannot be purely attributed to the same factors that govern decisions made by closed-source entities. Raymond makes the point that while it is possible for a hacker to secure a job offer, teaching position, or book deal from his open source programming exploits, these events do not occur frequently enough for such economic considerations to be a primary, let alone the only, motivation for a programmer to give away their source code. While there is variation in the motives and behaviors of individuals within the community at large, he identifies the traits that describe the culture as an example of a *gift culture*, a group consisting of members who give away goods and services (in this case, software) without receiving obvious compensation in return. Such cultures, he argues, can only flourish in environments where necessities, such as food, are abundant. In the modern world of software development, disk space, network bandwidth, and computing power are the plentiful necessities. In this culture, social status is tied to what you choose to give away.

A hacker's social standing with the group is tied to his or her reputation as a programmer. Peer recognition has evolved as the best way to measure the value of software contribu-

[7]The term "hacker" is overloaded and often misused. My usage of the term is the same as Raymond's: an enthusiastic programmer who enjoys writing code for its own sake, and most definitely not someone who breaks into computer systems illegally.

tions to open source projects. Software is complex, and it is difficult to objectively measure the quality of code submitted to a project. As a result, the open source world places a high value on hackers that can not only code well, but are able to judge the contributions of others fairly. This "judging" usually takes the form of a person with a position of authority within a project deciding to accept a hacker's contribution into the project's next release. Typically one must have established a reputation of objectivity as well as programming excellence in order to get such a position, and the only way to do this is to receive recognition from your peers. Reputation is everything.

When determining the attractiveness of one license choice over another, Colazo and Fang[7] took into account elements of social movement theory to explain developer choices. Social movement theory, as it relates to the open source movement, says that an individual will consider how personal ideals correspond with those of a project in addition to whether the benefit of participating is worth the economic cost. Software licenses fulfill different needs; they can be seen as technical, commercial, juridical, and political tools.[22] Studies that focus on the first three aspects risk minimizing the impact of the fourth. The open source community is defined by the conviction that source code should be freely redistributable, but there is much debate within the community over which redistribution rights are ideal and how they should be enforced. Licenses define the specific rights granted to users of a project, making the choice of license a way for a project's leader and developer community to identify the values they associate with the open source development as a whole. In their work, Colazo and Fang suggest that by choosing to work on a project with a particular license, a developer identifying himself as a member of a collective group, and that group affiliation plays a big part in the choice of which projects to contribute to. This lead them to the conclusion that open source developers should be more attracted to copyleft licenses.

Raymond[2] supplies evidence of the values shared by open source programmers with an interesting observation about the difference between the text of most open source licenses and the way in which code is written. All open source licenses give a user the right to make changes and redistribute them in some fashion. This implies that these licenses encourage developers to circulate their own customized versions of a program. In practice, this is rare. Open source programmers have determined that it is much more valuable to the group as whole to submit changes to the original project instead of creating their own fork of the project. Justifications for this range from the explanation that forking a project results in duplicate effort, to splitting a project makes it difficult to give the right people credit. In either case, this behavior lends strength to the idea that the manner in which a programmer contributes to a project is indicative of their values and ideas of software development.[2]

Determinants of the Choice of Open Source License by Sen, Subramaniam, and Nelson[23] examines the how a programmer's motivation and attitude towards the open source movement affects their license preference. This study puts individual motivations into two categories: intrinsic and extrinsic. Intrinsic motivations are those that appeal to a

developer's interest in programming for its own sake, rather than in response to an external pressure. This includes the challenge of solving a programming problem along with the pleasure of designing that solution. Extrinsic motivations are those that result from environment in which a developer works, and includes peer recognition and financial compensation. These motivations are a sign of how a developer expects their project to be accepted by the open source community as a whole, and is reflected in their choice of license.

Sen, Subramaniam, and Nelson conducted an online survey to determine how these motives affect license choice. They divided open source licenses into three categories, strong copyleft (e.g., the GPL), weak copyleft (.e.g., the LGPL), and non-copyleft (e.g., the BSD license). The survey sample consisted of open source developers that contribute to projects hosted on Sourceforge. A request to participate in the survey was sent 2,000 open source developers of which 196 responded. An analysis of the survey results drew several conclusions. The survey results showed that the average score on questions pertaining to intrinsic motivations was higher than those for extrinsic motivations. Developers that enjoy the challenge of programming are 72% more likely to choose a weak copyleft license over a non-copyleft one. Developers who are motivated by the extrinsic desire to increase their status among their peers are more likely to choose the non-copyleft license. As an explanation, the authors of this study suggest that this is because software released under a non-copyleft license has fewer restrictions placed on its use, and will therefore attract more users than software under a weak or strong copyleft license.

2.2.2 License Choice Matters to Projects

If the open source license type is important to individual programmers, it is also important to the projects that they contribute to. Several studies, including the ones already discussed, have established by showing that a project's choice of license type has an effect on the project's success.

Lerner and Tirole's license determination model[8] describes a project's license choice as the sum of the project's attractiveness to potential users, the restrictiveness of the license, and an economic factor. This economic factor is defined as the possible commercial benefit received by the project and the weight the project leadership places on receiving that benefit. A project will choose to go open source if there is at least one license type for which this sum exceeds the profit that could be realized if the project went proprietary. The model also shows that the developer community's license preference plays a role in project success as well. A developer will choose to contribute if the benefit they expect to receive exceeds the opportunity cost of contributing; that is, they will contribute if they stand to gain more than they would by working on a closed source alternative. This benefit increases with the restrictiveness of the license. The project stands to increase its chance of success by attracting as many contributors as it can, and choosing the right

license type can make the project more appealing to programmers.

To test the predictions made from their model, Lerner and Tirole conducted a study of the approximately 39,000 projects hosted on Sourceforge in May of 2002. They extracted a staggering amount of data about these projects, including license type, target environment, the human language the developers use to communicate, operating system, and the intended audience. The summary statistics of this data set showed 72% of the projects were released under the GPL and LGPL, which means that highly restrictive license are used much more often than their non-restrictive counterparts. The analysis of this data led to many conclusions about which factors affect license choice:

- Highly restrictive licenses are much more common in applications created for end-users than they are for software written for other developers. One possible explanation for this is that contributors to projects that produce software development tools want to increase the potential user base for their product, and releasing under a non-restrictive license makes their tools more attractive to commercial projects that do not want to make their source code accessible.
- Consumer applications, such as desktop tools and games, programs where the developers speak a language other than English, programs written for open source POSIX operating systems (Linux, FreeBSD, etc), and newer projects are all much more likely to be released under a highly restrictive license.

An analysis performed on their models for developer, community, and project license preferences predicted that participation in a project increases as the restrictiveness of the license decreases. Their study tested this prediction by looking at projects with similar levels of success, where success is measured by the number of developers working on the project and the frequency with which they contribute. Projects released under a license that is not highly restrictive showed significantly greater contributions, and therefore supported the model's prediction.

In order for an open source project to be successful, it must be able to attract volunteer contributors. Programmers decide to contribute to a project if they determine that the benefits received for contributing greater than the cost of not contributing. The challenge for a project leader is to maximize the number of developers that will determine that it is beneficial to them to contribute to the project. A number of studies have demonstrated that license choice can play a role in increasing a project's appeal to potential developers.

Colazo and Fang[7] observed that many open source projects fail due to lack of developer participation, and not necessarily that there is no demand for the product. Their study seeks to determine what effect the choice of license has on developer activity within a project. To do this, they identified actions taken within the project that could be used to indicate developer activity. This includes the number of developers working on a project, the amount of code generated by these developers, the speed at which the project pro-

gresses, and the rate at which developers leave the project. Like many other open source research projects, they used Sourceforge as a data source, which provided resources that were used to establish metrics for each of the actions listed above. For instance, it is possible to see which users commit code changes into the project's code repository and how often they do it, which was used to determine the number of participants and the level at which they contribute. If a certain amount of time had passed since a developer last committed a code change to the project, that developer was considered to have dropped out of the project. They measured project speed as the rate at which new versions of the software in their sample were released, under the assumption that successful projects operate under the principle of "release early, release often" discussed in[13].

The sample of open source projects extracted from Sourceforge was purposely biased to include only those projects that showed development activity after the first posting. This eliminated projects that were announced on the site but never developed, a common occurrence on many of the free open source project hosting sites. The authors estimate that this reduced the sample population by about 80%. Candidate projects for the sample were chosen from the remaining 20% if they were collaboratively developed, had publicly available project data (so that developer activity could be measured), and had produced software that had been ported to at least one other platform or computing environment. The last criterion was used to find projects that are relevant to a large user community, and being ported to a different system is evidence that such a community exists. The importance of the size of the user community is an extension of the assumption that many developers are motivated to contribute to open source projects in order to increase their reputation as contributors to useful projects.

The results of the study showed that projects released under a copyleft license ranked higher than non-copyleft programs in the number of developers on the project, developer activity, and development speed. Non-copyleft projects performed better on developer permanence, as their developer turnover rates were significantly lower than those of copyleft projects. The general conclusion that Colazo and Fang drew from these results is that there is a relationship between license choice and development activity in open source projects. Also, they concluded that activity in copyleft projects shows that self identification with open source development as a social movement is an important motive among developers, and is just as important as economic motives.

Mustonen's model[14] addresses project participation from the perspective of a commercial entities seeking to attract developers away from open source projects. Their principle method of accomplishing this is through the wages they offer to their employers. In order to participate in the market at all, the productivity of the best programmers and the wages paid to them must exceed a certain threshold. This threshold defines the point at which a productive programmer will choose to contribute to an open source project. If the wage offered by the commercial entity is high enough, the programmer will choose to work for them instead. The model compares proprietary and copyleft projects, and does not distinguish between open source license types.

2.2.3 License Choice Affects Commercial Decisions

Lerner and Tirole, in a widely cited paper from 2002[24], provide a basic overview of the economic considerations of open source software. Inspired by an analysis of four successful open source projects (Apache, Linux, Perl, and Sendmail), they examine two basic questions: what economic benefit does a person receive by contributing to an open source project, and what economic opportunities does the open source movement provide to commercial software companies?

How can a proprietary software vendor embrace open source principles and still remain profitable? Lerner and Tirole suggest a number of different ways. The first is by providing support services to an open source project. Red Hat is an example of this: their core product, Red Hat Enterprise Linux (RHEL), consists of a bundle of programs that can be obtained for free from numerous open source projects. RHEL itself is free, but Red Hat sells technical support and a system update service that greatly reduces the effort needed to set up and maintain RHEL systems. They have also taken the lead in providing certifications in various open source technologies and training courses to assist IT professionals in earning them.[8] Lerner and Tirole call this a reactive strategy, as it relies on software created by others.

Another approach is for commercial entity to produce their open source software. This strategy makes sense when releasing source code would result in a higher return of investment than what could be achieved by keeping the source closed. For example, by releasing open source drivers for their product, a hardware vendor might see an increase in sales as the driver is ported to run on different operating systems than they would if they provided a closed source driver that only ran on Windows.

A variation on this approach is for a commercial entity to partner with and/or make significant contributions to an existing open source project. Lerner and Tirole only hint at this possibility, perhaps because the practice was less common in 2002 than it is today. If a commercial entity has a need for some software, they might fulfill this need more effectively if instead of paying for a proprietary solution from another commercial entity they hire developers to adapt an open source alternative for their purposes. An example of this approach is the Eclipse, an open source programming environment[9] that began its life as part of an IBM product. IBM released Eclipse under an open source license, which has helped turned the software into one of the leading Java development environments. In exchange, IBM is able to reap the benefits of using a product that has a much larger developer base than might have been possible had they kept Eclipse as a proprietary tool.

Choosing an Open Source Software License in Commercial Context: A Managerial Perspective[22] examined license choices made by companies that produce open source software. It describes a preliminary model of open source license decision making based

[8] For more information on Red Hat, visit their website at http://www.redhat.com
[9] http://www.eclipse.org

on case studies of eight Finnish software companies. The case studies were not statistical, and consisted of loosely structured interviews to identify the factors that led to a license choice. After asking a set of predetermined questions, the researchers asked for additional information and may have discussed new topics that came up during the discussion. These conversations provided useful insights into license choice as a commercial decision. Licenses were grouped into two categories, restrictive (GPL, LGPL, MPL) and permissive (MIT, BSD, Apache). The case studies showed that a company's business model was the biggest influence on license choice. Smaller companies were more likely to create a business model, then choose the license that best supports this model. Larger companies had greater flexibility in experimenting with different alternatives. A company's background and views on the community also play a role in license choice. Companies that began as open source projects more likely to consider outside developers when making their license choice, and those with proprietary backgrounds did not consider community input. All of the companies interviewed placed high importance on keeping in control over the development process and retaining all of the results.

Lindman et al. used these interviews to propose a framework to describe license choice. The framework looks at the business model, software patents held by the company, community, leadership, external issues, and company size as key factors in the license decision.

The paper also notes a fundamental difference between open source and proprietary licenses other than the distribution of source code. In a traditional proprietary software development environment, a company's relationship with its programmers is defined by formal employment terms or contracts. The employees are obligated to produce code for the project as part of their employment. The software license used by the product provides terms of use to end-users. If the company releases source code under an open source license, they may attract contributions to developers that are not employees, and cannot be compelled to submit code. Choosing a license poses trade-offs to a commercial firm that go beyond the typical concerns of protecting intellectual property rights. Lindman et al. found that this change in relationship between a company and those who contribute code to their project played an important part in the decision of whether or not to release their code under an open source license.

Figure 2.2 contains a fishbone diagram of summarizing considerations that affect the choice of on open source license.

2.3 Methods of Studying Open Source

Most open source projects rely on contributions from volunteers, and are accessible over the web. Interested parties can often gain access to source code, bug reports, archives of email communication between developers, and other information providing details

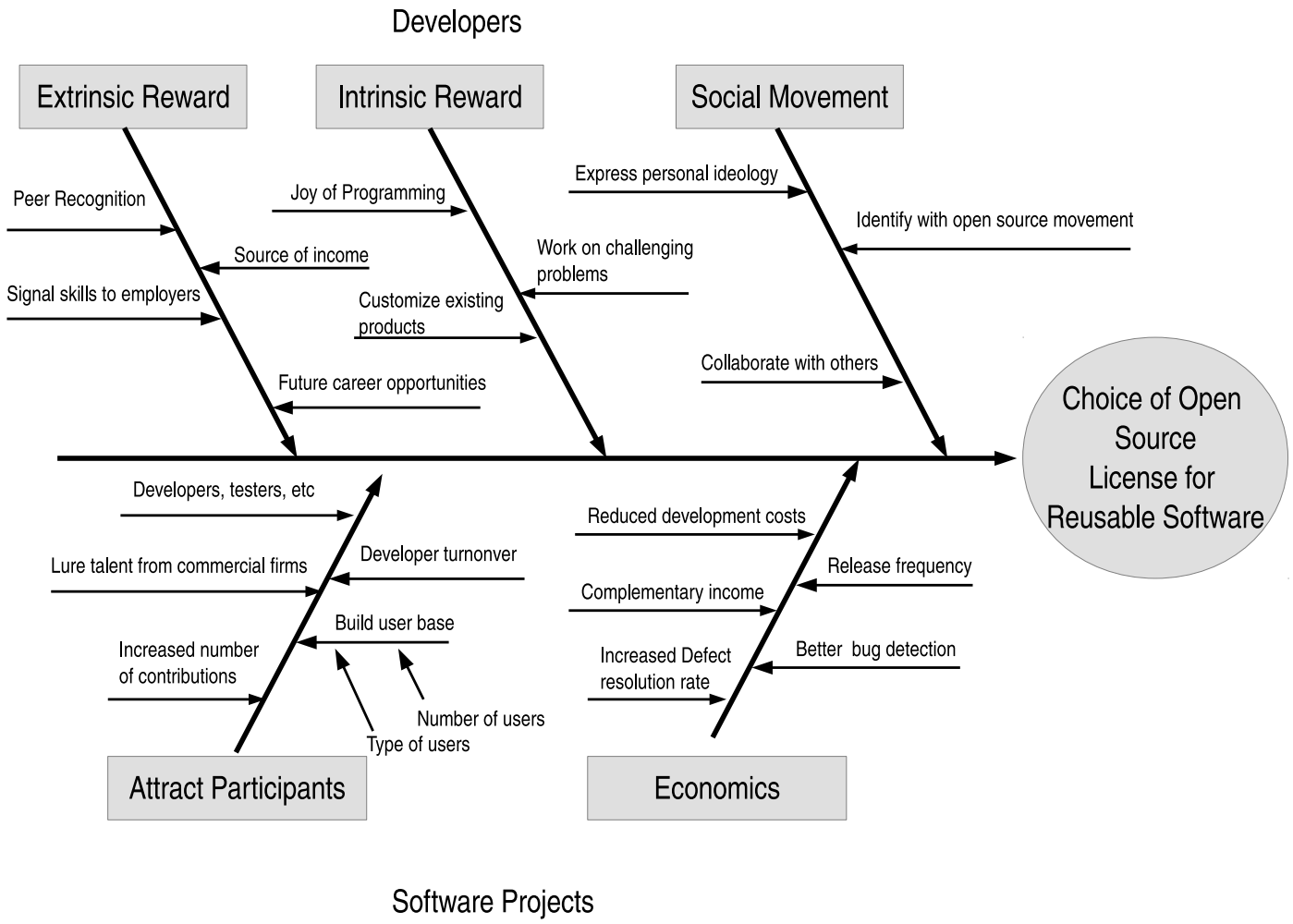

Figure 2.2: Fishbone diagram of factors that influence the adoption of an open source license

of the project's development process. The amount of freely accessible information on thousands of open source projects provides researchers with a wealth of data on software engineering practices. This section summarizes the data collection methods used by the papers examined in this review. A summary of this section is presented in table 2.3.

Gasser et al.[25] recognized the value of the accessibility of open source software and proposed a framework to organize this data to support the research of the open source movement by multiple disciplines. They identified five main objects of study: Source Code and Artifacts, Processes, Projects, Communities, and Knowledge. Data for each of these objects can be found in code repositories, blogs, wikis, mailing list archives, discussion forums, documentation, FAQs, and several other web-based storage methods. Integrating these methods can be difficult, and may require a variety of extraction tools and interfaces in order to retrieve data for one research question from all of them. The paper recommends that the open source research community as a whole begin to address these integration issues by creating resources to support empirical research. They propose a few ideas on how to accomplish this, including the creation of tools to extract data from diverse sources and the creation of new repositories that provide data on open source projects to the public. Many of the papers in this review demonstrate methods that can be used to implement these suggestions.

Some studies consist of case studies of individual projects that attempt to draw conclusions about open source development as a whole. Lindman, Paganini, and Rossi[22] chose eight companies that publish open source software and conducted semi-structured interviews with employees who had the best understanding of open source licensing. The employees were asked the same questions about the decision-making process that led to a choice of open source license for their product. The semi-structured nature of the interview allowed the interviewers to ask for clarifications and additional information for each response. Information collected during these interviews was used to formulate a general model of the decision making process.

Other case studies focused on the experiences learned from of individual project efforts. Pizka[26] details the challenges encountered by long-term project to develop a distributed operating system. The team attempted to write a system from scratch; after the project fell well behind schedule they decided to start over and adapt existing code to build their project. The second approach yielded better results. The paper describes the lessons learned from this experience and provides evidence that it makes more sense to leverage existing components when building complex software systems. Kritkos, et. al.[27] suggest a reuse process based on their experiences on as developers on other projects. Their process identifies iterative steps a programmer might use to break a programming assignment into a series of requirements, and how to identify components that will meet these requirements. The authors propose that following these steps can encourage the use of open source components to produce quality software efficiently.

Surveys of open source project participants is another method used to collect data for

research papers. Li, Conradi, et al. [28, 29]surveyed IT companies in Norway, Italy, and Germany to determine what factors lead to the adoption of open source and commercial software components. Their study examined decisions made by commercial entities that publish software, so online open source project directories would be of limited use in creating their survey sample. Instead, they used traditional methods: survey recipients were selected from software company lists provided by the Norwegian Census Bureau, a comparable German agency, and the Italian yellow pages. A questionnaire designed in English, then translated into the native languages of the recipients, had received 110 usable responses at the time of publication.

Ajila and Wu[30] used a similar approach in their study of how the use of open source components affects the economic side of software development. They conducted an initial exploratory study consisting of interviews with employees of six companies. This results of this study were used to create a formal survey that was sent to a random sample of 120 software organizations in the United States and Canada.

Sen, Subramaniam, and Nelson[23] used an online survey to collect data from open source developers in their research of the factors that determine the choice of licenses used by an open source project. They identified survey participants by examining projects hosted on Sourceforge, a website that provides project management resources to open source projects. [10] Invitations to participate in the survey were sent to 2,000 programmers whose email addresses were mined from the site. The 196 usable responses they received formed the basis of their study.

Other studies use source code and other content extracted from open source project repositories. Mockus[3] relies heavily on public code repositories in his analysis of open source code history. Mockus's ambitious project started from the observation that since source code is the result of software development, an index of source code collected from all publicly accessible code repositories would assist in the research of software development methods. His paper describes the results of experiments and techniques used to collect the complete version control histories of a large sample of open source software. The project's first step was to identify repositories that hosted open source version control systems. In addition to sites such as Sourceforge, Google Code, and Savannah, which each host many projects, they also included sites for large scale or well known projects like the Linux kernel, Eclipse, and Mozilla. They also searched software directories, such as the one maintained by the FSF[11], that do not host projects but provide URLs to their source code. Information collected from the Linux distribution repositories, such as RedHat and Debian, was used to identify the most poplar software and create the project's sample. The full source code version history of the projects in the sample was collected by using scripts to copy the source code from the project's version control software and store it into an external database. Most of the projects used one of a handful of source control

[10]For more information, visit http://sourceforge.net

[11] http://directory.fsf.org/wiki/Main_Page

systems (Subversion, Git, CVS, etc) which provide APIs that can be used to extract its contents. All versions of each file in a project's source code was stored in a database. The database contained the complete version history of each project in the sample, and served to be an example of how a universal code index might be created.

Mockus used the index in a second study that examined large-scale code reuse in open source software[4]. The study examined the directory structures of project the source code and used common directory names and file contents to identify when one project reused code from another. They concluded that approximately half of the files in their code index were used in more than one project, suggesting that reuse within open source projects is common and widespread.

The work of Ananthanarayanan et al.[31] describes a method of extracting system requirements and dependency information from multiple sources. By providing system requirements, software components that meet those requirements, and dependency information in a structured format, many system administration and development analysis tasks can be automated in a way that is not practical when this information is stored in multiple, unrelated documents. The system described in this paper automates extracting this data; since most of the information is available on web pages, they used a web crawler seeded with a list of URLs to find pages related to a component. Requirements information was identified on these pages by a text analysis component. Extracted data was used to populate a dependency model represented in a structured format; this allows analytic tools to be developed that support decision making. The authors report that their experiments using the tool results in 80% of dependencies identified and 72% of them identified correctly.

Ramel's study of open source software reuse[5] used both Sourceforge and the SuSE Linux package repository as a data source. Many programming languages support the concept of packaging, where code can be grouped into packages that can be included or referenced by another program. Ramel measured external reuse by determining what percentage of packages used to develop a software product were external to the project. The study applied this metric to a sample of open source projects written in Java and C++. All of the Java projects in the sample were hosted on Sourceforge. Ramel used JDepend, a tool that analyzes dependencies of Java programs, to identify all of the packages needed for each project in the sample, and then used a custom program to determine how many of these were internal and external to the project. To apply the metric to projects written in C++, Ramel identified the shared libraries used by the project, and as with the Java applications, determined which of these were external to the project. The SuSE Linux package repository was used as a data source for the C++ programs because the software it contains is organized in a way that makes it easy to identify the dependencies of the software in the sample.

Stewart et al.[6] used open source project metrics taken from Freshmeat, an open source

project hosting site[12], in their study on the impact of license choice and organizational sponsorship on the development activity of open source projects. Their sample consisted of 138 projects from three software categories. Freshmeat provided descriptive data for each project as well as data that was used to create the metrics used in the study. Descriptive data included the age of the project, open source license type, and whether the project was affiliated with a non-profit organization. The number of subscribers signed up to receive information on a project was used to measure user interest, and the number of releases over a given time period was used to measure development activity. Analysis of the data collected from the sample projects led to the conclusion that the restrictiveness of a project's license and its organizational affiliation influence user interest in the project.

Colazzo and Fang[7] adopted a similar approach in their investigation into the impact of license choice on development activity. Their sample consisted of 244 projects hosted on Sourceforge that are included in at least one of the three most popular Linux distributions. Sourceforge provided descriptive data such as the license type, programming language, age, and description for each project. Developer membership and their coding activity, as well as the speed of development and developer turnover, were calculated by examining the version control system for each project in the sample and identifying the rate that code was produced and the number of developers producing it. Their study concluded that copyleft programs attract more developers and have shorter time between releases than non-copyleft programs.

Lerner and Tirole also used Sourceforge as a data source[8]. In addition to the descriptive data taken from each project's Sourceforge home page, a variety of supplemental data was obtained directly from Sourceforge staff. This data included website activity and the date in which a project was first posted to the site. The size of the sample appears to have been much larger than those in other studies reviewed here. Lerner and Tirole note that when they collected their data in May of 2002, Sourceforge hosted approximately 39,000 projects, and that Sourceforge staff provided them with additional data on about 10,000 of these projects. This enabled them to make several observations on the correlation between the type of open source license and various project descriptors.

[12]Freshmeat has since been re-branded as Freecode. See http://freecode.com for more information.

Summary of Open Source Research Methods	
Case Studies	
Lindman, Paajanen, & Rossi[1]	Conducted interviews with 8 companies and produced a preliminary model of the decision process that leads to the the release of a product under an open source license.
Pizka[13]	Describes the lessons learned while developing a distributed operating system. The project became successful after adapting existing open source components proved more effective than writing components from scratch.
Kritkos, Kakarontzas, & Stamelos[14]	Describes a repeatable process a programmer can use to find reusable components. The process was developed by observing methods used by programmers on previous projects.
Surveys	
Li et al[12][24]	Surveyed 110 IT companies from Norway, Germany, and Italy about their use of commercial and open source components. They found that companies who valued quality and the availability of tech support were more likely to use commercial components, and those that valued cost and access to source code were more likely to use open soure components.
Ajila & Wu[11]	Used a preliminary survey of developers and project managers to form hypotheses about how reusing open source components affects development economics, then performed a survey of 60 software companies to test these hypotheses. They found that the degree of adoption of open source components had a positive impact on cost, quality, and productivity.
Sen, Subramaniam, & Nelson[9]	Surveyed programmers that contributed to projects hosted on Sourceforge.
Repository Analysis	
Mockus[26]	Extracted source code histories from several projects hosted at muliple sources (Sourceforge, Freshmeat, Savannah, etc). This code was stored in a single database in attempt to create a master open source code repository.
Mockus[27]	The repository created in [26] was analyzed to determine reuse patterns in the source code of open source projects. Found that about half of the files in the repository appear in more than one project.
Ananthanarayanan et al[28]	Describes the design of a repository system containing resuable components. This system aims to simplify the identification of components that can fulfill a given set of requirements.
Ramel[4]	Measured the level of external reuse of Java programs hosted on Sourceforge and C++ programs taken from the SuSE Linux package repository. The study found the average external reuse level of both sets of programs to be about 50%.
Stewart, Ammeter, & Maruping[10]	Using data extracted from Freshmeat, this study investigated the impact of open source license type on user interest in a software project. They found that software released under a non-restrictive license will attract greater user interest over time than those that use a restrictive license.
Colazo & Fang[3]	Used data extracted from Sourceforge to measure the development activity and release frequency of 244 open source projects. Projects released under a copyleft license attracted more developers and had less time between releases.
Lerner & Tirole[6]	Presents a large-scale summary of open source projects hosted on Sourceforge.

Table 2.3: Methods used to research open source software

2.4 Open Source Reuse

Software reuse is the creation of new software from existing code, documentation, or other objects[1]. Effective reuse is intended to improve software quality and developer productivity while lowering production costs[32]. Proponents of open source software argue that the ability to modify source code for use in another project provides greater potential for reuse. Rather than develop a solution to a programming problem from scratch, a programmer can use an existing solution from an open source project and adapt it for a new purpose. The open source movement encourages this behavior; Raymond observes that "great programmers know what to reuse"[13].

A paper by Markus Pizka[26] describes the experiences of a research team working on a system for distributed computing and the lessons they learned about reusing open source software. The goal of their project was to produce low-level and complex software, including a compiler for a parallel and distributed programming language and the kernel that could run the resulting code. Their first approach was to implement these features by wrapping existing components to run on an existing system. To accomplish this, they developed a compiler that would translate programs written in their custom language to C++, which could then be compiled to run on a UNIX system. The resulting software worked, but performance was poor and well below an acceptable level. After attempting to increase performance by using C as the intermediate language, they concluded that the strategy of wrapping existing components was not going to produce the results they needed.

Based on this experience, the project team responsible for the kernel portion of the project decided to implement a new kernel from scratch. Their decision was based on the assumption that adapting the source of an existing kernel would prove too tedious and time consuming, and that open source software's reputation of being sparsely or poorly documented would make it difficult to adapt an existing product for their purposes. This would be especially true for a kernel, given the complexity of the tasks it would be expected to perform. Compounding these problems was the lack of documentation on how open source alternatives were implemented. The time spent figuring out how the code worked could be spent writing new code that already performed the tasks necessary for their project. However, the project quickly fell way beyond schedule. The original goal was to produce a working kernel within 2 people years, but after 6 years the code was still not usable. The reason for this was that most of their time was spent solving problems in kernel design that had already been solved by existing kernels. By choosing to start from scratch they failed to capitalize on the success of others.

The experience of trying to produce a working kernel led the project to try a different strategy when building their compiler. Instead of building one from scratch, they decided to modify GCC[13], an existing open source compiler, to compile their new language.

[13]The GNU Compiler Collection. See http://gcc.gnu.org for more information.

GCC is a very large and complex program, as it compiles several languages and runs on a wide variety of platforms. While there is documentation on how to use the software, the program's design and implementation must be learned by examining the source code. Modifying it to compile a new language was seen as very ambitious. However, progress on the new compiler occurred at a higher than expected rate. Within 3 months, a single undergraduate student with limited programming experience was able to develop a working prototype. It turned out that although reading unfamiliar source code is frustrating at first, understanding came fairly quickly. Tremendous gains were made by leveraging the work that had already been done by the GCC project. At this point, the kernel development team abandoned their efforts and began adapting the Linux kernel for their purposes. As with the compiler, work progressed much more rapidly, and the development schedule of the project as a whole moved forward at a much faster rate.

The importance of this case study is that it provides support for the argument that access to source code improves the reusability of open source projects. Despite a lack of documentation and the complexity of the tasks, significant gains in productivity can be made by leveraging the code of others. The key is to avoid solving a problem that has already been solved, and instead to focus on using existing solutions to solve new problems.

2.5 Component Reuse

The basic method of measuring reuse is to determine the percentage of pre-existing artifacts used to create a software product. This metric can be applied at various levels. For instance, a low reuse level might be measured by the percentage of lines of reused code in a module. Determining the percentage of reused modules in the project would produce a higher reuse level measurement[1]. In her study of open source software reuse, Ramel[5] examined reuse at the package level, where a package is a resource, such as a shared library, around which a programmer can structure their code. Reuse level was calculated by dividing the number of external packages used to create a software product by the total number of packages in the product. Ramel applied this measurement to two groups of software projects. The first group consisted of 112 projects written in Java retrieved from Sourceforge and open source projects hosted at Java-Source.net.[14] The second group consisted of 396 projects written in C++ taken from the SuSE Linux package repository. The study found that average external reuse level of both groups of software to be about 50%. The Java applications hosted on Sourceforge averaged 45%, and the projects from Java-Source.net averaged 52%. The C++ applications had an average external reuse level of 48%.

Ajila and Wu, in their paper *Empirical study of the effects of open source adoption on software development economics*[30], examine how the use of open source components affects

[14]For more information, visit http://www.java-source.net.

software development. Their preliminary research showed that although the use of open source components by commercial entities is on the rise, there was lack of empirical evidence on how the use of these components might affect economic considerations of cost, productivity, and product quality. To develop their hypotheses they conducted a preliminary study of software development organizations that use open source components by conducting interviews with their developers, QA managers, and project managers. These interviews sought to identify an approximation of the decision making process that leads to the use of open source components. The responses they received were used to form their research hypotheses. These hypotheses make predictions on how various aspects of open source reuse affects three basic concerns of software development: cost, productivity, and quality.

The first aspect is the degree of adoption of open source components. Ajila and Wu observed from their preliminary study that organizations attempt to create products faster, cheaper, and better than their competition in order to gain market share. The choice to reusing existing open source components was cited as a way to achieve all three of these goals. The next aspect is the maturity level of the software reuse program used by these entities. To determine the effect of maturity level on project success, the study ranked these programs on a scale ranging from "initial/chaotic" to "ingrained". Noticing a lack of research on the role a software developer's experience with reuse plays on the success of an open source reuse program, Ajila and Wu added this aspect to their study. Respondents of the preliminary survey mentioned the importance of the selection criteria used to identify components for reuse. Lastly, they posit that the size of a company impacts the adoption and effectiveness of an open source reuse program. Their hypotheses was that the degree of each of these aspects has a positive relationship to development costs, productivity, and product quality.

A survey of 60 software companies, of varying size, products, and customers, provided the data for the study. The results largely confirmed the hypotheses. The degree of adoption of open source components had positive impact on quality, productivity, and cost, in that order. Ajila and Wu observed agreement among developers that open source components tend to be of high quality and using them can lead to a higher quality product. Gains in productivity were seen in a shortened time to market. Project cost reductions were least affected by open source reuse, suggesting that gains in this area are likely to be the result of a higher quality product leading to reduced maintenance costs. Reuse maturity level was found to have a significant impact on productivity and quality, but as with the degree of adoption there was no significant effect on costs. While developer experience with reuse did have a positive affect on product quality, the survey did not find that developer reuse experience had a significant effect overall on project economics. This result did not agree with the information gathered during their literature review that reuse experience is critical to the success of a reuse program. Component selection criteria were found to have a significant impact on product quality and, to a lesser extent, on cost, but no impact on productivity. Finally, the study found no significant differences in

open source component reuse adoption between large and small companies.

Ajila and Wu summarize their conclusions by noting that open source reuse has a strong impact on software quality, a moderate impact on productivity, and the least impact on development costs. They express some concerns over the limitations of their survey methods, and suggest a larger study of open source reuse data as future work.

The study conducted by Li, et al.[28]makes an attempt to determine whether there is an advantage in choosing open source components over commercial equivalents. To answer this question, the study conducted a survey to identify characteristics of projects that use open source components vs. those that used commercial components, and to examine the motives that led to the choice of one type of component over the other. By identifying which projects use these components, and why, they can determine risks and benefits associated with the two component types. The survey sample consisted of projects that were "finished"; that is, they had completed core development for at least one release and may be in a maintenance phase. 115 software projects from Norway, Italy, and Germany completed the survey. Of these, 71 used commercial components only, 39 used open source components only, and 5 used both open source and commercial components. These last 5 were removed from the sample.

The survey results showed no significant differences between the characteristics of projects using open source components vs. those using commercial components. Both component types were used in several application domains, both were used in small, medium, and large IT companies, and both were used in projects that emphasized the importance of a shortened time to market, reliability, and performance. The expectations of projects using open source and those using commercial components were the same: both types expected their component choice to reduce development time and effort and to reduce maintenance efforts. However, the specific motives that led to the choice of one type over the other was found to be significantly different. Projects using commercial components listed quality and the ability to get technical support from the vendor as the primary reasons for their decision. The cost, availability of source code, and the possibility that a vendor may go out of business were the main motivations of projects that used open source components.

Among the conclusions drawn from these results is that users of closed source components were not more satisfied with the reliability and/or performance of their choice, even though they listed performance as a key factor in their decision. Also, projects using open source components were not less satisfied with the technical support they were able to receive than projects using commercial components, despite that the ability to get technical support from the vendor was another key reason for choosing a closed source component.

Kritikos, Kakarontzas, and Stamelos look at component reuse from the practitioner's point of view in their paper *A Semi-Automated Process for Open Source Code Reuse*[27]. Their study describes a method an individual programmer might use to determine which parts of their software project can be developed by reusing existing components. This method

is different from those proposed in other papers in that it provides a practical approach born from actual behavior, as opposed to other methods that develop a prescriptive model based on a theoretical model. Consequently their method is probably much more applicable to the situation faced by most open source developers.

Kritikos, et al. define component-based reuse as combining existing components to create new software, a definition that is consistent with that found in other papers discussed in this review. To facilitate reuse, each requirement is considered as a candidate for implementation by one component. If a requirement is too complex for this to be practical, the requirement is decomposed into smaller pieces, and the analysis is repeated. The end result is a set of requirements that can either be implemented from an existing component or is trivial enough to be implemented from scratch. A trivial requirement might include something like writing data to a file, for which numerous examples can be easily found for any programming language. The authors call these examples "bibliography code", and do not consider their use as component reuse.

At this point components that can implement the requirements must be located. This can be the most difficult part of the process. Numerous online code repositories provide thousands of freely available open source components for consideration. Assuming that these repositories can be searched effectively, there are two potential outcomes that need to be avoided. First, in some cases it might take longer to identify the right component than it would take to implement a solution from scratch. The second is that the requirement is too specific, and no such component exists. Searching for a nonexistent component uses time that would be better spent elsewhere. This paper acknowledges the need for better automated systems that can help overcome all of these challenges. Developing these systems is left as an area for future research.

The Amos Project[33], a project aimed at making it easier to build software using open source components, proposed one method of identifying components for reuse. Their paper describes a conceptual model behind the implementation of an algorithm that can be used to search a repository of reusable components. The repository stores the list of capabilities provided and the prerequisites for each component. These requirements and capabilities are expressed using a dictionary of terms that facilitate the search process. The search algorithm takes as its input the list of requirements and returns sets of components that can meet these requirements. The components in the result set have their own requirements, which become the input for the next iteration of the algorithm. Since a repository could contain thousands of packages, with several sets of components fulfilling the initial set of requirements, determining an optimal solution can be computationally prohibitive. To offset this problem, the algorithm can take into account heuristics to increase the chances that an optimal solution is found. Given the algorithm's use of first order logic to return results, the creators of the Amos project decided to use Ciao Prolog[15], an open source version of Prolog released under the GNU GPL, for their imple-

[15]http://ciaohome.org

mentation. The system would be accessed and administered through a web interface, and the descriptive information for each component stored in a database. One of the goals for the project is to produce a set of tools that an organization can use to maintain its own component repository of freely available and internally developed software.

The paper briefly discusses the problem of identifying the right components, not just those that fulfill the requirements, and to ensure that the sets of components returned by a search are compatible. While no algorithm for accomplishing this is included in the paper, the authors note two ways of handling this challenge. One is for the person using the search engine to perform the task by hand, and evaluate each set of components returned by the search. The other is for the software to use descriptive information for each component to determine component compatibility. Details on how this might be accomplished are not included the paper. Nevertheless, the paper presents a theoretical framework and clear design for implementation that promises to simplify component reuse.

Chapter 3

Methods

A review of the literature shows that the choice of an open source license can effect factors such as the number of programmers that decide to contribute to a project and the rate in which development occurs. It also shows that reuse, at various levels, is fairly widespread in the open source development community. To date, no studies examine the role the choice of license plays in influencing this reuse. This study makes an attempt to determine if license choice makes a significant impact on the reuse within and external to a software project.

3.1 Independent Variables

This study relies on two independent variables: license type and software category.

3.1.1 License Type

License Type: While there are many different open source licenses, most open source software is released under one of the five described in section 2.1.2. These are the GPL, LGPL, ASL, BSD, and MIT licenses. As demonstrated during the literature review, most research on the impacts of license choice do not consider the licenses individually, but rather group them into categories based on the restrictions they place on how modifications to the software they cover are distributed. Table II shows how some researchers have categorized these five licenses. The same approach will be taken in this study, with licenses being placed in one of two license types:

- *Restrictive*- includes all versions of the GPL and LGPL. Sometimes referred to as copyleft licenses, these licenses require that any modifications be redistributed un-

der the original license, which ensures that the source code of the software will always be freely accessible. The current GPL is a direct descendant from the original GNU license, which played an important part in establishing the open source movement.

- *Permissive-* includes all versions of the ASL, BSD, and MIT licenses. None of these three licenses require that the source code of modifications be released. An open source project might release a software component under one of these licenses to increase the potential users of their product. In addition to other open source projects, such components may be used by entities that wish to use them to create proprietary (closed-source) software.

Grouping the licenses into categories simplifies the process of identifying the implications of license choice. In addition to the differences between individual licenses discussed in section 2.1.2, there are differences between the different versions of a single license. These differences are small, and are not likely to impact the effectiveness of the licenses at promoting reuse.Grouping them into categories allows us to focus on the significant differences between open source licenses that have a direct role on reuse.

3.1.2 Software Category

Since the literature suggests that the purpose or intended audience for a software product affects license choice, it makes sense to examine reuse patterns along similar lines. All of the software components used in this study fall into one of four categories:

- *Development tools-* components providing basic building blocks for creating other programs. The components in this category include things such as programming languages, compilers, interpreters, and debugging tools.
- *Applications-* consumer oriented programs aimed at end-users. This is a broad category which includes programs such as office applications, games, multimedia players, and more. These programs are likely to have been produced using the tools in the development tools and system environment categories.
- *System environment-* components such as shared libraries and command shells which provide functionality to other programs, such as shared libraries and disk utilities. The components in this category sit between the development tools and applications; however they are not used directly by end users but rather used by developers when building applications.
- *User interface-* components that provide input and output controls used to interact with a computer system. This includes desktop window managers and mouse and

keyboard drivers. Like software in the system environment category these components are needed to run other applications, but users do interact with them directly.

As mentioned above, software that falls into the development tools and system environment categories are expected be more likely to be released under a permissive license, whereas those in the applications and user interface categories are more likely to use a restrictive license.

3.2 Dependent Variables

The dependent variables in this study are determined by quantifying the dependency relationships between software packages found in the Fedora Linux package repository.

3.2.1 Package Dependency Count

Frakes and Terry[1] identify methods of quantifying the amount of reuse. The basic method takes the form of identifying the percentage of reused objects that appear in a new software object. This metric can be applied at many different levels, such as the percentage of reused lines of code or the percentage of pre-existing components used in a project. They use the term *external reuse* to refer to the number of items taken from an external source, and *internal reuse* to refer to the number of items created specifically for use in the current project.

Ramel[5] used this method to measure reuse in open source software. She applied the basic metric at the *package* level. In this context the term *package* refers to the practice common in most programming languages of breaking functionality into units that can be imported or referenced as needed. The implementation of packaging varies across languages (jar files in Java, shared libraries in C/C++, etc) but the general concept is the same. By identifying the internal and external packages used to create the software in her sample, Ramel was able to calculate the percentage of external reuse for each of them.

In this study, the term *package* means something different: it refers to an item in the Fedora Linux software repository. A Fedora package is a software component and some descriptive information bundled for distribution and ease of installation. From this point forward the term *package* refers a Fedora package. A more detailed description of packages appears in section 3.3.2. For now it will suffice to know that a package might depend on one or more other packages in order to function properly. For instance, a word processor might rely on the spell checking capabilities provided by a third-party library that is bundled into its own package. Use of this library counts as an instance of external reuse by the word processor application. The *package dependency count* is the total number

of Fedora packages depended on by a package. This count indicates the external reuse of other Fedora packages in a manner similar to that used by Ramel[5] to identify pre-existing software components used by the applications in her sample. However, package dependency count works at a higher level of abstraction.

Note that the package dependency count includes all of a package's dependencies regardless of the license or category that the dependency belongs to. This is done because no matter the intended use of a dependency it still counts as an instance of external reuse. A component might be reused in ways not anticipated by its authors. Filtering the dependency list of a component based on dependency characteristics would be a way of placing unnecessary limits on what counts as reuse.

3.2.2 Dependent Package Count

All of the studies on open source reuse reviewed above, including Ramel's, attempt to measure the amount of reuse within a set of applications. In addition to determining the package dependency count, this study seeks to measure software reuse from a different perspective. When creating a new piece of software, a developer will identify the components, both internal and external, that are needed to provide the functionality dictated by the requirements of the program. Software development projects, both proprietary and open source, seek to use existing components in their products in order to reduce costs and time-to-market. In addition, the open source community has largely embraced the idea that re-implementing a solution to a problem that has already been solved is something to be avoided. An existing component that has been around for some time and has been used by many other projects has been well tested and is more likely to provide a high quality solution to a specific problem domain. Also, there is evidence that the availability of existing components frequently plays a large part in the design of new applications; that is, a developer will use existing components to craft a solution rather than decompose a problem into pieces and look for components to provide those pieces[34, 26, 28].

All of this suggests that there is value in determining how often a component has been reused by other software projects. Measuring how often a component is reused can help developers with evaluating the suitability of a component to their project. It also enables researchers to rank components by their reusability and identify software providers that have a record of producing effective reusable components, and thereby provide subjects for case studies on how to develop reusable software. In our case, this metric is used to determine if open source license types play a role in encouraging reuse.

Frakes and Terry define *reuse frequency* as a the number of references to a component that is being reused[1]. This metric can be used to determine to what extent a component is being reused within an application. Reuse frequency served as the inspiration for the second dependent variable used in this study. The *dependent package count* of a Fedora package is the number of times that package appears on the dependency list of another

package. This count indicates how often the package is reused. A package's appearance on a dependency list shows that it provides functionality necessary for another package to work properly. As with external and internal reuse described above, this metric can be applied at various levels. The dependent package count is applied at a relatively high level. A Fedora package can be a variety of software objects. Most often it is a library or some piece of executable code, but it can also be a configuration file, a collection of documents, or a stand-alone application. The dependent package count includes any instance where the package is depended on, regardless of the type of function the package provides. The dependent package count is the total number of other packages that require a specific Fedora package in order to run properly.

The dependent package count is intended to give an indication of the *reusability* of a package within a Linux software repository. *Reusability* has been defined as the probability that a software artifact is reusable[1, 32]. By determining how often a package has been used by other packages in the past we can get a sense to its likelihood of reuse in the future.

3.2.3 Limitations

The package dependency and dependent package count variables provide simple, high-level measurements of reuse that are well suited for investigating the relationship between license type and reuse for all of the packages within the Fedora package repository. However, their simplicity imposes some generalizability limitations.

For one, they do not take into account the size or complexity of the package being measured. This means that neither measurement can determine how much of a package's functionality is provided by the pre-existing packages it reuses. In addition, each dependency is treated equally, regardless of how a package might actually be reused. A package might be used extensively by another package, but provide less functionality to a third, but both dependency relationships are weighted the same. Also, the size and complexity of a package does not affect its dependency counts. This can lead essential system libraries to have dependent package counts that may underestimate the extent of their reuse by other packages. For example, programs written in the Java programming language require an implementation of the Java runtime environment in order to be executed. One such implementation is provided by the package *openjdk* [1]. This package also provides an implementation of the standard API that is likely to be used extensively throughout most Java applications. *openjdk* may provide the bulk of the functionality needed by an application, but that application's dependency on *openjdk* counts once when determining the application's package dependency count.

Despite these limitations, the package dependency and dependent package counts identify reuse relationships between packages in the Defora repository. These counts can be

[1] http://icedtea.classpath.org/

used to identify high-level trends in reuse, which in turn can be used to examine the role license type plays in predicting reuse.

3.2.4 Package Dependency Count Hypotheses

The most restrictive open source licenses, the GNU GPL and LGPL, are also the licenses most closely identified with the open source movement. These licenses were created to encourage modification of source code. Since license choice is also a way for projects to signal their beliefs on how software should be created and distributed, projects choosing to release software under a restrictive license are receptive to the idea of incorporating existing components in their designs. Consequently, we expect the package dependency counts of software released under a restrictive license to be higher than those for packages released under a permissive license. The basic package dependency count hypothesis tested in this study is:

$$H_1 : \tilde{\chi}_r > \tilde{\chi}_p$$

where $\tilde{\chi}_r$is the median package dependency count of software released under a restrictive license and $\tilde{\chi}_p$ is the median package dependency count of software released under a permissive license.

Does this hypothesis apply to software in different categories? It is expected that the answer to this question is yes, and that package dependency counts will be higher for packages released under a restrictive license type regardless of software category. All software is created by programmers, and those who share the opinions of open source development promoted by restrictive licenses may be more likely to reuse existing components. This study examines reuse within four software categories, which leads to four package dependency count hypotheses.

Hypothesis 2 is that packages in the software development tools category released under a restrictive license will have a greater median package dependency count than those choosing a permissive license:

$$H_2 : \tilde{\chi}_{rd} > \tilde{\chi}_{pd}$$

where $\tilde{\chi}_{rd}$is the median package dependency count of software development tools released under a restrictive license and $\tilde{\chi}_{pd}$ is the median package dependency count of software development tools released under a permissive license.

Hypothesis 3 is that packages in the applications category released under a restrictive license will have a greater median package dependency count than those choosing a permissive license:

$$H_3 : \tilde{\chi}_{ra} > \tilde{\chi}_{pa}$$

where $\tilde{\chi}_{ra}$is the median package dependency count of applications released under a restrictive license and $\tilde{\chi}_{pa}$ is the median package dependency count of applications released under a permissive license.

Hypothesis 4 predicts that packages in the system environment category released under a restrictive license will have a greater median package dependency count than those choosing a permissive license:

$$H_4 : \tilde{\chi}_{rs} > \tilde{\chi}_{ps}$$

where $\tilde{\chi}_{ra}$is the median package dependency count of system environment packages released under a restrictive license and $\tilde{\chi}_{pa}$ is the median package dependency count of system environment packages released under a permissive license.

Hypothesis 5 is that packages in the user interface category released under a restrictive license will have a greater median package dependency count than those choosing a permissive license:

$$H_5 : \tilde{\chi}_{ru} > \tilde{\chi}_{pu}$$

where $\tilde{\chi}_{ru}$is the median package dependency count of user interface components released under a restrictive license and $\tilde{\chi}_{pu}$ is the median package dependency count of user interface components released under a permissive license.

3.2.5 Reusability Hypotheses

Permissive licenses place fewer restrictions on how a piece of software can be modified and redistributed. Fewer restrictions should increase the attractiveness of the software for reuse in a different project. The number of other projects that might reuse the software should be larger than the number of projects that would reuse a similar component released under a restrictive license. This suggests that software packages released under a permissive license will have higher dependent package counts than packages released under a restrictive one. This leads to the basic hypothesis tested in this study:

$$H_6 : \tilde{\chi}_r < \tilde{\chi}_p$$

where $\tilde{\chi}_r$ is the median dependent package count of software released under a restrictive license and $\tilde{\chi}p_r$ is the median dependent package count of software released under a

permissive license.

Lerner and Tirole[6] observe that software developed for other programmers are significantly less likely to be released under a highly restrictive license. Development tools, by their nature, are intended to be used to create new products, and the use of a permissive license may encourage this. This suggests that tools released under these licenses are reused more often. Hypothesis 7 is:

$$H_7 : \tilde{\chi}_{rd} < \tilde{\chi}_{pd}$$

where $\tilde{\chi}_{rd}$is the median dependent package count of packages in the software development tools category released under a restrictive license and $\tilde{\chi}_{pd}$ is the median dependent package count of packages in the software development tools category released under a permissive license.

The same study found that applications targeted at end users are much more likely to be released under a restrictive license. While such applications are typically not intended to be used to create new software products, it is possible for them to fulfill a dependency of another program. In this case it is expected that applications packages released under a restrictive license are reused more often. Hypothesis 8 is:

$$H_8 : \tilde{\chi}_{ra} > \tilde{\chi}_{pa}$$

where $\tilde{\chi}_{ra}$is the median dependent package count of packages in the applications category released under a restrictive license and $\tilde{\chi}_{pa}$ is the median dependent package count of packages in the applications category released under a permissive license.

System environment packages provide shared libraries necessary for other packages to function. They are similar to development tools packages in that they are intended for use by other programmers to add functionality to applications. As such, it is expected that permissive licensing encourages reuse of these packages more effectively than restrictive licenses. Hypothesis 9 is:

$$H_9 : \tilde{\chi}_{rs} < \tilde{\chi}_{ps}$$

where $\tilde{\chi}_{rs}$is the median dependent package count of packages in the system environment category released under a restrictive license and $\tilde{\chi}_{ps}$ is the median dependent package count of packages in the system environment category released under a permissive license.

The fourth category is user interface packages. These packages are used to render graphical output and receive input from a user, and are essential for use by end user applications. For these packages, permissive licenses are predicted to foster reuse more effec-

tively then restrictive licenses. Hypothesis 10 is:

$$H_{10} : \tilde{\chi}_{ru} < \tilde{\chi}_{pu}$$

where $\tilde{\chi}_{ru}$is the median dependent package count of packages in the user interface category released under a restrictive license and $\tilde{\chi}_{ps}$ is the median dependent package count of packages in the user interface category released under a permissive license.

3.3 Data Collection

3.3.1 Dependency Analysis

The package dependency and dependent package counts were determined through dependency analysis. A dependency relationship exists between two Fedora packages if one must be present for the other to function. Package *x* is a dependency of package *y* if *x* must be present in a system in order for *y* to be usable. One advantage of building software from readily available packages is that when a modifications are made only the affected package needs to be upgraded. The other packages used by an application do not need be changed. In order for this to work it is necessary to be able to identify the dependencies of each package in order to determine which software is affected when a package is modified.

Managing the dependency relationships between a large number of packages, such as can be found in an open source software repository, can become very complex. In their study of dependencies within the Debian Linux software repository, Abate, Boender, et al.[35] describe some useful definitions for these relationships that are applicable to this project. A *repository* is a set of packages and the dependencies between them. As defined earlier, a package is a software object bundled for distribution. The actual contents of a package can vary widely, though in most cases a package consists of either a stand-alone application or a library of executable code intended to provide functionality to other packages. Abate, Boender, et al.[35] refer to the type of dependency described above as a *strong dependency.* Stated formally, a strong dependency exists between two packages *p* and *q* in repository *R* if for every installation of a subset of packages from *R* that includes *p* also includes *q.* This relationship can be expressed as:

$$p \Longrightarrow Rq$$

When *p* is installed on a system, its strong dependency with *q* must be identified, as well as any packages on which *q* strongly depends, and so on until the entire dependency

tree has been explored. Upon completion q's dependencies are said to be *resolved*. Most repository management software includes tools to automate dependency resolution and greatly simplify the installation and removal of packages from the system.

Dependency resolution can be used to determine both the package dependency count and dependent package count of a Fedora package. To continue the previous example, package p has an package dependency count of at least 1, as it uses the functionality provided by q. The package dependency count of p is therefore the total number of packages with which p has a strong dependency relationship.

Conversely, dependent package count is determined by looking at dependency resolution from the point of view of the package being depended on. Package q has a dependent package count of at least 1, as it provides functionality critical for the use of p. The complete dependent package count for q is therefore the total number of packages for which q fulfills a dependency.

The basic method used to collect data for this study involved choosing a package repository to provide a sample of open source packages, and using that repository's dependency information and dependency resolution tools to determine the package dependency and dependent package counts for each package in the repository.

3.3.2 Software Repository

To perform this study it was important to identify a software repository that contained a good representation of open source packages and provided a method by which license type, software category, and dependency relationships could be extracted. Linux software repositories meet these criteria. Most Linux distributions maintain their own repository of open source software, which is in turn implemented using one of several available package management systems. This study used the Fedora Linux[2] repository as a data source. A number factors influenced this decision:

1. Fedora originated as a community version of Red Hat, one of the leading enterprise-class Linux distributions. As such, it inherited a stable platform and lots of high quality packages. Fedora is a robust and well supported system that serves as a test bed for Red Hat Enterprise Linux.

2. Fedora is one of the most popular Linux distributions. DistroWatch, a website that tracks information about hundreds of Linux distributions, currently ranks Fedora as the third most popular distribution[3]. Fedora's popularity helps ensure that its package repository is kept up to date with the most frequently used current open source software.

[2]For more information visit http://fedoraproject.org/
[3]http://distrowatch.com/dwres.php?resource=major

3. The RPM Package Management system, which Fedora uses to organize its repositories, tracks the metadata needed to carry out this study and provides tools which can be used to automate data collection.

Fedora releases a new version of its operating system approximately every 6 to 8 months. Each version has its own repository. While repositories are constantly being updated as new software becomes available and new versions of existing software are released, the majority of the packages are migrated from previous versions into the repository of a new release. This means that there is not much of a difference in the packages included in each version's repository, and that the repository for one version of Fedora provides a comprehensive sample of open source software. For this study, the repository of the most current Fedora release, Fedora 16, was used as a data source.

Within a version's repository, packages are divided into two categories, 32bit and 64bit. An individual Fedora installation will use packages from one of these two groups according to the hardware used to run the installation and whether a 32bit or 64bit version of the operating system was installed. Most of the software included in the repository appears in both categories, so data was only extracted from the packages in one of them, the 64bit category. An online version of the Fedora Package Database can be accessed at https://admin.fedoraproject.org/pkgdb.

The individual packages within the repository consist of open source programs that have been bundled into an RPM file. This file contains a compiled binary version of the program, configuration files, and documentation that are copied to a Fedora system during installation. The RPM also includes descriptive information about the program, the most important of which for this study is the dependency list of other RPMs that must be present on a Fedora system before the package can be installed. Individual RPMs are created and maintained by volunteers. A packaging committee provides the guidelines that packages are expected to conform to, and must approve each addition to the repository. This helps ensure that software in the repository is of sufficient quality and is aligned with the underlying objectives of the Fedora project.

3.3.3 Limitations of Using a Fedora Repository As a Data Source

There are some limitations and drawbacks of using a Fedora Linux repository as a data source. First, the decision of the repository maintainers as to which packages to include and which to exclude is a form of selection bias that may limit the applicability of the package dependency and dependent package counts extracted from the repository to open source software as a whole. It is hoped that this limitation might also be a strength. Other studies that have relied on the various forges as a data source must contend with a large amount of software of dubious quality, as well as the problem of omitting software hosted in repositories not used as a source. The selection bias of using the Fedora

repository mitigates these drawbacks by providing a sample of open source packages that were successfully compiled into a functional state and drawn from a variety of host sites. Still, it is difficult to determine if this selection bias results in the exclusion of a significant number of worthy packages from this study.

Use of the Fedora repository as a primary data source may limit the generalizability of the results of this study to other open source software collections. The package dependency and dependent package counts tally the dependencies between Fedora packages. These specific dependencies are defined by the volunteers who create the packages, and may not be the same as those defined by the maintainers of repositories for other Linux distributions. However, differences in dependency relationships for the same software in different repository systems are not expected to be significant. Most open source software projects do not include the source code of their dependencies when distributing their own source code. If one open source application requires features provided by a code library, for example, users are expected to download the library from the website maintained by the library's authors. Package creation tends to mirror this process by bundling the individual applications into their own packages. This should ensure some consistency in how software is bundled by the various Linux distribution maintainers. Package dependency and dependent package counts should therefore be generally applicable to software provided in other open source package repositories.

Another potential source for error is the descriptive information provided for each package. Information such as software category and license type is provided by volunteer contributors. While a committee approves each package submission, the number of packages in the repository increases the chance that one of these items is an inaccurate for a subset of all packages. An error of this kind would not prevent a package from functioning, and thus might not be discovered and corrected for some time. Also, the assignment of a software category requires a judgment call on the part of the package maintainer, which inevitably results in some packages being assigned to one category when they just have easily been assigned to another. Hopefully the number of packages in each category will reduce the impact of such assignments.

Despite these drawbacks, the benefits of using the Fedora Linux repository for this study should on the whole outweigh the disadvantages, and provide a better representation of open source software that is actively used and supported.

3.3.4 Data Extraction

The original plan for data collection was to use the *Yum*[4] utility to extract licensing and dependency data from the repository. *Yum* is a tool for managing the installation of RPMs from one or more repositories. Its primary purpose is to resolve and manage dependen-

[4]http://yum.baseurl.org/

cies. When a user uses it to install a package, *Yum* will determine which dependencies are needed that are not already installed, and install them as well. It will also detect conflicts if the requested package or one of its dependencies is not able to be installed. This makes it very easy for a user to add and remove programs on an RPM-based system. *Yum* supports multiple repositories; given a list of repositories, it will resolve searches for packages and their dependencies across all of them. This makes it easy to add third party repositories, which often contain non-open source software, to a system.

Yum also allows a user to query a repository for information about packages. The first attempt to collect data for this project made extensive use of this feature. A copy of the Fedora 16 repository was downloaded from a server on the Fedora repository mirror list[5] and stored on a web server on a local area network. This was done to provide a high bandwidth connection to the repository and reduce the amount of time needed to execute queries against it. A script written in the Perl programming language was used to determine the license, software category, and reuse count for each package, and store the results into a relational database. The script performed three basic steps. First, it used *Yum* to query the repository for a list of all of its packages. Second, for each package in the list, it used *Yum* to query the license and software category. Last, it would use *Yum* to query the repository for a list of all of the other packages which list the package as a dependency, and assigned the number of items in the result as the package's reuse frequency.

This script worked, but had some drawbacks. First among these was performance. The Fedora 16 repository for the 64bit architecture contains a little more than 10,000 entries. Even with the increase in connection speed gained by using a locally hosted repository it took close to 48 hours to complete one execution of the script. The other drawback was that using a script increased the risk of introducing errors in the data. Thorough testing was required to ensure the data collected by the script was accurate. This required repeated execution of the script, increasing the impact of the script's performance.

In the course of double-checking the results of queries using *Yum*, it was discovered that the *Yum* itself was basically a front end for another relational database. Repositories managed by *Yum* store package metadata in an SQLite[6] database. SQLite is a database implementation that saves its data in files and does not require a database server. It is a popular choice for application developers who want to their programs to save complex data types without incurring the overhead necessitated by traditional relational database implementations. When *Yum* is used to query information about a package, it in turn runs a query against an SQLite database contained in a file hosted on the repository site.

[5]http://mirrors.fedoraproject.org/publiclist/
[6]http://www.sqlite.org/

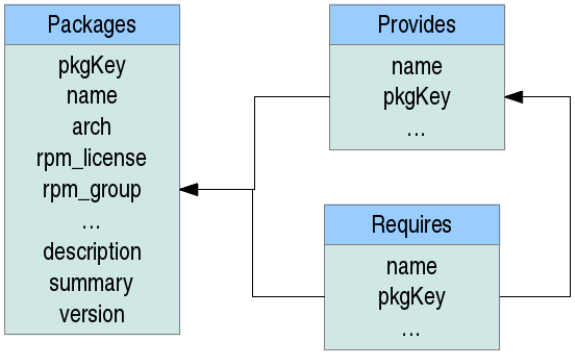

Figure 3.1: Abbreviated *Yum* database entity relationship diagram

Figure 3.1 shows an entity relationship diagram for a *Yum* database, abbreviated to show the columns used by this study to extract external reuse and reuse frequency data. The package table contains descriptive information, such as name, description, license, architecture (32 or 64 bit), and software category (rpm_group), for each package in the repository. This table serves as a master index of all the software available for installation on a Fedora system. The Requires table contains the names of dependencies indexed by the ID of the package that requires it. The Provides table lists the name of each dependency, index by the ID of the package that provides it.

It became apparent that a much more efficient approach would be to query the SQLite database directly, and not use *Yum* at all. SQL queries were written to join the Packages, Provides, and Requires tables, and determine for each package its number of dependencies and the number of times it was depended on. To facilitate the identification of packages with the correct license type, two additional tables, Restrictive and Permissive, were added to the database. As there names suggest, these tables contained lists of those licenses in the two license groups. By joining against these tables packages that do not belong to the group were filtered out of the result sets. As an example, here is the query used to extract the package dependency count of applications released under a restrictive license.

```
select a.name, count(distinct d.pkgKey)
from packages a
left outer join requires b on a.pkgKey=b.pkgKey
left outer join provides c on b.name=c.name
left outer join packages d on c.pkgKey=d.pkgKey
join restrictive e on a.rpm_license=e.rpm_license
where a.arch='x86_64' and a.rpm_group like 'Applications%'
group by a.name
```

Figure 3.2: External reuse SQL query example

A complete list of the queries used to collect data for this study is included in the appendix.

A database management tool called sqliteman[7]was used to execute the queries. Sqliteman accepts SQL queries as input, applies them to an SQLite database, and either displays the results on screen or saves them to a file. Since the database itself is contained in a single file, it could be downloaded to the computer running sqliteman and eliminate the need to send queries over a network. This, as well as eliminating the need for *Yum*, was responsible for a large performance increase. SQL makes it possible to query data for large numbers of packages concurrently; for instance, instead of running 10,000 individual queries to retrieve the license for each individual package in the database, one query can be used to get the license for all 10,000 packages. This made it possible to develop queries to extract the necessary data, test them, and verify their results in less time than it took to complete a single iteration of the original script.

In all, ten sets of data were extracted, two for each of the ten hypotheses listed above. These data sets are summarized in table 3.1. Each data set consists of the external reuse and reuse frequency for each software package in the group relevant to each hypothesis. For example, the first hypothesis is interested in the external reuse of software released under permissive and restrictive licenses. One data set used to test this hypothesis contains a list of reuse counts for all of the packages in the repository that was released under a license in the permissive category, and the other data set contains reuse counts for each package released under a restrictive license.

Each data set was stored in a plain text file to facilitate being imported into the R statistical analysis tool.

[7]http://sqliteman.com/

Data Set Descriptions
Hyptothesis 1: Permissve vs. Restrictive Data set A: Package dependency count of each package with a permissive license Data set B: Package dependency count of each package with a restrictive license
Hypothesis 2: Software Development Tool Category- Permissive vs. Restrictive Data set A: Package dependency count of each software development tool with a permissive license Data set B: Package dependency count of each software development tool with a restrictive license
Hyptothesis 3: Application Category- Permissve vs. Restrictive Data set A: Package dependency count of each application with a permissive license Data set B: Package dependency count of each application with a restrictive license
Hypothesis 4: System Environment Category- Permissive vs. Restrictive Data set A: Package dependency count of each system evironment package with a permissive license Data set B: Package dependency count of each system environment package with a restrictive license
Hypothesis 5: User Interface Category- Permissive vs. Restrictive Data set A: Package dependency count of each system evironment package with a permissive license Data set B: Package dependency count of each system environment package with a restrictive license
Hyptothesis 6: Permissve vs. Restrictive Data set A: Dependent package count of each package with a permissive license Data set B: Dependent package count of each package with a restrictive license
Hypothesis 7: Software Development Tool Category- Permissive vs. Restrictive Data set A: Dependent package count of each software development tool with a permissive license Data set B: Dependent package count of each software development tool with a restrictive license
Hyptothesis 8: Application Category- Permissve vs. Restrictive Data set A: Dependent package count of each application with a permissive license Data set B: Dependent package count of each application with a restrictive license
Hypothesis 9: System Environment Category- Permissive vs. Restrictive Data set A: Dependent package count of each system evironment package with a permissive license Data set B: Dependent package count of each system environment package with a restrictive license
Hypothesis 10: User Interface Category- Permissive vs. Restrictive Data set A: Dependent package count of each system evironment package with a permissive license Data set B: Dependent package count of each system environment package with a restrictive license

Table 3.1: Data sets collected for each hypothesis

Chapter 4

Results

In this section we examine the data collected from the repository to determine whether there is any support for the hypotheses presented in section 3. We start with an overview of package dependency and dependent package counts for all packages in the repository, and then examine how license type affects reuse in both the repository as a whole and by software category.

4.1 Summary Statistics

First we look at some high-level descriptions of the data across all license types.

4.1.1 Package Counts by License Type and Category

Package Counts by License Type and Category: The Fedora 16 repository for 64-bit architectures contains 12,261 packages. Of these, 77% are released under one of the five open source licenses being considered in this study. Packages not released under one of these licenses were excluded from the study, leaving a sample of 9,570 packages for analysis. The licensing of these packages is summarized in table 4.1:

License	Package Count	% of Total
GPL	5,019	52%
LGPL	2,244	24%
BSD	1,166	12%
MIT	814	9%
ASL	327	3%
Total	9,570	100%
Total Restrictive	7,263	76%
Total Permissive	2,307	24%

Table 4.1: Package Count by License

The table shows that restrictive licenses in general, and the GPL and LGPL in particular, are the most common licenses used by software in the repository. This is consistent with Lerner & Tirole's overview of open source software[8] that found the GPL and LGPL to cover 72% of their sample, and suggests that the GNU vision of open source development is still dominant in the minds of developers.

Next we consider the number of packages in each of the four software categories.

Development Tools	3,440
Applications	3,012
System Environment	2,402
User Interface	409
Total	9,263

Table 4.2: Package Count by Groups

More packages belong to the development tools category than any other. This may reflect the appeal of the Linux operating system to programmers, open source advocates, and technically proficient users who enjoy creating new software tools. Open source software encourages the modification of source code, which in turn could lead to relatively more tools to make changing code and creating new applications easier. The applications category has the second highest number of packages. Critics of Linux cite its complexity and lack of applications as contributing factors to its failure to maintain a significant presence in the desktop computing market. While these criticisms are not always correct, the relative number of development tools to applications may show the preferences of developers who write software for the platform. The user interface category has the fewest packages, about one fifth of the number of the next largest category. The user interface category includes a number of components used to render graphics, such as the X window system. These components are often very complex, and are shared by programs in

order to provide a consistent look and feel across many applications. These two factors may explain the relatively small size of this category.

Note that 307 of the 9,250 packages examined in this study are not accounted for in this table. Most of these 307 packages are not associated with any group and the repository lists their category as "unspecified". The rest belong to categories of which they are the only members, producing a sample size that is too small to perform a meaningful analysis.

	Software Category							
	Development Tools		Applications		System Environment		User Interface	
License	Count	% of Category	Count	%	Count	%	Count	%
GPL	1,219	35%	2,337	78%	1,055	44%	251	64%
LGPL	1,137	33%	294	10%	685	29%	34	8%
MIT	395	12%	99	3%	185	8%	107	26%
BSD	549	16%	219	7%	358	15%	17	4%
ASL	140	4%	63	2%	119	5%	0	0%
Restrictive	2,356	68%	2,631	87%	1,740	73%	285	70%
Permissive	1,084	32%	381	13%	662	28%	124	30%

Table 4.3: Package Counts by License and Software Category

Table 4.3 breaks down each software category by individual license. It summarizes the number of packages by license type and software category. It shows the number of packages released under a given license, and what percentage of all packages in that category this count represents, for each of the four software categories. The final row of the table shows the counts and percentages of restrictive and permissive licenses for each group. The dominance of the GPL and LGPL seen in the repository as a whole is also reflected in each category, particularly among applications where it is used by 87% of packages. It is interesting to note that the Apache license is the least used license for each group. This license is the most restrictive of the permissive licenses, and one might expect it to have a count closer to that of the restrictive licenses. It's also the youngest of the five licenses, which may explain why it has achieved only a fraction of the adoption of the other four licenses.

4.1.2 Package Dependency Count

Package dependency counts were determined by counting the number of packages each package depends on. It provides a way to measure the number of pre-existing components used to develop the software and gives an indication of reuse within a specific package. It is useful to examine the dependency counts for packages released under both restrictive and permissive licenses to serve as a point of comparison of the individual

groups. Figure 4.1 contains a histogram showing the frequency distribution of dependency counts reuse for all packages.

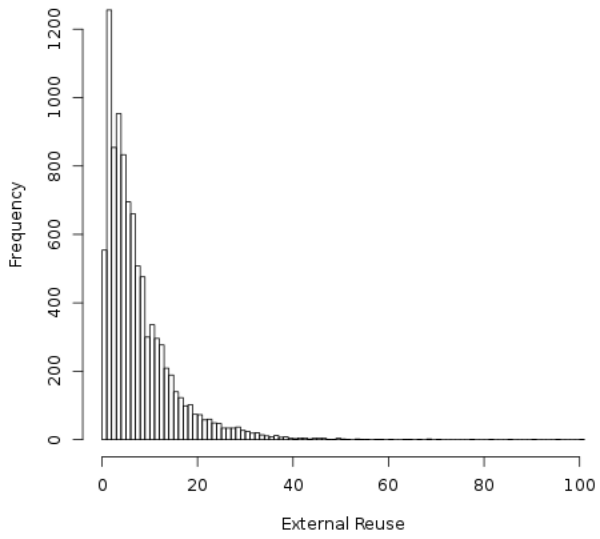

Figure 4.1: Histogram of package dependency counts of all packages

This graph allows us to make some observations on package dependency in the Fedora repository. One is that the data is not normally distributed. The distribution is positively skewed and forms a reverse-J shape. For counts greater than 1, as the package dependency count increases, the number of packages with that count decreases. Plainly stated, packages are more likely to reuse fewer packages, and packages reusing many packages become rare as the number of reused packages increases.

1st Quartile	3	Range	0 - 101
2nd Quartile	6	Median	6
3rd Quartile	11	Mean	8.181
4th Quartile	101	SD	7.606
n	9,570		

Table 4.4: Summary of package dependency counts of all packages

75% of the packages have a dependency count of 11 or less, a count that is much smaller than the maximum of 101. The 4th quartile, which contains 2,145 packages, accounts for most of the values in the range, which causes the mean package dependency count to

exceed the median. The quartile also contains a number of outliers, which are readily apparent in a box plot of the data.

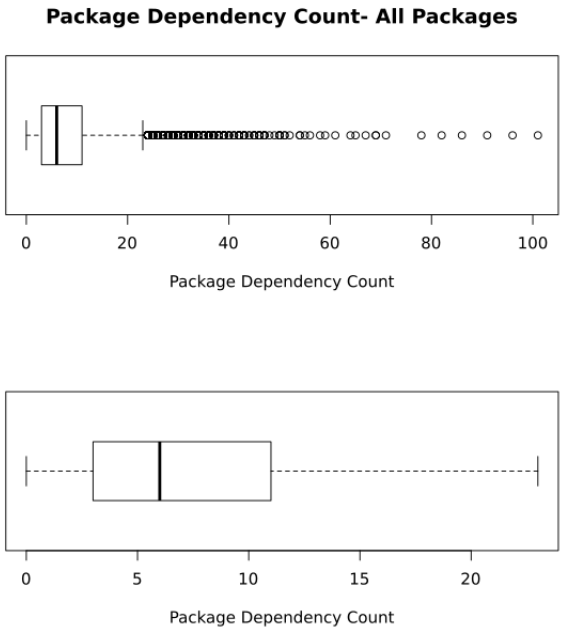

Figure 4.2: Box plot of the package dependency counts for all packages

As figure 4.2 shows, there are many outliers in the data set. The number of packages in the first 3 quartiles means that any package reusing more than 22 packages is an outlier. These outliers tend to be complex pieces of software that perform many tasks which can be implemented using many existing components. For instance, the package with the highest package depenency count is *Leksah*[1], an integrated development environment (IDE) for the Haskell programming language. IDEs typically provide programmers with compilers, debugging tools, profilers, and other development tools all in one application. It is not surprising that such a tool has the highest package dependency count in the repository.

The package dependency count data for many of the license type and category subsets described below will share the same characteristics of the package dependency count data for all packages in the repository. All of the groups show a concentration of values in the lower part of the range, with a number of extreme outliers in the 4th quartile.

[1]http://leksah.org/

4.1.3 Dependent Package Count

Dependent package counts were determined by counting the number of times a package appears on the dependency list for another package. This count shows how many times a package has been reused in the creation of a new package. An overview of the dependent package count of all packages in the repository offers some interesting contrasts to the dependency count data in the previous section.

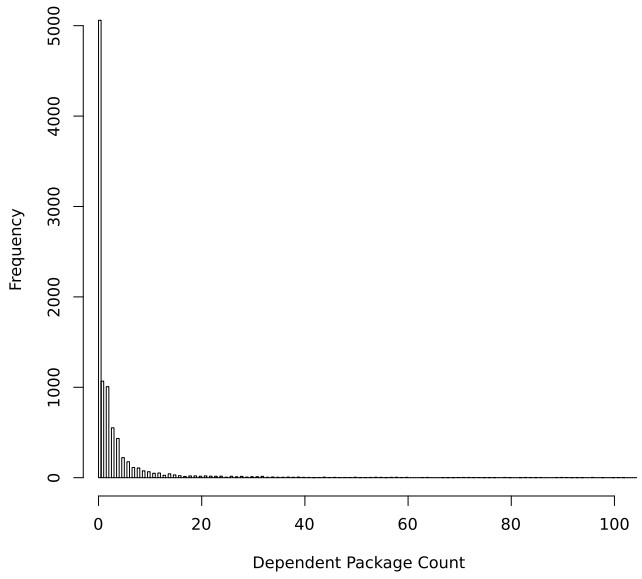

Figure 4.3: Histogram of the dependent package counts of all packages

The basic trend of the data, with frequency decreasing to the right, shows the same positively skewed J-shaped distribution as the package dependency count histogram in the previous section. The graph shows that packages that are frequently reused by other packages become increasingly rare as the dependent package count increases. This trend may demonstrate the difficulty in designing software for reuse.

This is where the similarities between package dependency and dependent package counts end. The most striking difference between the two data sets is how often a measurement of zero occurs in each group. Only 127 packages, or 1.3% of the sample, have a package dependency count of 0. This contrasts sharply with the dependent package count data, which shows that more than half of the packages have 0 dependent packages. Because these packages are not reused at all, the median dependent package count of the sample is also zero.

1st Quantile	0	Range	0 - 1850
2nd Quantile	0	Median	0
3rd Quantile	3	Mean	5.684
4th Quantile	1850	SD	45.625
n	9,570		
Packages with a count of zero	5,060		

Table 4.5: Summary of the dependent package counts of all packages

The prevalence of zero counts throughout all of the dependent package count subsets poses a challenge in interpreting the data. On one hand, a large number of zeros in a data set might demonstrate that a particular factor (license type or software category) discourages the reuse of packages in that group. On the other hand, these zeros drive the median dependent package count down to the point where it affects whether there are statistically significant differences between two groups. One way to handle this issue is to remove packages with zero dependent packages from the sample. Table 4.6 shows the summary statistics of the dependent package counts for all packages after removing counts equal to zero.

1st Quantile	1	Range	1 - 1850
2nd Quantile	2	Median	3
3rd Quantile	3	Mean	12.06
4th Quantile	1850	SD	65.885
n	4,510		

Table 4.6: Summary of the dependent package counts greater than zero

In addition to reducing the sample size to less than half of the original sample, both of the mean and median dependent counts increases significantly. However, as shown by histogram 4.4, the overall shape of the distribution remains the same.

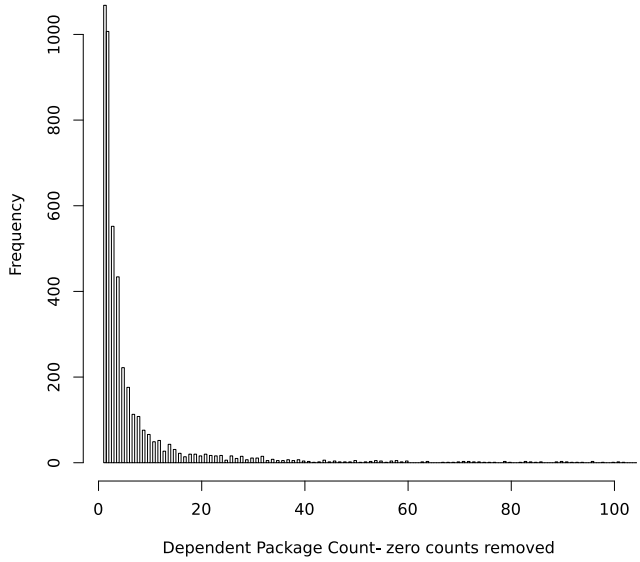

Figure 4.4: Histogram of dependent package counts greater than 0

Since the distribution shape was not affected, the decision was made to leave dependent package counts of zero in all of the data sets used for analysis. Omitting them entirely would raise the median dependent package counts while simultaneously ignoring the possibility that certain license types are not as effective at promoting reuse as others.

In contrast to the large number of packages that are not reused are the relatively small number of packages that are reused extensively. The maximum dependent package count is 1,850. The package with this count is *glibc*, the implementation of the C Library that is standard on most Linux systems and is likely to be used by any program written in C. The other extreme outliers provide similar functionality. As a result, the outliers for dependent package count data are much more numerically distant from the rest of the data than the package dependency count outliers.

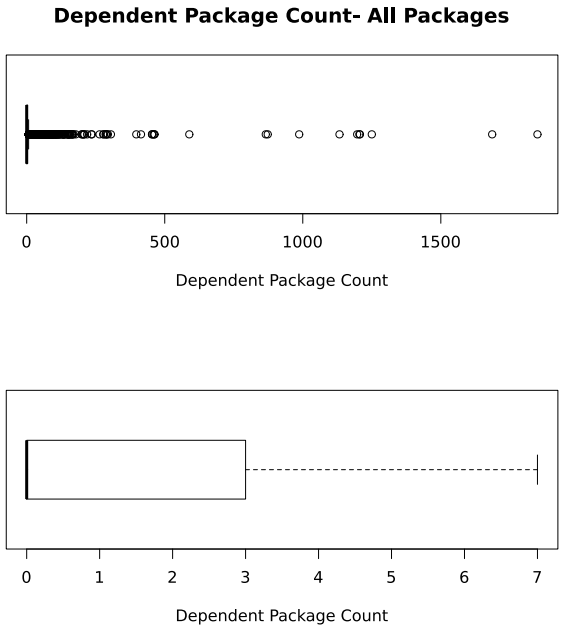

Figure 4.5: Box plot of the dependent package counts of all packages

The number of outliers and their distance from the first three quartiles make it difficult to view the values associated with the median and percentiles of the box plot. Figure 4.5includes a second plot with the outliers not shown so that the entire box can be seen.

4.2 Package Dependency Count

The data used to determine package dependency counts for the repository as a whole was divided into subsets by license type and software categories. There were five subsets in total, one for each package dependency count hypothesis.

4.2.1 Restrictive vs Permissive

The package dependency counts for all packages discussed in section 4.2.2 were separated into two groups according to license type. The first consisted of packages released under a restrictive license, and second contains those released under a permissive license. Both sets include packages from all software categories. The nature of each of these two sample sets are not drastically different from that of the whole.

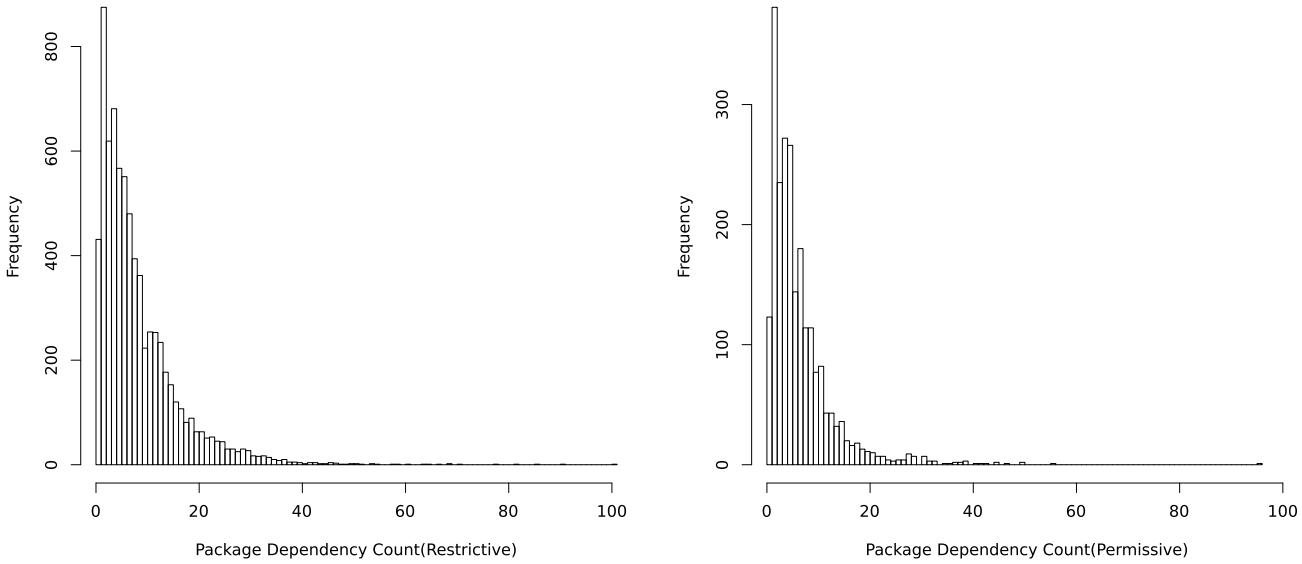

Figure 4.6: Histograms of the package dependency counts of all packages by license type

The histogram of each group shows the same reverse-J pattern as that of all packages, that frequency decreases as reuse count increases. However, a look at the descriptive statistics of the two groups in table 4.7 shows that the packages in the restrictive group may possess higher reuse counts.

Restrictive		Permissive	
Range	0 - 101	Range	0 - 96
1st Quartile	3	1st Quartile	3
3rd Quartile	12	3rd Quartile	9
Median	6	Median	5
Mean	8.606	Mean	6.842
SD	7.888	SD	6.463
n	7,263	*n*	2307

Table 4.7: Summary of the package dependency counts of all packages by license type

This can also be seen in box plots for each group. In addition to having a greater median reuse count, the packages with a restrictive license have a greater number of outliers than those with a permissive license, contributing to the impression that restrictive packages exhibit greater reuse.

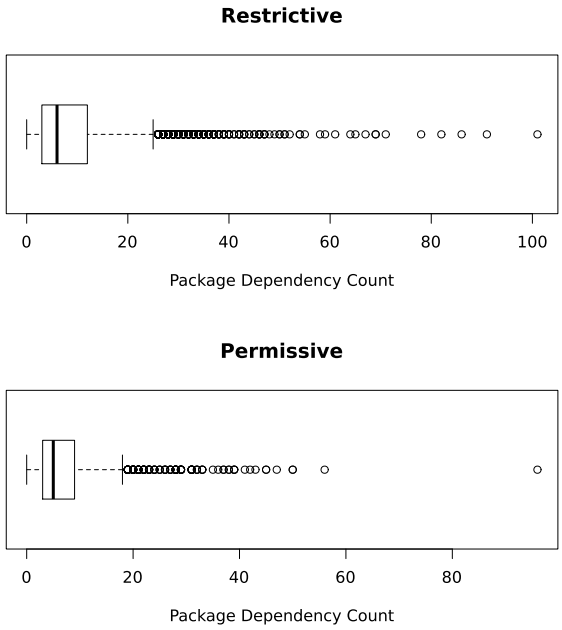

Figure 4.7: Box plots of package dependency counts of all packages by license type

Are these apparent differences correct and statistically significant? To answer these questions, the medians are compared through the use of notched box plots, which place an indentation, or notch, around the median. If the notches of two (or more) graphs do not overlap, this indicates that the medians are significantly different at a confidence level of 95%[36].

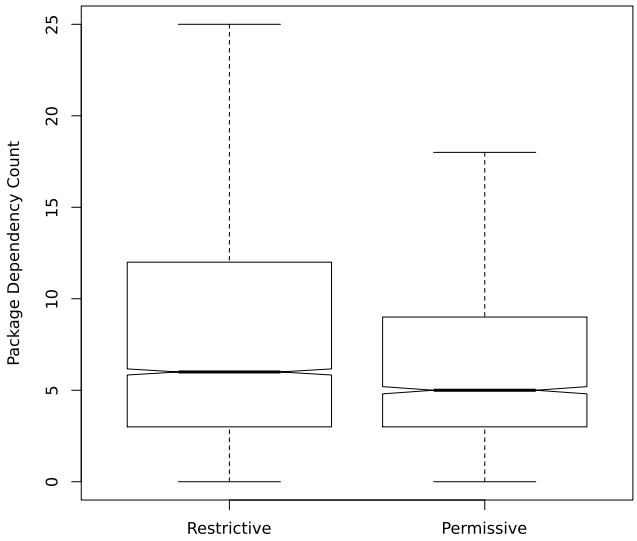

Figure 4.8: Comparison of median package dependency count of all packages by license type

As in figure 4.5, the outliers have been removed to make it easier to read the graph. The outliers were used in the calculation to determine the placement of the notches, but omitted from the final plot. This makes it easier to determine the extent to which notches overlap. For this reason they have been omitted from all of the notched plots that appear in this study.

In this case, the notches of the two box plots do not overlap, indicating a significant statistical difference between the medians of the restrictive and permissive groups. This confirms the prediction made in hypothesis 1 that software packages released under a restrictive license exhibit a higher level of reuse than those using permissive licenses. However, the difference between the two medians is not very large. Whether this difference is important will be discussed in section 5.

4.2.2 Package Dependency Counts by Software Category

The rest of section 4.2 will examine differences in the package dependency counts of restrictive and permissive software packages within four software categories. First it is helpful to determine if there is a significant difference between the package dependency counts of these four categories without taking license type into account. This will provide

a useful baseline to which results of the comparisons of license type can be compared. Figure 4.9 shows a notched box plot that contains graphs for each category.

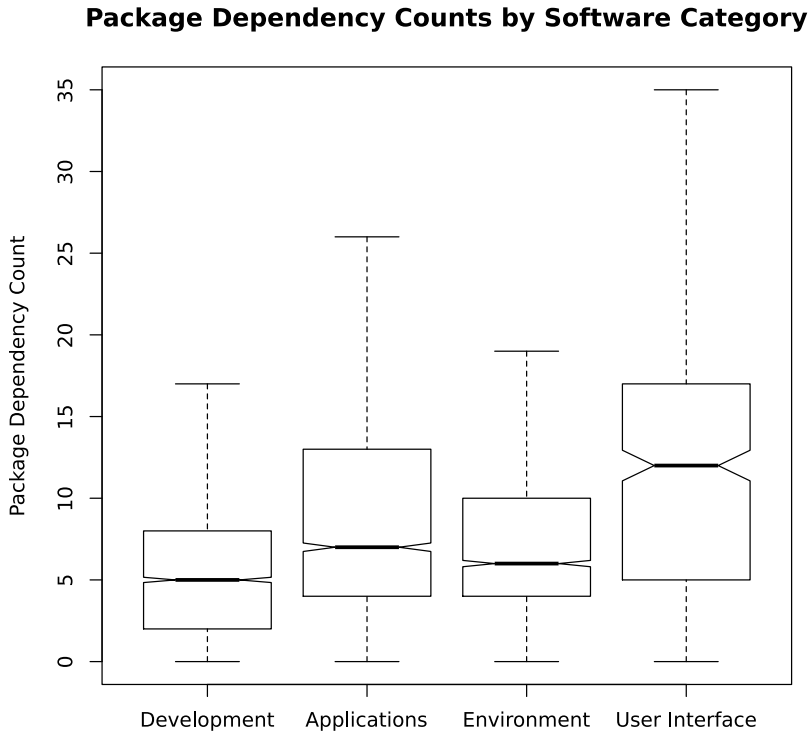

Figure 4.9: Comparison of package dependency count by software category

As none of the notches overlap, the graph shows that there is a statistically significant difference between the median dependency counts of all four groups. This graph raises some interesting questions. Why is software from one category likely to reuse more (or less) packages than that from a different category? One explanation is that software in that category is more complex and most incorporate more pre-existing components in order to fulfill its requirements.

4.2.3 Software Development Tools

Packages in the development category include software libraries and programming language tools that are intended for use by programmers for creating new software. Package dependency counts for the packages in this group follow the same basic trend as packages

in the other groups. When graphed, this data shows the reverse-J shape, with frequency decreasing as reuse increases.

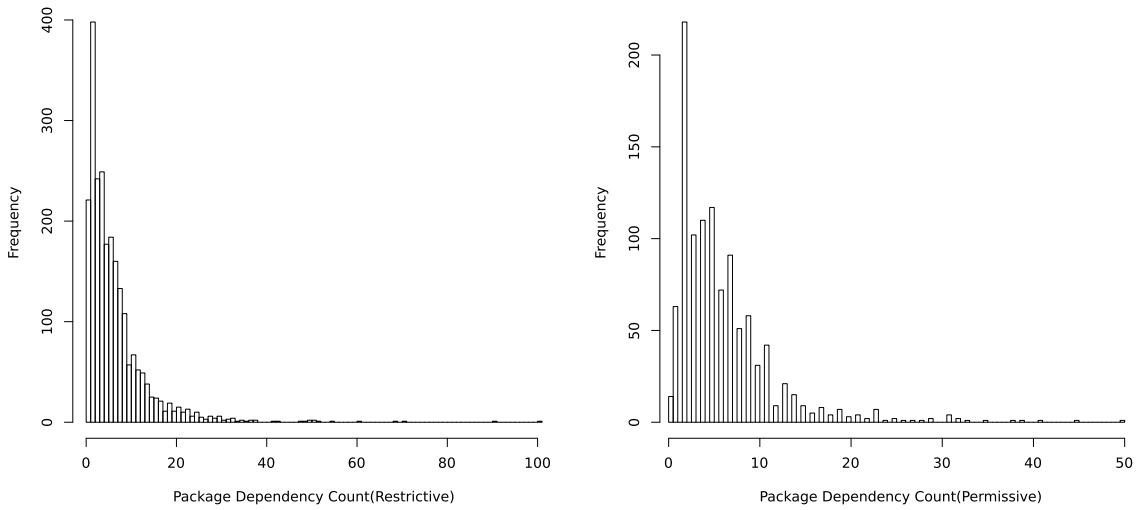

Figure 4.10: Histogram of package dependency counts of software development tools by license type

Software development tools in the permissive group do not fit this trend as closely as the restrictive group. The restrictive group contains a little over twice the number of packages that the permissive group has. However, there does not appear to be a large difference between the package depenency counts of development tools released under restrictive or permissive licenses.

Restrictive		Permissive	
Range	0 - 101	Range	0 - 50
1st Quartile	2	1st Quartile	2
3rd Quartile	9	3rd Quartile	8
Median	5	Median	5
Mean	6.883	Mean	6.254
SD	7.296	SD	5.707
n	2,356	*n*	1,084

Table 4.8: Summary of package dependency counts of software development tools by license type

While the restrictive group includes a number of outliers that have a dependency count

higher than the maximum value for the permissive group, the values of the mean, median, and 1st and 3rd quartiles are almost identical.

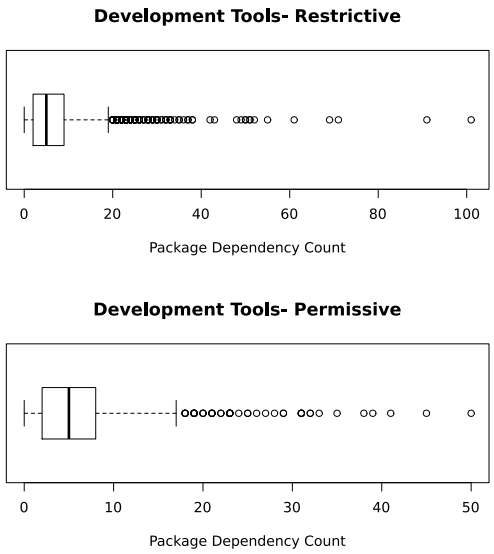

Figure 4.11: Box plots of package dependency counts of software development tools by license type

The software with largest package dependency count in the restrictive group is the aforementioned *Leksah* which has an package depenency count of 101. The leader of the permissive group is the Haskell Platform[2], the set of libraries needed to write Haskell applications. It is interesting the package dependency count leaders for both groups would include development tools for the same programming language. Perhaps the developers of Haskell are more successful at reusing code than those of other languages.

Despite the presence of these outliers, performing the notched box plot test confirms that there is no significant difference between the medians of both groups.

[2]http://hackage.haskell.org/platform/

Figure 4.12: Comparison of median package dependency count of software development tools by license type

Hypothesis 2 predicted that development tools released under a permissive category would have a higher median dependency count than those released under a restrictive license, but this is not the case. The notches for both groups overlap, and the null hypothesis is not rejected.

4.2.4 Applications

The applications category includes software targeted at end-users. At 87%, this category contains the highest percentage of packages released under a restrictive license. This is consistent with observations made by previous studies that predict use of restrictive licenses by open source products that want to increase the size of their user base.

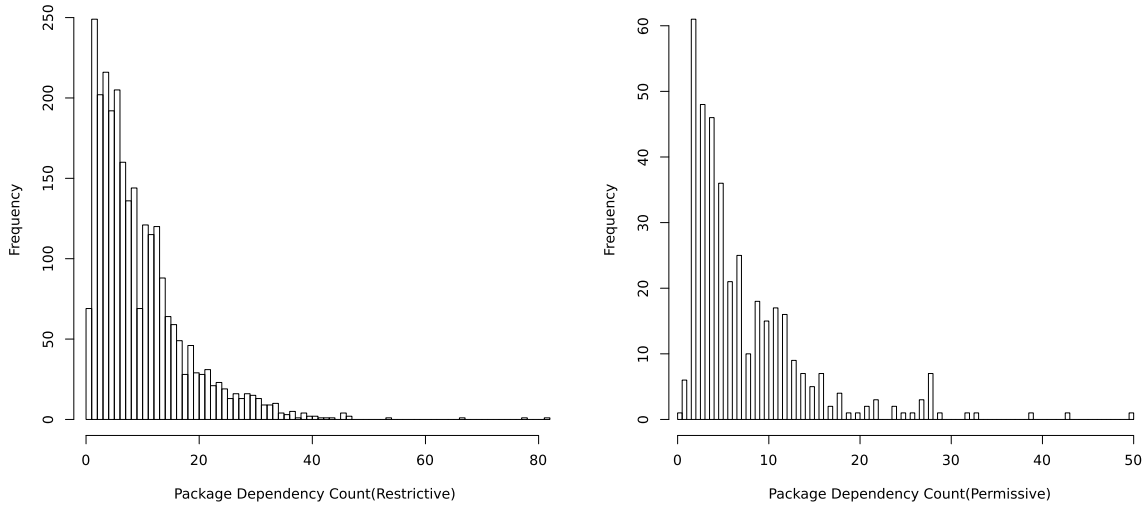

Figure 4.13: Histogram of package dependency counts of applications by license type

Both groups file the now familiar trend of a decrease in frequency as dependency count increases, though the permissive group has a less than perfect fit to this trend. Perhaps this is a function of its smaller sample size; the restrictive group has almost 7 times as many members.

Restrictive		Permissive	
Range	0 - 82	Range	0 - 50
1st Quartile	4	1st Quartile	3
3rd Quartile	13	3rd Quartile	10
Median	8	Median	5
Mean	9.97	Mean	7.782
SD	8.142	SD	7.055
n	2,631	*n*	381

Table 4.9: Summary of the package dependency counts of applications by license type

The restrictive group shows a higher package dependency count for all of the descriptive statistics. The restrictive group also has more outliers than the permissive group, but this might be due to the difference in sample sizes, and not an indication of an actual difference between the two groups.

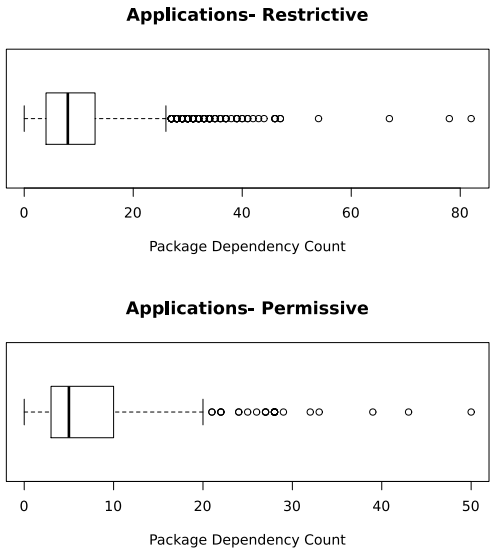

Figure 4.14: Box plots of package dependency counts of applications by license type

The leader of the restrictive group is *Anaconda*[3], an application used to manage the installation of Fedora systems. For the permissive group, *Condor* [4], a workload management application for computationally intensive jobs, leads with an package dependency count of 50.

Comparing the notched box plots of the two groups shows that there is a difference in the two medians.

[3]http://fedoraproject.org/wiki/Anaconda
[4]http://www.cs.wisc.edu/condor/

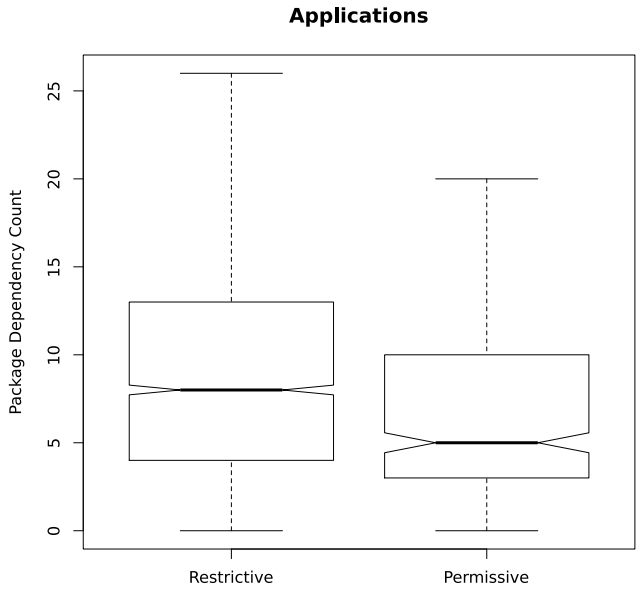

Figure 4.15: Comparison of median package dependency count of applications by license type

Hypothesis 3 predicted that the median package dependency count for applications released under a restrictive license would be greater than those released under a permissive license. This is confirmed by the graph, which shows no overlap between the notches surrounding each median. There is a statistically significant difference between the restrictive group's median of 8, and the permissive group's median of 5, allowing us to reject the null hypothesis.

4.2.5 System Environment Packages

Packages in the system environment category provide utilities that are required to run other software. Applications and other software use these packages to provide specific functionality and integration with the system.

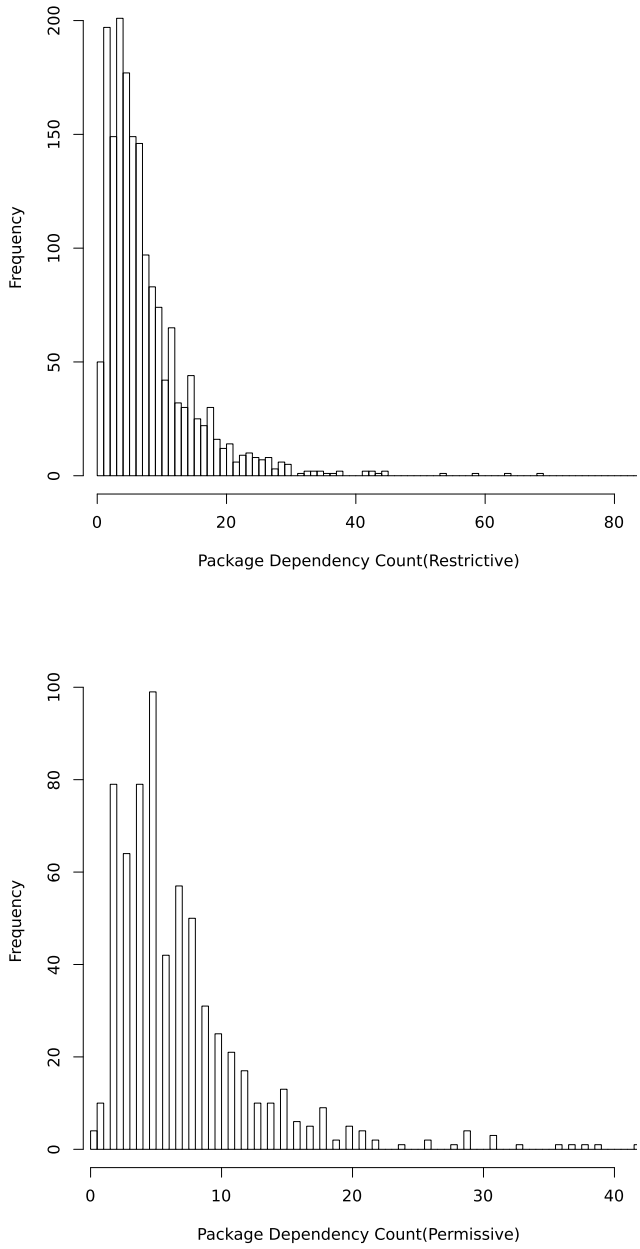

Figure 4.16: Histogram of package dependency counts of system environment components by license type

As with the other software categories, the restrictive group retains the reverse-J shape showing that frequency decreases as package dependency count increases. The permis-

sive group, which has a few peaks, does not fit the trend as well.

Restrictive		Permissive	
Range	0 - 86	Range	0 - 47
1st Quartile	4	1st Quartile	4
3rd Quartile	10	3rd Quartile	9
Median	6	Median	5
Mean	8.187	Mean	7.319
SD	7.240	SD	6.059
n	1,740	*n*	662

Table 4.10: Summary of package dependency counts of system environment components by license type

The difference in package dependency counts between the restrictive and permissive groups is small in comparison to the difference between the restrictive and permissive groups of the application category. The ranges, median, and quartile values are closer to those of the development tools category, which may be explained by the similarity between the two package categories: each is used to create or provide functionality to other software.

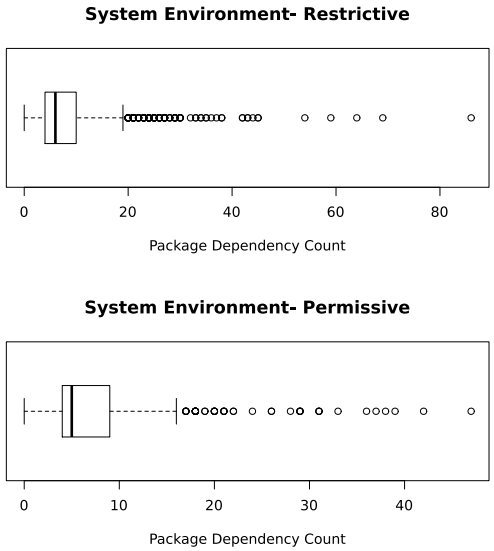

Figure 4.17: Box plots of package dependency counts of system environment components by license type

Outliers in the restrictive group include *ghc-leksah* and *ghc-pandoc,* which provide shared

libraries needed for Haskell development. *NorduGrid*[5], which provides resources to manage distributed computing for physics research, is the leader of the permissive group.

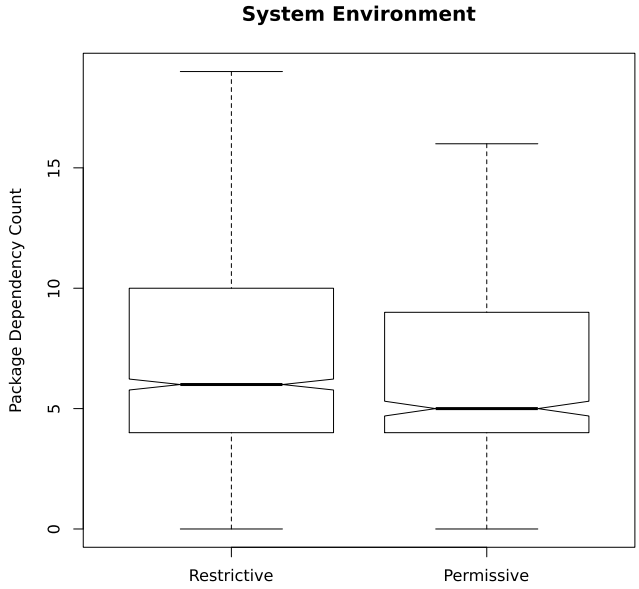

Figure 4.18: Comparison of median package dependency counts of system environment packages by license type

Applying the notched box plot comparison in figure 4.18 of the two shows a difference in their medians. The notches do not over lap, demonstrating a statistically significant difference between the median package dependency count of each group. The restrictive group has a greater median package dependency count than the permissive group, confirming the prediction made by hypothesis 4.

4.2.6 User Interface

User interface packages provide applications and other software with display rendering and input capturing functionality, and help provide a consistent look and feel to the user. This software category has the fewest members, perhaps due to the complexity associated with creating feature-rich graphical user interfaces.

[5]http://www.nordugrid.org/

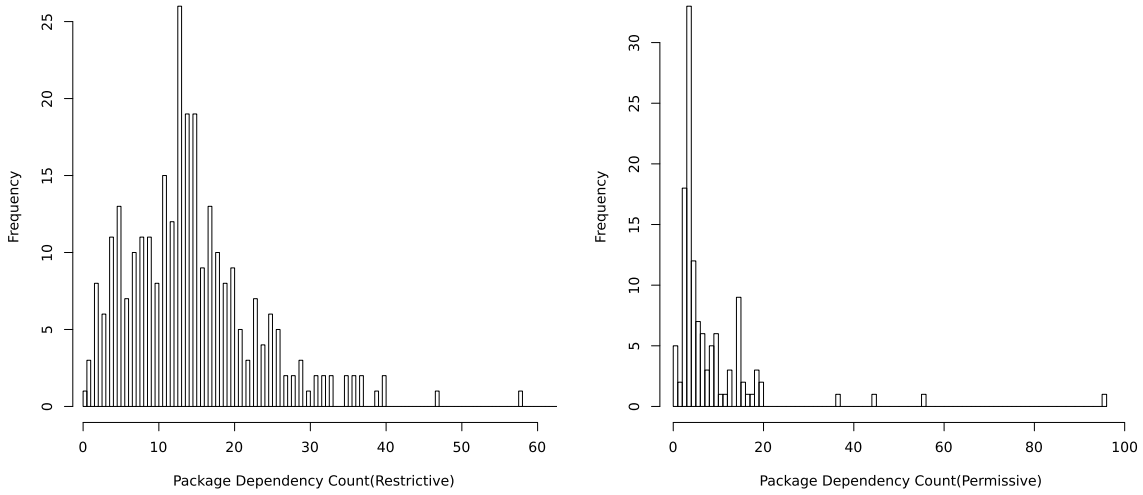

Figure 4.19: Histograms of package dependency counts of user interface packages by license type

The histogram shows that this category differs the most from the trend of frequency decreasing as package dependency counts increase. The restrictive group is closer to a normal distribution than any of the other package dependency count data sets. Instead of peaking at a frequency of 0 or 1, the graph shows a peak of 14, which is the same as that group's median. Why would this group differ so much from the rest? One explanation is the importance of providing a consistent look and feel to an application for a given platform. Software from the user interface category must reuse a relatively small set of existing components to ensure this consistency. That this trend is not followed by the permissive group lends strength to the prediction that restrictively licensed software is more effective at promoting external reuse.

Restrictive		Permissive	
Range	0 - 65	Range	0 - 96
1st Quartile	9	1st Quartile	4
3rd Quartile	19	3rd Quartile	10
Median	14	Median	5
Mean	14.85	Mean	8.532
SD	9.25	SD	11.140
n	285	*n*	124

Table 4.11: Summary of package dependency counts of user interface packages by license type

This is much greater than the permissive group's median package dependency count of 5; in fact, the difference between the two medians is the greatest of all the package dependency count data sets in the study.

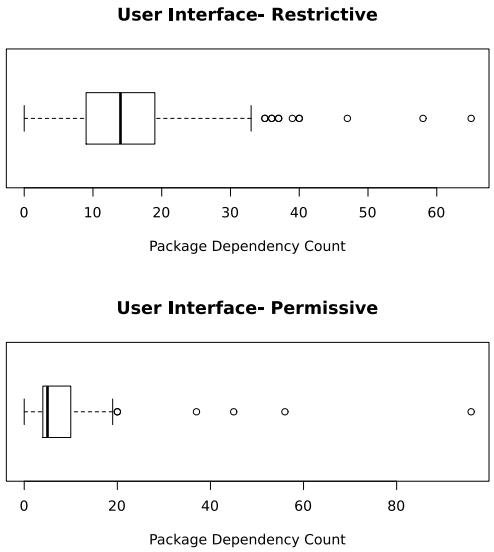

Figure 4.20: Box plots of package dependency counts of user interface packages by license type

The relatively small size of the user interface category and its frequency distribution result in fewer outliers. The package dependency count leader of the restrictive group is *KDE*[6], a popular desktop environment for Linux. The permissive group includes *Bluetile*[7], a window manager for the Gnome desktop (an alternative to KDE), which leads the user interface category with an package dependency count of 96.

[6]http://www.kde.org/
[7]http://www.bluetile.org/

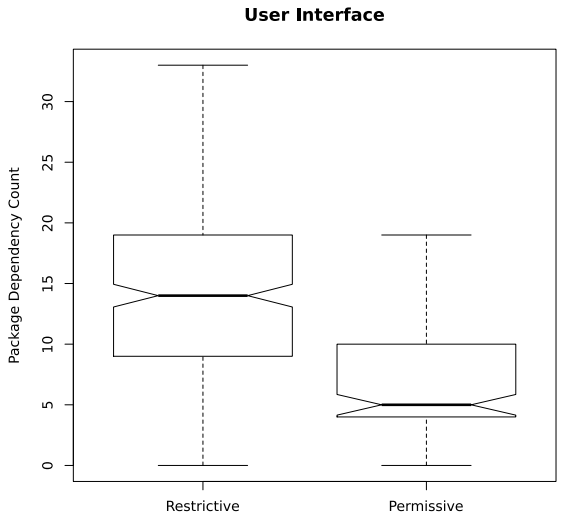

Figure 4.21: Comparison of median package dependency count of user interface packages by license type

Hypothesis 5 predicts that the difference in median is statistically important. Using the notched box plot to compare the two medians results in support for this prediction.

4.3 Dependent Package Count

Next we turn our attention to examining dependent package count by license type and software category. These cases are considered in the same order as in the section on package dependency count.

4.3.1 Restrictive vs. Permissive

The dependent package count for packages released under restrictive and permissive licenses follow the same positively skewed, reverse-J shaped trend in frequency distribution as the sample of all software packages in the repository.

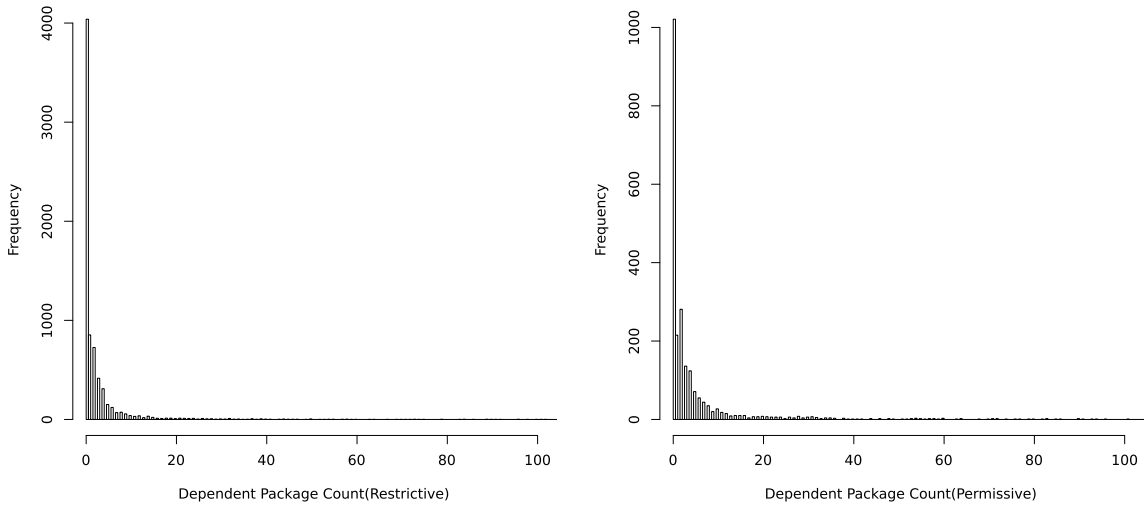

Figure 4.22: Histograms of dependent package counts of all packages by license type

That is, the majority of both sets have dependent package counts clustered at the bottom end of the range, with frequency falling off sharply as dependent package count increases.

Restrictive		Permissive	
Range	0 - 1850	Range	0 - 1250
1st Quartile	0	1st Quartile	0
3rd Quartile	2	3rd Quartile	4
Median	0	Median	1
Mean	4.4	Mean	9.726
SD	42.622	SD	53.81
n	7,263	*n*	2,370
Packages with a count of zero	4,039	Packages with a count of zero	1,021

Table 4.12: Summary of dependent package counts of all packages by license type

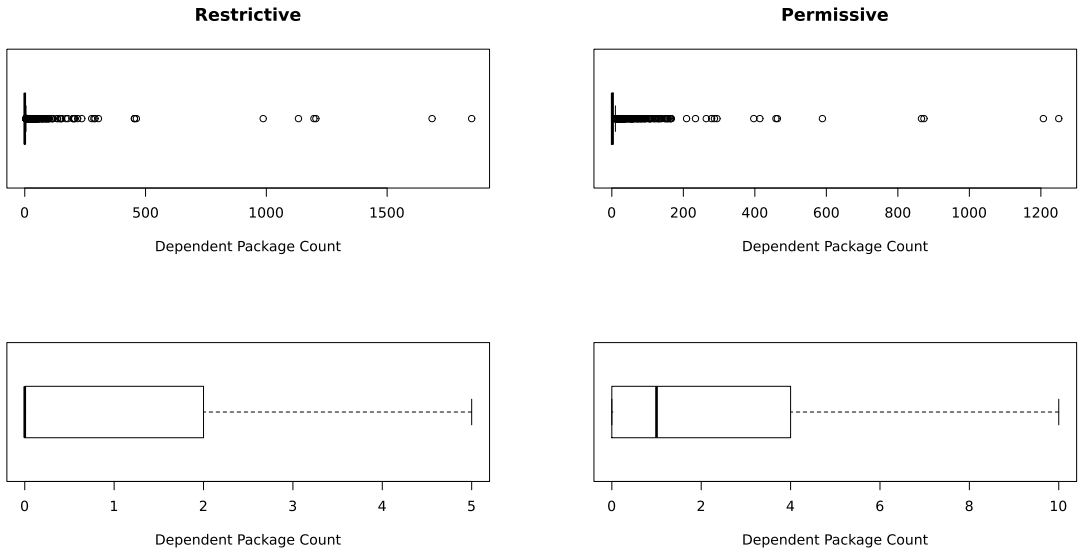

Figure 4.23: Box plots of dependent package counts of all packages by license type

The permissive group does have a higher median dependent package count. Comparing the notched box plots of both sets confirms that this difference is statistically significant.

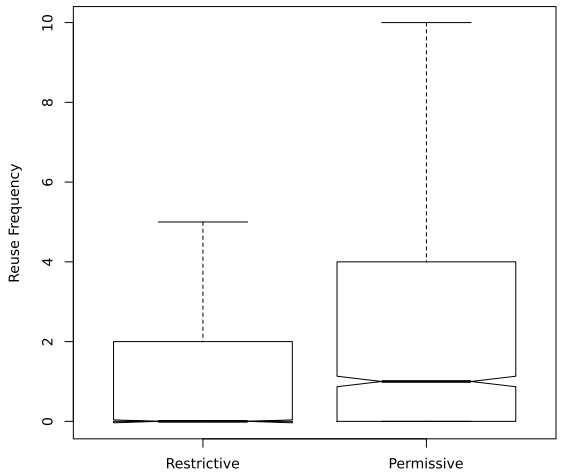

Figure 4.24: Comparison of median dependent package count of all packages by license type

Hypothesis 6 predicted that packages released under a permissive license would be reused

more often than those released under a restrictive one because they have fewer restrictions placed on how they can be incorporated into other works. This graph supports this hypothesis; however, the difference between a median of 0 and a median of 1 is not all that big. Whether this difference between the two medians is important will be discussed in the next section.

4.3.2 Dependent Package Count by Software Category

As was done in the section on package dependency count, the first analysis of dependent package count by license and software category is a comparison of dependent package count by category regardless of license type.

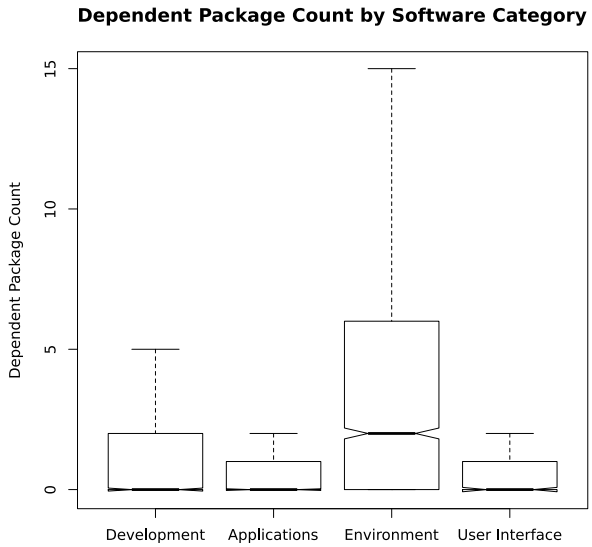

Figure 4.25: Comparison of dependent package count by software category

The results of this comparison are very different from that of package dependency count across all categories. The package dependency count comparison showed that there was a statistical difference between all four groups; that is, package dependency count for each category of software package is statistically different from that of the other three. This is not the case here. The development tools, applications, and user interface groups all have a median dependent package count of zero. Only the system environment category shows a positive median. In addition, these packages include the bulk of the packages that have a dependent package count in the 4th quartile of the set of all packages. That this group has the highest value is not surprising, as it consists of packages that provide

essential functions to other packages. In other words, the system environment packages are specifically designed with reuse in mind. That the development category contains the second highest number of highly reused packages is also not surprising, as development tools are intended to be used to build other applications.

That the applications and user interface categories have almost identical graphs is unexpected. Applications are typically finished products, and while it is possible to use a package in this group as part of a new application, this is not likely to be the typical case. User interface packages, on the other hand, provide functionality that can be shared by other programs, so one might expect their dependent package count to be higher than packages in the applications category. An examination of the packages that make up these two categories would be needed to determine why this is not the case.

4.3.3 Software Development Tools

The frequency distribution of the dependent package count of development tools for each license type are similar to the distribution for all categories by license type. The restrictive group contains more than twice as many packages as the permissive group, but both continue the trend of a decrease in frequency as dependent package count increases.

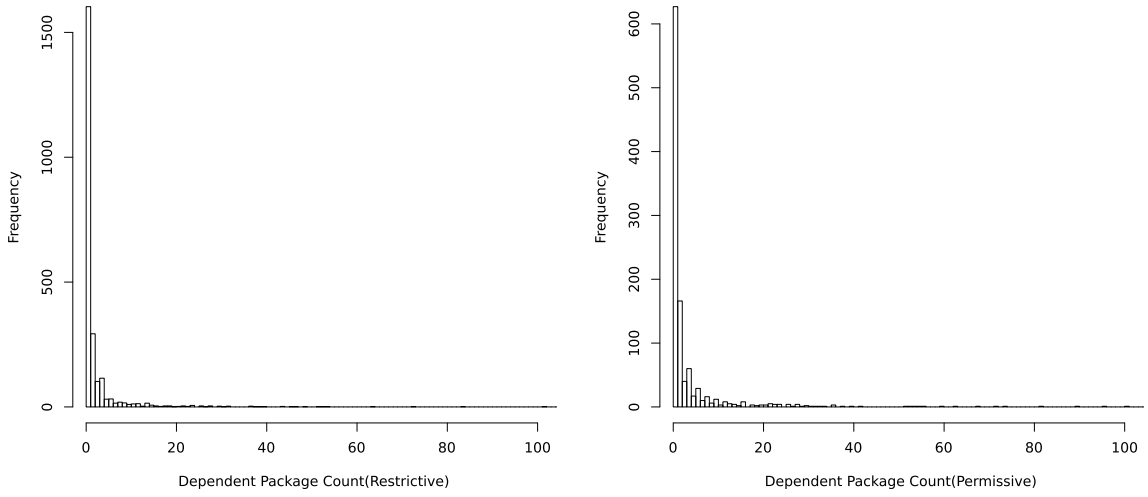

Figure 4.26: Histograms of dependent package counts of software development tools by license type

As is the case for the dependent package count of the software development category as a

whole, both groups have a median dependent count of zero. The rest of their descriptive statistics are also very similar.

Restrictive		Permissive	
Range	0 - 1686	Range	0 - 866
1st Quartile	0	1st Quartile	0
3rd Quartile	2	3rd Quartile	3
Median	0	Median	0
Mean	3.669	Mean	6.124
SD	39.27	SD	33.03
n	2,356	*n*	1,084
Packages with a count of zero	1,378	Packages with a count of zero	547

Table 4.13: Summary of dependent package counts of software development tools by license type

A noticeable exception to this is the range; the restrictive group has a maximum value of almost twice that of the permissive group. This is misleading, however, as there is a large gap between the highest and second highest dependent package count for each group. It becomes apparent when the data is viewed on a box plot that the difference between the outliers of both groups is not as drastic as the range implies.

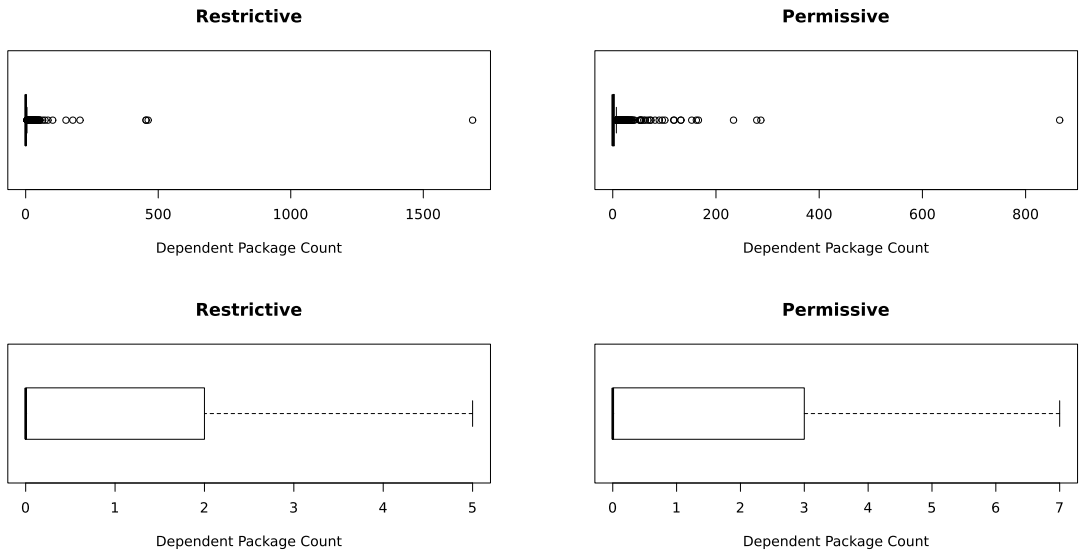

Figure 4.27: Box plots of dependent package counts of software development tools by license type

The extreme outlier in the restrictive group, with a dependent package count of 1,686, is *pkgconfig*[8], a tool used to determine compiler settings and help automate the building of software packages. *libxml2*[9], the leader of the permissive group, provides XML support to applications.

Using a notched box plot to compare the medians shows that there is no statistical difference between the two groups.

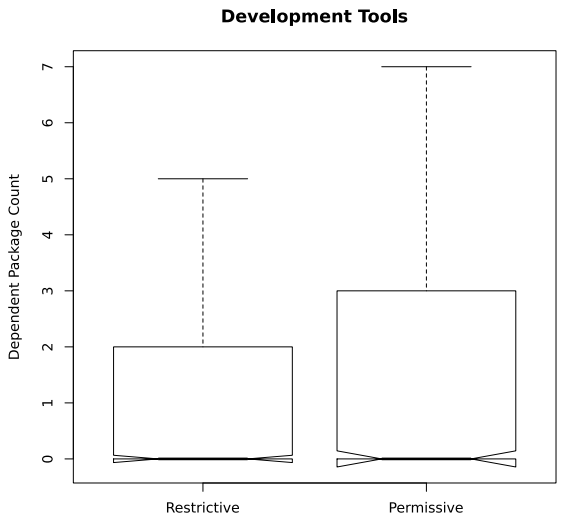

Figure 4.28: Comparison of median dependent package count of software development tools by license type

Hypothesis 7 predicted that the development tools released under a permissive license would have a higher median dependent package count, but there is no evidence to support this. The permissive group has a higher value for its 4th quartile and upper extreme, which hints at a trend of a greater dependent package count for this category, but this is not enough to influence the result of the comparison.

4.3.4 Applications

As pointed out in the section on package dependency count of the applications category, this group have the highest percentage of packages released under a restrictive license.

[8]http://pkgconfig.freedesktop.org
[9]http://xmlsoft.org/

This does not appear to have any effect on the frequency distribution of dependent package count of restrictive and permissive packages within the group.

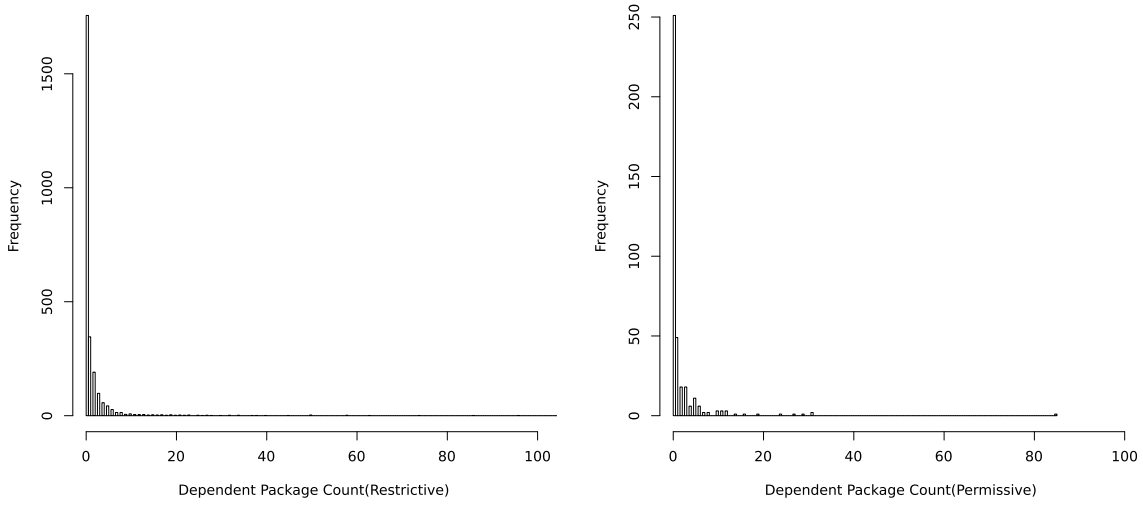

Figure 4.29: Histograms of dependent package counts of applications by license type

These packages also continue the trend of a decrease in frequency as dependent package count increases. As with the dependent package counts of all applications, the restrictive and permissive groups both have a median dependent package count of 0. The rest of the descriptive statistics of the two groups are very similar.

Restrictive		Permissive	
Range	0 - 136	Range	0 - 85
1st Quartile	0	1st Quartile	0
3rd Quartile	1	3rd Quartile	1
Median	0	Median	0
Mean	1.523	Mean	1.73
SD	6.19	SD	5.88
n	2,631	*n*	381
Packages with a count of zero	1,756	Packages with a count of zero	251

Table 4.14: Summary of dependent package counts of applications by license type

The restrictive group contains more outliers and packages with a dependent package count greater than 25.

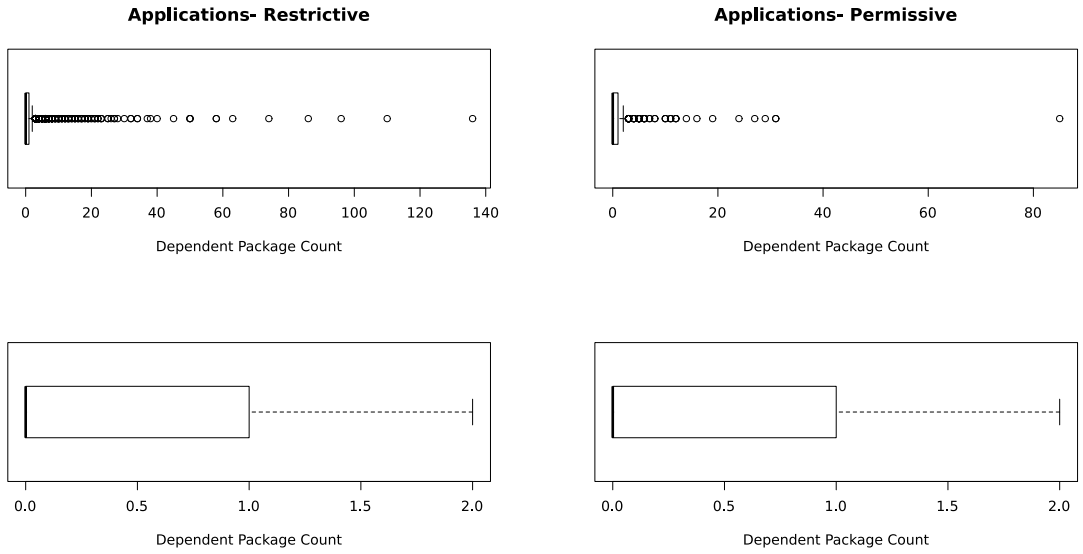

Figure 4.30: Box plots of dependent package counts of applications by license type

The package with the highest dependent package count in the applications category is *GStreamer*[10], a multimedia framework for playing and recording sound and video. This package is released under a restrictive license, and has a much greater dependent package count than the leader of the permissive group, *root-cint*[11], a C++ interpreter. This suggests that the packages from the restrictive group might be reused more often, a prediction made in hypothesis 8. However, using a notched box plot to compare the median dependent package count shows that there is no statistical difference between the medians of the two groups.

[10]http://gstreamer.freedesktop.org/
[11]http://root.cern.ch/

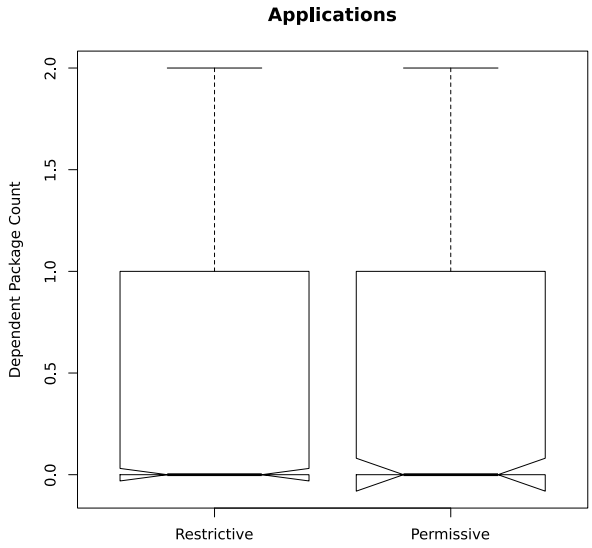

Figure 4.31: Comparison of median dependent package count of applications by license type

As noted earlier, packages in the applications category typically help users solve a particular problem, and are not necessarily intended to be used as components of other packages. This suggests that they are not designed with reuse as the number one goal, making it less surprising that there is no difference in the dependent package counts of packages released under different license types.

4.3.5 System Environment Packages

The system environment packages provide a contrast to the dependent package counts of the development tools and applications packages just discussed. These packages are designed to provide functionality to other packages so they are expected to show a greater dependent package count than packages in the other categories. Like packages in most of the other categories, their dependent package count shows the reveres-J shape denoting a decrease as dependent package count increases.

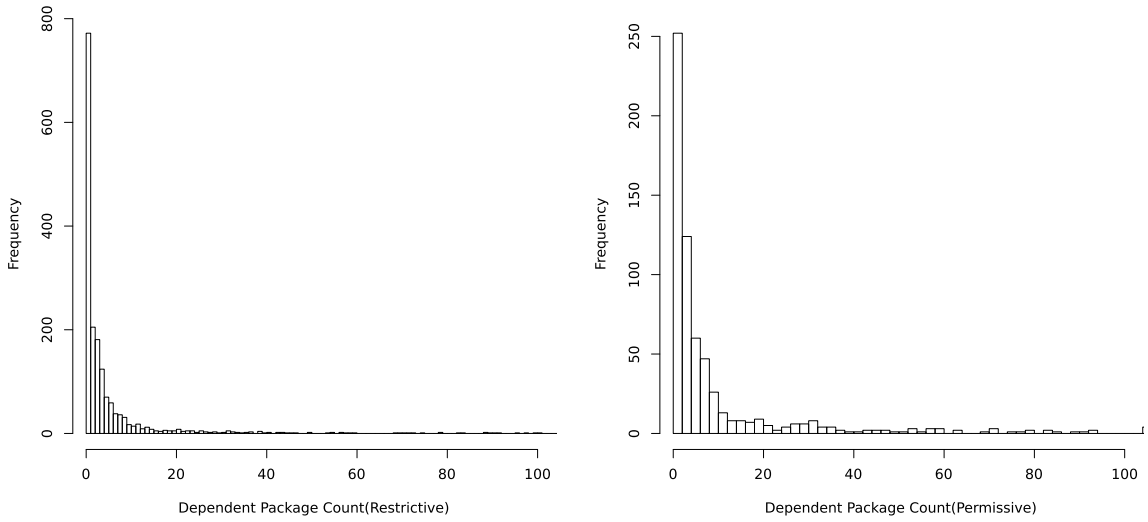

Figure 4.32: Histograms of dependent package counts of system environment packages by license type

Their differences with other categories become apparent when reviewing their descriptive statistics. The system environment category contains the most reused packages. 7 packages in the sample have a dependent package count greater than 1,000, and 6 of them belong to this category. System environment packages also have the highest mean and median dependent package counts, and the lowest percentage of packages that are not reused at all.

Restrictive		Permissive	
Range	0 - 1850	Range	0 - 1250
1st Quartile	0	1st Quartile	2
3rd Quartile	5	3rd Quartile	9
Median	2	Median	4
Mean	10.66	Mean	21.11
SD	73.34	SD	83.39
n	1,740	*n*	662
Packages with a count of zero	568	Packages with a count of zero	125

Table 4.15: Summary of dependent package counts of system environment packages by license type

Like all of the other data sets, this category has a lot of outliers with the range of values in the 4th quartile much larger than the range of the first 3 quartiles.

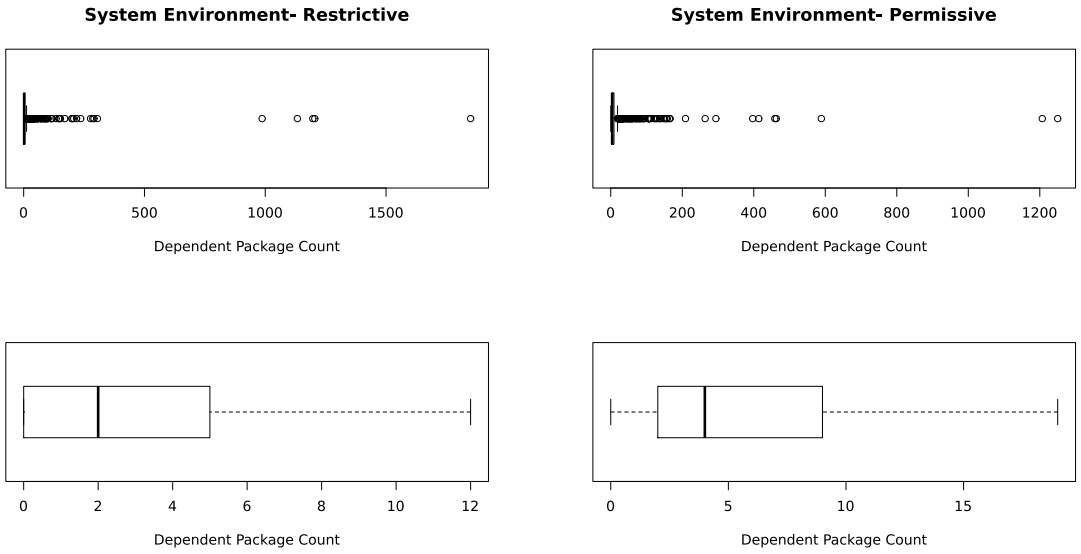

Figure 4.33: Box plots of dependent package counts of system environment packages by license type

The system environment category is home to *glibc*[12], the single most reused package in the repository. As the implementation of the standard C library for a platform written in C, this library is reused extensively. The leader of the permissive group is *libX11*[13], the standard library on Linux systems to interact with user displays and render graphics and read mouse and keyboard input.

Comparing the notched box plots for the restrictive and permissive subsets of system environment packages shows a significant difference between the two groups.

[12]http://www.gnu.org/software/glibc/
[13]http://www.x.org

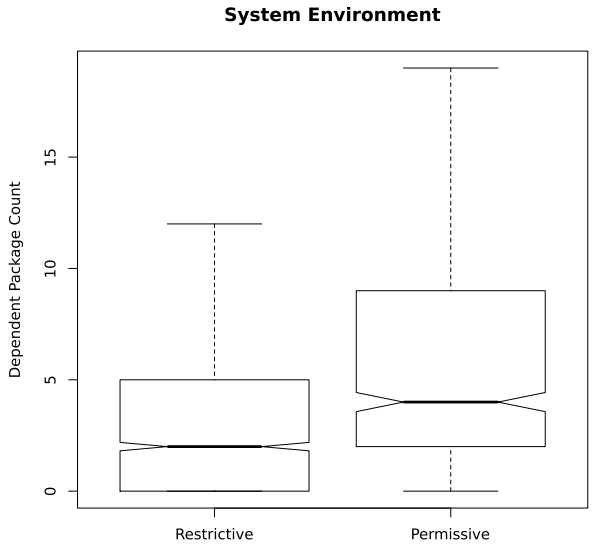

Figure 4.34: Comparison of median dependent package count of system environment packages by license type

Packages released under a permissive license have fewer rules on how they can be incorporated into other projects, so hypothesis 9 predicted that this group would show a greater dependent package count than the restrictive group. The graph shows that there is evidence to support this claim.

4.3.6 User Interface

The final subset examined in this section contains the dependent package counts of packages in the user interface category. Most of these packages provide a functionality to render graphics, such as the various open source windowing system used to create consistent desktop appearance and behavior. Many of the packages are intended for use within other programs; others include standalone applications that provide enhanced alternatives to other applications.

Packages in this group exhibit the expected trend of a decrease in frequency as dependent package count increases.

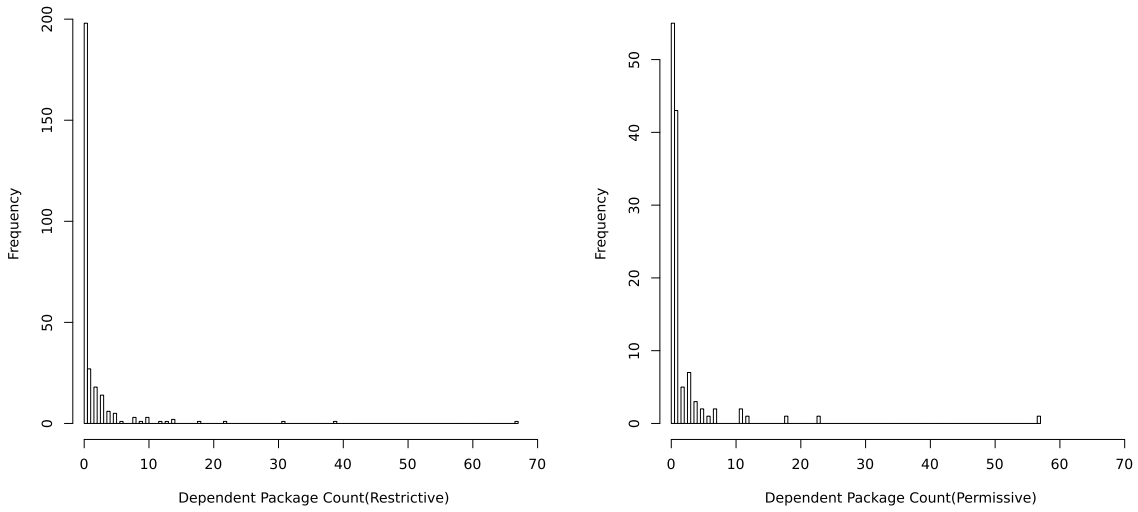

Figure 4.35: Histograms of dependent package counts of user interface packages by license type

The range of values for user interface packages is the smallest of the four categories; each of the other categories contain packages that are reused more frequently. This is surprising given the expectation that user interface components be used in other applications, but it may be explained by the nature of graphical interfaces used by Linux. Unlike other operating systems created by Microsoft and Apple, there is no single common desktop environment for Linux. Instead, users can choose between several environments with different approaches on how a system should look and feel. Many applications include a user interface optimized for one of these environments, distributing the dependent package counts across multiple packages and preventing the counts for any one of them form becoming very large.

Restrictive		Permissive	
Range	0 - 67	Range	0 - 57
1st Quartile	0	1st Quartile	0
3rd Quartile	1	3rd Quartile	2
Median	0	Median	1
Mean	1.589	Mean	2
SD	5.57	SD	5.95
n	285	*n*	124
Packages with a count of zero	198	Packages with a count of zero	55

Table 4.16: Summary of dependent package counts of user interface package by license type

The user interface category also has the fewest number of packages, which translates to fewer outliers, though like all of the groups in this study the range of the outliers exceeds the range of the first quartile.

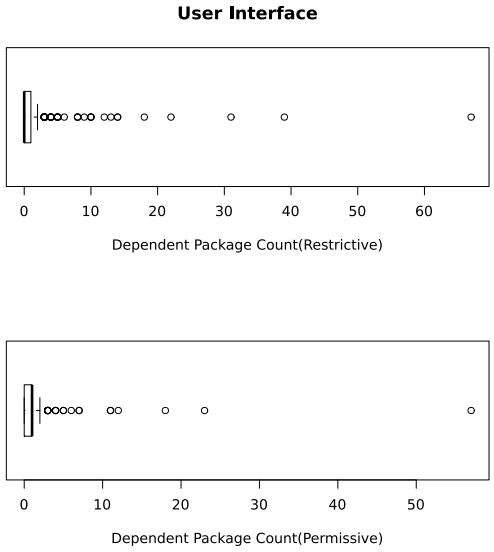

Figure 4.36: Box plots of dependent package counts of user interface packages by license type

The packages with the highest dependent package counts are the core applications for KDE (restrictive) and X11 (permissive), extensions of the desktop environment and graphics display systems mentioned above.

The user interface category contains a higher percentage of packages with a dependent

package count of zero than the collection of all packages regardless of category, which keeps the median of both groups fairly low. However, there is a statistically significant difference between the two medians.

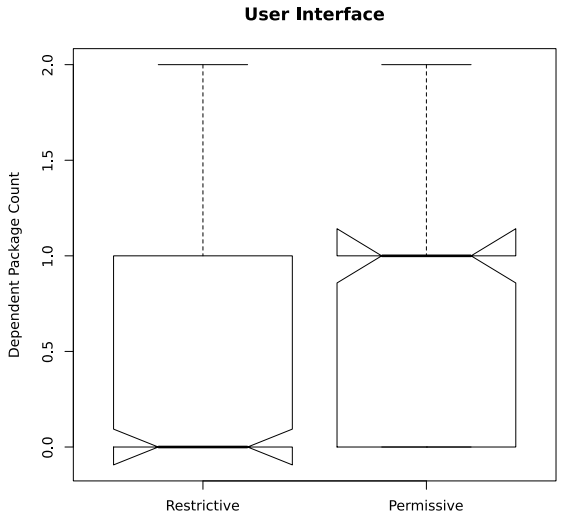

Figure 4.37: Comparison of median dependent package count of user interface packages by license type

Hypothesis 10 predicts user interface packages released under a permissive license to have a higher median dependent package count than those released under a restrictive license, and a comparison of the medians using notched box plots supports this. Like other cases where there is a difference in medians, the difference is not very great. This is discussed further in the next section.

Chapter 5

Conclusions

5.1 Hypothesis Review

5.1.1 Package Dependency Count

The following table summarizes the results presented in the previous section.

Package Dependency Count Hypotheses Summary of Results	
Hypothesis	Supported?
H1 All Packages: Restrictive > Permissive	Yes
H2 Development Tools: Restrictive > Permissive	No
H3 Applications: Restrictive > Permissive	Yes
H4 System Environment: Restrictive > Permissive	Yes
H5 User Interface: Restrictive > Permissive	Yes

Table 5.1: Package dependency count hypotheses summary

The general prediction that packages released under a restrictive license will show greater package dependency counts than those released under a permissive license holds is supported by the analysis. One possible explanation for this put forth earlier is that the choice of a restrictive license shows that a software project adheres to certain principles, including the right for anyone to use existing software as they see fit and the belief that reinventing an existing solution is not an optimal use of resources. These principles might indicate a greater propensity toward finding existing software components to reuse than that of developers preferring a permissive license.

For one software category this does not seem to be the case. An analysis of the data

did not support hypotheses 2, which predicted greater package dependency counts for software development tools. This is a surprising result. The GNU project, which played an important role at beginning of the open source movement, began its work by creating development tools that could be used to create other software. These tools were released under the most restrictive licensing terms. It was expected that this would lead to a greater package dependency counts among packages in the software development category. However, the median package dependency count of both license types in this category is the same. Programmers who write compilers, interpreters, debuggers, and other tools for programmers apparently have views on reusing existing components that is not affected by their preference for a certain license.

While there are statistically significant differences between the median package dependency count by license type, perhaps the more important question is whether these differences are important. For two of the groups, all packages and system environment packages, the median package dependency count of restrictively licensed software is 6, and permissively licensed is 5. This is not a very big difference; in fact, if the difference were any smaller, there would be no difference at all. The difference in medians for these two groups it does not seem to be important. For all intents and purposes, license type may not be an accurate predictor of how many packages are reused by packages within these two groups.

For some categories, the difference is much greater. Applications released under a restrictive license have a median package dependency count of 8, as compared to a median of 5 for their permissively licensed counterparts. The difference for the user interface category is the greatest, with medians of 14 for the restrictive group and 5 for the permissive group. For these two categories, choice of license type plays appears to play an important role in affecting a project's inclination to reuse existing components.

What is it about packages in these two categories that cause them to show a greater differences in package dependency counts than packages in the other two software categories? Software in these categories may lend themselves to higher package dependency counts because they are more likely to provide a range of functionality that can be implemented by multiple components. A word processor application, for example, might have a need for components that provide lexical analysis, reading and writing data to a file, graphical displays of text, and many others, whereas a system environment component (such as the file i/o component used by the word processor) does not need to reuse as many. This certainly explains why these categories show package dependency counts. That the software in these categories released under a restrictive license reuse the features of more pacakges supports the idea that developers preferring restrictive licenses place a greater value on the importance of reusing existing components.

5.1.2 Dependent Package Count

The results of the analysis of dependent package count hypotheses is summarized in the following table.

Dependent Package Count Hypotheses Summary of Results	
Hypothesis	Supported?
H6 All Packages: Restrictive < Permissive	Yes
H7 Development Tools: Restrictive < Permissive	No
H8 Applications: Restrictive > Permissive	No
H9 System Environment: Restrictive < Permissive	Yes
H10 User Interface: Restrictive < Permissive	Yes

Table 5.2: External reuse hypotheses summary

Packages released under a permissive license are expected to be reused more often than their restrictively licensed counterparts because they have less restrictions on how they can be used. The analysis of the dependent package counts of all packages in the sample supports this prediction. There is a statistically significant difference between the median dependent package count of packages released under a permissive license and those released under a restrictive one. However, this difference is not very large. Software in the permissive group have a median dependent package count of 1, and those in the restrictive group have a median of 0. The median dependent package count for software packages in the groups analyzed in this study are rather small, primarily due to the large number of packages that are not reused at all. The package dependency count statistics in this study come from the reuse of only 47% (4,510 out of 9,570) of the packages in the sample. With over half of the packages having a dependent package count of 0, the median dependent package count of each group is kept pretty low.

The biggest difference is found in the system environment category. Packages in this category that are released under a permissive license have a median dependent package count twice that of those released under a restrictive license (4 vs. 2). A similar situation exists for user interface packages, where the median dependent package count of the permissive group is 1, and that of the permissive group is 0. In addition to the general disposition towards higher dependent package counts for packages released under a permissive license, software in these categories are explicitly intended to be reused. The result is that components from these two categories that are released under a permissive license are more likely to be used to create other software.

No statistically significant difference was found between the median dependent package count of the permissive and restrictive groups of packages in the software development tools and applications categories. The hypotheses that predicted there would be a signif-

icant difference did not take into account the impact of packages in these sets that were not reused at all. Both categories have a median reuse of zero. That applications are not reused all that often does make sense when you consider that they are often built by reusing components from other categories are not necessarily intended to be reused themselves. Software development tools, on the other hand, are used to create new programs, and were therefore expected to be reused. It turns out that software development tools may be used to create software, but do not become part of the new software and are not reused directly.

5.1.3 Summary

The fundamental question considered in this study is whether or not license type affects software reuse. Our conclusion is that yes, choice of license does play a part in predicting reuse patterns in open source development. Open source software released under a restrictive license is more likely to reuse external components than those released under permissive licenses. This is true for all software, with one noticeable exception. The number of components reused by software intended to help programmers write new programs is not affected by license choice.

In addition, packages that are released under a permissive license are more likely to be reused by another program than those released under a restrictive one. This holds true for all software categories except for development tools and applications. It is interesting to note that comparing the package dependency and dependent packages count statistics presents a paradox of sorts. Packages released under a restrictive license reuse more components than those released under a permissive one. However, the permissive packages are the ones more likely to be reused.

This conclusion helps shed some light on whether there are tangible differences between licenses. That license choice affects reuse supports the claim that license choice matters. The open source movement, to the extent that it can be defined, can use this information to evaluate the merits of various licenses in promoting the open source ideals. These results also make suggestions to open source projects as to which license type they should use for their products. Projects developing components intended for use by other projects may increase the appeal of their products by releasing them under a permissive license. The trade off to this, however, is that in doing so they lose some control over how those components are modified and redistributed. Hopefully the results of this study can help those making the decision of what license type to choose to pick the license best suited for achieving their objectives.

5.2 Future Work

This study attempted to look at the big picture of how differences in software licenses affect reuse. Subsequent studies that examine licenses and software categories in greater detail might be able to provide additional insight into this topic. For instance, do differences between individual licenses, as opposed to license type, affect reuse? This study assumed the answer to this question to be no, but this might not be the case. Not all researchers categorize the LGPL as a restrictive license. Perhaps it would be worth repeating this experiment for a third category to include weak-copyleft licenses such as the LGPL.

Also, how does license choice affect reuse of software in a particular domain? The four categories examined in this paper are fairly broad, and further research can confirm the trends identified within are maintained for groups that are smaller and more rigidly defined. Examining reuse patterns within a domain would also allow for the identification of multiple packages that provide the same functionality to determine whether the lack of available alternatives is an intervening variable. Is a package's dependency or dependent package count as high as it is because it is the only package that provides a certain set of capabilities? Given the open source movement's preference for not solving the same problem twice, the lack of alternatives may prove to be a significant influence on package reuse. *glibc* has the highest dependent package count of all the packages examined in this study, but it is also the only implementation of the C standard library in the Fedora repository.

To what extent do package dependency and dependent package counts vary by programming language? The packages in the Fedora repository are implemented in a wide variety of languages, though the four software categories used in this study do not distinguish between them. Another for future work would be to repeat this analysis on groups of packages broken down by language or language type. This could determine if choice of programming language has a bigger affect on reuse patterns than choice of license type.

This study also used a fairly broad definition of reuse; it took into account *that* a package was reused, but not *how* it was reused. A more specific metric would take into account how much functionality a component provides to the software that reuses it. Future research could then use this metric to refine the understanding of how license choice affects reuse.

As discussed above, the results of this study may well be applicable to packages stored in the repositories of other, non-Fedora, Linux distributions due to similarities in how software is bundled for distribution. Since most Linux distributions provide tools to resolve package dependencies it would not be too difficult to repeat this study using another repository as a data source. This would serve to help verify the generalizability of these results.

Determining the package dependency and dependent package counts for each package makes it possible to identify individual software packages that are particularly good at reusing components or being reused themselves. What is it about the outliers that caused them to have such high package dependency and dependent package counts? An interesting direction for future research would be to perform in depth case studies of these packages to determine why this is so. What development methods do they have in common? What caused the projects that produced these packages to adopt they license they did? To what extent did their license choice reflect their ideas on how software should be reused? Case studies of multiple packages can help identify trends in open source development and provide a greater understanding of how reuse occurs in open source projects.

The methods used in this study can also be used to create a catalog of reusable components ranked by their dependent package count. Finding the right components to reuse can be a challenge, and can lead to increased defect rates and production delays if not done effectively. The catalog could help programmers and software architects identify components in a particular domain that have a history of being reused successfully by other projects. By examining these other projects, developers can get a sense of how the component is being used, and be in a better position to determine if it applies to their design. In the end, this can only help the open source development community continue to produce new and innovative software.

Chapter 6

Appendix

Fedora repositories store package information in multiple SQLite databases. SQLite is a relational database management system that stores a database in a single file. Fedora repositories have a subdirectory named *repodata* that contain the databases for that repository. These databases contain descriptive information for each package in the repository, as well as the dependency relationships between them. The primary database, which appears in a compressed file named primary.sqlite.bz2, provided the data for this study. A copy of this database was downloaded on March 3, 2012, from http://dl.fedoraproject.org/pub/fedora/linux/releases/16/Everything/x86_64/os/repodata/. A complete list of sites hosting mirrors of the Fedora repository, for all versions, can be found at http://mirrors.fedoraproject.org/publiclist/.

This appendix lists the queries used to generate the package dependency and dependent package counts analyzed above.

6.1 License Type Tables

To facilitate the grouping of packages according to license type, two tables were added to the local copy of the primary database. These tables, named *restrictive* and *permissive*, contained the list of all the licenses used by packages within the repository that belong to either of the two license types. All subsequent queries joined against the two license tables, which simplified the task of creating data sets for each license type.

6.1.1 Restrictive License Table Query

The table of restrictive licenses was created with this query:

```
CREATE TABLE restrictive AS
SELECT DISTINCT rpm_license FROM packages
WHERE
  -- GPL variants, excluding dual/multiple licenses
  (rpm_license NOT LIKE '%OR%'  AND rpm_license NOT LIKE '%AND%'
  AND rpm_license like '%GPL%'
  AND rpm_license NOT LIKE '%LGPL%'
  AND rpm_license NOT LIKE '%AGPL%'
  AND rpm_license NOT LIKE '%NGPL%')

  OR

  -- LGPL variants, excluding dual/multiple licenses
  (rpm_license NOT LIKE '%OR%'  AND rpm_license NOT LIKE '%AND%'
  AND rpm_license like '%LGPL%'
  AND rpm_license NOT LIKE '%,%' )
```

6.1.2 Permissive License Table Query

A similar query was used to create the table of permissive licenses:

```
CREATE TABLE restrictive AS
SELECT DISTINCT distinct rpm_license FROM packages
WHERE
  -- MIT license
  (rpm_license NOT LIKE '%OR%'
  AND rpm_license NOT LIKE '%AND%'
  AND rpm_license like '%MIT%'
  AND rpm_license NOT LIKE '%,%')

  OR

  -- BSD License
  (rpm_license NOT LIKE '%OR%'
  AND rpm_license NOT LIKE '%AND%'
  AND rpm_license like '%BSD%'
  AND rpm_license NOT LIKE '%,%')

  OR
  -- Apache License
 (rpm_license NOT LIKE '%OR%'
```

```
  AND rpm_license NOT LIKE '%AND%'
  AND (rpm_license like '%ASL%'  OR rpm_license like '%Apache%')
  AND rpm_license NOT LIKE '%,%' )
```

6.2 Package Dependency Count Queries

The queries in this section were used to collect package dependency counts.

6.2.1 All Packages By License Type

Restrictive

```
SELECT a.name, count(distinct d.pkgKey)
FROM packages a
LEFT OUTER JOIN requires b ON a.pkgKey=b.pkgKey
LEFT OUTER JOIN provides c ON b.name=c.name
LEFT OUTER JOIN packages d ON c.pkgKey=d.pkgKey
JOIN restrictive e ON a.rpm_license=e.rpm_license
WHERE a.arch='x86_64'
GROUP BY a.name
```

Permissive

```
SELECT a.name, count(distinct d.pkgKey)
FROM packages a
LEFT OUTER JOIN requires b ON a.pkgKey=b.pkgKey
LEFT OUTER JOIN provides c ON b.name=c.name
LEFT OUTER JOIN packages d ON c.pkgKey=d.pkgKey
JOIN permissive e ON a.rpm_license=e.rpm_license
WHERE a.arch='x86_64'
GROUP BY a.name
```

6.2.2 By License Type and Software Category

Development Tools- Restrictive

```
SELECT a.name, count(distinct d.pkgKey)
```

```
FROM packages a
LEFT OUTER JOIN requires b ON a.pkgKey=b.pkgKey
LEFT OUTER JOIN provides c ON b.name=c.name
LEFT OUTER JOIN packages d ON c.pkgKey=d.pkgKey
JOIN restrictive e ON a.rpm_license=e.rpm_license
WHERE a.arch='x86_64'
AND a.rpm_group LIKE 'Development%'
GROUP BY a.name
```

Development Tools- Permissive

```
SELECT a.name, count(distinct d.pkgKey)
FROM packages a
LEFT OUTER JOIN requires b ON a.pkgKey=b.pkgKey
LEFT OUTER JOIN provides c ON b.name=c.name
LEFT OUTER JOIN packages d ON c.pkgKey=d.pkgKey
JOIN permissive e ON a.rpm_license=e.rpm_license
WHERE a.arch='x86_64'
AND a.rpm_group LIKE 'Development%'
GROUP BY a.name
```

Applications- Restrictive

```
SELECT a.name, count(distinct d.pkgKey)
FROM packages a
LEFT OUTER JOIN requires b ON a.pkgKey=b.pkgKey
LEFT OUTER JOIN provides c ON b.name=c.name
LEFT OUTER JOIN packages d ON c.pkgKey=d.pkgKey
JOIN restrictive e ON a.rpm_license=e.rpm_license
WHERE a.arch='x86_64'
AND (a.rpm_group LIKE 'ApplicatiONs%' or a.rpm_group LIKE 'Amusements/%')
GROUP BY a.name
```

Applications- Permissive

```
SELECT a.name, count(distinct d.pkgKey)
FROM packages a
LEFT OUTER JOIN requires b ON a.pkgKey=b.pkgKey
LEFT OUTER JOIN provides c ON b.name=c.name
LEFT OUTER JOIN packages d ON c.pkgKey=d.pkgKey
```

```
JOIN permissive e ON a.rpm_license=e.rpm_license
WHERE a.arch='x86_64'
AND (a.rpm_group LIKE 'ApplicatiONs%' or a.rpm_group LIKE 'Amusements/%')
GROUP BY a.name
```

System Environment- Restrictive

```
SELECT a.name, count(distinct d.pkgKey)
FROM packages a
LEFT OUTER JOIN requires b ON a.pkgKey=b.pkgKey
LEFT OUTER JOIN provides c ON b.name=c.name
LEFT OUTER JOIN packages d ON c.pkgKey=d.pkgKey
JOIN restrictive e ON a.rpm_license=e.rpm_license
WHERE a.arch='x86_64'
AND a.rpm_group LIKE 'System Environment%'
GROUP BY a.name
```

System Environment- Permissive

```
SELECT a.name, count(distinct d.pkgKey)
FROM packages a
LEFT OUTER JOIN requires b ON a.pkgKey=b.pkgKey
LEFT OUTER JOIN provides c ON b.name=c.name
LEFT OUTER JOIN packages d ON c.pkgKey=d.pkgKey
JOIN permissive e ON a.rpm_license=e.rpm_license
WHERE a.arch='x86_64'
AND a.rpm_group LIKE 'System Environment%'
GROUP BY a.name
```

User Interface- Restrictive

```
SELECT a.name, count(distinct d.pkgKey)
FROM packages a
LEFT OUTER JOIN requires b ON a.pkgKey=b.pkgKey
LEFT OUTER JOIN provides c ON b.name=c.name
LEFT OUTER JOIN packages d ON c.pkgKey=d.pkgKey
JOIN restrictive e ON a.rpm_license=e.rpm_license
WHERE a.arch='x86_64'
AND a.rpm_group LIKE 'User Interface%'
GROUP BY a.name
```

User Interface- Permissive

```
SELECT a.name, count(distinct d.pkgKey)
FROM packages a
LEFT OUTER JOIN requires b ON a.pkgKey=b.pkgKey
LEFT OUTER JOIN provides c ON b.name=c.name
LEFT OUTER JOIN packages d ON c.pkgKey=d.pkgKey
JOIN permissive e ON a.rpm_license=e.rpm_license
WHERE a.arch='x86_64'
AND a.rpm_group LIKE 'User Interface%'
GROUP BY a.name
```

6.3 Dependent Package Count Queries

The queries in this section were used to extract dependent package counts.

6.3.1 All Packages By License Type

Restrictive

```
SELECT c.name, count(distinct a.pkgKey) AS 'Reuse Count'
FROM packages c
JOIN provides b on b.pkgKey=c.pkgKey
LEFT OUTER JOIN requires a ON a.name=b.name
JOIN restrictive d ON c.rpm_license=d.rpm_license
WHERE c.arch='x86_64'
GROUP BY c.name
```

6.3.2 All Packages By License Type

Permissive

```
SELECT c.name, count(distinct a.pkgKey) AS 'Reuse Count'
FROM packages c
JOIN provides b ON b.pkgKey=c.pkgKey
LEFT OUTER JOIN requires a ON a.name=b.name
JOIN permissive d ON c.rpm_license=d.rpm_license
WHERE c.arch='x86_64'
```

```
GROUP BY c.name
```

6.3.3 By License Type and Software Category

Development Tools- Restrictive

```
SELECT c.name, count(distinct a.pkgKey) as 'Reuse Count'
FROM packages c
JOIN provides b ON b.pkgKey=c.pkgKey
LEFT OUTER JOIN requires a ON a.name=b.name
JOIN restrictive d ON c.rpm_license=d.rpm_license
WHERE c.arch='x86_64'
AND c.rpm_group LIKE 'Development%'
GROUP BY c.name
```

Development Tools- Permissive

```
SELECT c.name, count(distinct a.pkgKey) as 'Reuse Count'
FROM packages c
JOIN provides b ON b.pkgKey=c.pkgKey
LEFT OUTER JOIN requires a ON a.name=b.name
JOIN permissive d ON c.rpm_license=d.rpm_license
WHERE c.arch='x86_64'
AND c.rpm_group LIKE 'Development%'
GROUP BY c.name
```

Applications- Restrictive

```
SELECT c.name, count(distinct a.pkgKey) as 'Reuse Count'
FROM packages c
JOIN provides b ON b.pkgKey=c.pkgKey
LEFT OUTER JOIN requires a ON a.name=b.name
JOIN restrictive d ON c.rpm_license=d.rpm_license
WHERE c.arch='x86_64'
AND (c.rpm_group LIKE 'Applications%' or rpm_group LIKE 'Amusements/%')
GROUP BY c.name
```

Applications- Permissive

```
SELECT c.name, count(distinct a.pkgKey) as 'Reuse Count'
FROM packages c
JOIN provides b ON b.pkgKey=c.pkgKey
LEFT OUTER JOIN requires a ON a.name=b.name
JOIN permissive d ON c.rpm_license=d.rpm_license
WHERE c.arch='x86_64'
AND (c.rpm_group LIKE 'Applications%' or rpm_group LIKE 'Amusements/%')
GROUP BY c.name
```

System Environment- Restrictive

```
SELECT c.name, count(distinct a.pkgKey) as 'Reuse Count'
FROM packages c
JOIN provides b ON b.pkgKey=c.pkgKey
LEFT OUTER JOIN requires a ON a.name=b.name
JOIN restrictive d ON c.rpm_license=d.rpm_license
WHERE c.arch='x86_64'
AND c.rpm_group LIKE 'System Environment%'
GROUP BY c.name
```

System Environment- Permissive

```
SELECT c.name, count(distinct a.pkgKey) as 'Reuse Count'
FROM packages c
JOIN provides b ON b.pkgKey=c.pkgKey
LEFT OUTER JOIN requires a ON a.name=b.name
JOIN permissive d ON c.rpm_license=d.rpm_license
WHERE c.arch='x86_64'
AND c.rpm_group LIKE 'System EnvirONment%'
GROUP BY c.name
```

User Interface- Restrictive

```
SELECT c.name, count(distinct a.pkgKey) as 'Reuse Count'
FROM packages c
JOIN provides b ON b.pkgKey=c.pkgKey
LEFT OUTER JOIN requires a ON a.name=b.name
JOIN restrictive d ON c.rpm_license=d.rpm_license
```

```
WHERE c.arch='x86_64'
AND c.rpm_group LIKE 'User Interface%'
GROUP BY c.name
```

User Interface- Permissive

```
SELECT c.name, count(distinct a.pkgKey) as 'Reuse Count'
FROM packages c
JOIN provides b ON b.pkgKey=c.pkgKey
LEFT OUTER JOIN requires a ON a.name=b.name
JOIN permissive d ON c.rpm_license=d.rpm_license
WHERE c.arch='x86_64'
AND c.rpm_group LIKE 'User Interface%'
GROUP BY c.name
```

Bibliography

[1] W. Frakes and C. Terry, "Software reuse: metrics and models," *ACM Computing Surveys (CSUR)*, vol. 28, no. 2, pp. 415–435, 1996.

[2] E. Raymond, "Homesteading the noosphere," *First Monday*, vol. 3, no. 10, pp. 1–28, 1998.

[3] A. Mockus, "Amassing and indexing a large sample of version control systems: Towards the census of public source code history," in *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*, 2009, pp. 11–20.

[4] ——, "Large-scale code reuse in open source software," in *Emerging Trends in FLOSS Research and Development, 2007. FLOSS'07. First International Workshop on*, 2007, pp. 7–7.

[5] S. Ramel, "Metrics of software reuse for free and open source software," *Centre de Recherche Public Henri Tudor*.

[6] K. J. Stewart, A. P. Ammeter, and L. M. Maruping, "Impacts of license choice and organizational sponsorship on user interest and development activity in open source software projects." *Information Systems Research*, vol. 17, no. 2, pp. 126–144, Jun. 2006.

[7] J. Colazo and Y. Fang, "Impact of license choice on open source software development activity," *Journal of the American Society for Information Science and Technology*, vol. 60, no. 5, pp. 997–1011, May 2009.

[8] J. Lerner and J. Tirole, "The scope of open source licensing," *Journal of Law, Economics, and Organization*, vol. 21, no. 1, pp. 20 –56, Apr. 2005.

[9] S. Levy, *Hackers: Heroes of the Computer Revolution - 25th Anniversary Edition*, anv upd ed. O'Reilly Media, May 2010.

[10] "Open sources: Voices from the open source revolution," http://oreilly.com/openbook/opensources/book/index.html, Mar. 1999. [Online]. Available: http://oreilly.com/openbook/opensources/book/index.html

[11] S. Williams, *Free as in Freedom: Richard Stallman's Crusade for Free Software*, first edition ed. O'Reilly Media, Mar. 2002.

[12] R. Stallman *et al.*, "The GNU manifesto," *Dr. Dobb's Journal of Software Tools*, vol. 10, no. 3, pp. 30–35, 1985.

[13] E. Raymond, "The cathedral and the bazaar," *Available in http://tuxedo. org/âĹij esr/writings/cathedral-bazaar*, 1998.

[14] M. Mustonen, "Copyleft: the economics of linux and other open source software," *Information Economics and Policy*, vol. 15, no. 1, pp. 99–121, 2003.

[15] A. M. S. Laurent, *Understanding Open Source and Free Software Licensing*, annotated, first edition ed. O'Reilly Media, Aug. 2004.

[16] "Mission | open source initiative," http://www.opensource.org/. [Online]. Available: http://www.opensource.org/

[17] "The GNU general public license," http://www.gnu.org/copyleft/gpl.html. [Online]. Available: http://www.gnu.org/copyleft/gpl.html

[18] "The GNU lesser general public license," http://www.gnu.org/copyleft/lgpl.html. [Online]. Available: http://www.gnu.org/copyleft/lgpl.html

[19] "Apache license, version 2.0," http://www.apache.org/licenses/LICENSE-2.0.html. [Online]. Available: http://www.apache.org/licenses/LICENSE-2.0.html

[20] "Open source initiative OSI - the BSD license," http://www.opensource.org/licenses/bsd-license.php. [Online]. Available: http://www.opensource.org/licenses/bsd-license.php

[21] "Open source initiative OSI - the MIT license (MIT)," http://www.opensource.org/licenses/mit-license.php. [Online]. Available: http://www.opensource.org/licenses/mit-license.php

[22] J. Lindman, A. Paajanen, and M. Rossi, "Choosing an open source software license in commercial context: A managerial perspective," in *Software Engineering and Advanced Applications, Euromicro Conference*. Los Alamitos, CA, USA: IEEE Computer Society, 2010, pp. 237–244.

[23] R. Sen, C. Subramaniam, and M. L. Nelson, "Determinants of the choice of open source software license," *Journal of Management Information Systems*, vol. 25, no. 3, pp. 207–240, Dec. 2008.

[24] J. Lerner and J. Tirole, "Some simple economics of open source," *Journal of Industrial Economics*, pp. 197–234, 2002.

[25] L. Gasser and W. Scacchi, "Towards a global research infrastructure for multidisciplinary study of free/open source software development," *Open Source Development, Communities and Quality*, pp. 143–158, 2008.

[26] M. Pizka, "Adaptation of large-scale open source software-an experience report," in *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on*, 2004, pp. 147–153.

[27] A. Kritikos, G. Kakarontzas, and I. Stamelos, "A semi-automated process for open source code reuse," in *5th International Conference on Evaluation of Novel Approaches in Software Engineering (ENASE'10)*, 2010, pp. 24–25.

[28] J. Li, R. Conradi, O. Slyngstad, C. Bunse, U. Khan, M. Torchiano, and M. Morisio, "An empirical study on off-the-shelf component usage in industrial projects," *Product Focused Software Process Improvement*, pp. 54–68, 2005.

[29] J. Li, R. Conradi, O. Slyngstad, C. Bunse, M. Torchiano, and M. Morisio, "An empirical study on decision making in off-the-shelf component-based development," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 897–900.

[30] S. A. Ajila and D. Wu, "Empirical study of the effects of open source adoption on software development economics," *Journal of Systems and Software*, vol. 80, no. 9, pp. 1517–1529, Sep. 2007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121207000076

[31] R. Ananthanarayanan, V. Chenthamarakshan, H. Chu, P. Deshpande, R. Krishnapuram, and S. Mohammed, "Dependency analysis framework for software service delivery," in *Services Computing, 2009. SCC'09. IEEE International Conference on*, 2009, pp. 89–96.

[32] W. Frakes and K. Kang, "Software reuse research: Status and future," *Software Engineering, IEEE Transactions on*, vol. 31, no. 7, pp. 529–536, 2005.

[33] M. Carro, "The amos project: An approach to reusing open source code," in *Proceedings of the CBD*, 2002, pp. 59–70.

[34] A. Pohthong and D. Budgen, "Reuse strategies in software development: an empirical study," *Information and Software Technology*, vol. 43, no. 9, pp. 561–575, 2001.

[35] P. Abate, R. Di Cosmo, J. Boender, and S. Zacchiroli, "Strong dependencies between software components," in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, 2009, pp. 89–99.

[36] R. McGill, J. Tukey, and W. Larsen, "Variations of box plots," *American Statistician*, pp. 12–16, 1978.