Evolution and Architecture of Open Source Software Collections: A Case Study of Debian

by

Raymond Nguyen

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2012

Abstract

Software has been studied at a variety of granularities. Code, classes, groups of classes, programs and finally large scale applications have been examined in detail. What lies beyond is the study of software collections that group together many individual applications. Collecting software and distributing it via a central repository has been popular for a while in the open source world, and only recently caught on commercially with Apples Mac app store and Microsofts Windows store. In many of these software collections, there is normally a complex process that must be followed in order to fully integrate new applications into the system. Moreover, in the case of open source software collections, applications frequently rely on each other for functionality and their interactions can be complex. We know that there are thousands of applications in these software collections that people depend on worldwide, but research in this area has been limited compared to other areas and granularities of software. In this thesis, we explore the evolution and architecture of a large open source software collections by using Debian as a case study.

Debian is a software collection based off the Linux kernel with a large number of packages spread over multiple hardware platforms. Each package provides a particular service or application and is actively maintained by one or more developers. This thesis investigates how these packages evolve through time and their interactions with one another. The first half of the thesis describes the life cycle of a package from inception to end by carrying out a longitudinal study using the Ultimate Debian Database (UDD)[29]. The birth of packages is examined to see how Debian is growing. Conversely, package death is also analyzed to determine the lifespan of these packages. Moreover, four different package attributes are examined. They are package age, package bugs, package maintainers and package popularity. These four attributes combine to give us the overall biography of Debian packages. Debians architecture is explored in the second part of the thesis, where we analyze how packages interact with each other by examining the package dependencies in detail. The dependencies within Debian are extensive, which makes for an interesting architecture, but they are complex to analyze. This thesis provides a close look at the *layered* pattern. This pattern categorizes each package into one of five layers based on how they are used. These layers may also be visualized to give a concise view of how an application is structured. Using these views, we define five architectural subpatterns and anti-subpatterns which can aid developers in creating and maintaining packages.

Acknowledgements

First and foremost, I would like to give my thanks to my supervisor Professor Ric Holt. Dr. Holt has been an incredible mentor and teacher. His guidance throughout my time at the University of Waterloo has been superb and without his advice and support, this research would not have been possible.

I would also like to extend my thanks to the other members of the SWAG lab. Firstly, to Professor Mike Godfrey whose has helped me to publish a paper in CSMR 2012. Without his advice and encouragement, the publication would not have been successful. Also, to the rest of the SWAG members for their support: Professor Reid Homes, Ian Davis, Olga Baysal, Sarah Nadi, Divam Jain, Kimiisa Oshikoji and Wei Wang.

Moreover, my family and friends are a vital part of my life and I am grateful for having such supportive people. They have always believed in me and their encouraged me every step of the way.

Finally, I would like to thank NSERC and the University of Waterloo for the financial assistance during my time as a Master's student.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis investigates the evolution and architecture of open source software collections. We use Debian as a case study to describe how it has grown and changed over a 12 year time span. In addition, the architecture of Debian is presented through a number of patterns that encapsulate how the software within this collection interacts with each another.

We define a software collection as a large number of independently developed products that are engineered to work together. In the open source world, these collections take the form of software distributions which group thousands of applications and libraries into one integrated system. Examples include Red Hat Linux, Ubuntu, FreeBSD and Debian. Software packages found within these distributions are normally developed externally, while a different set of people involved with the distribution integrates the software with the rest of the distribution. Software collections are constantly changing, evolving and growing. New features get added to existing applications and new software is always waiting to be included. In the case of free and open source software collections, it is up to the people on the distribution team to take the original source code and package it in a way that it seamlessly integrates into the collection. For the majority of software found in an open-source software collection, the people who do the packaging are not those who wrote the software; however, communication between the two parties is encouraged. A software collection is defined through its packages. Therefore, to study the evolution of software collections is to study how packages come and go. In the first half of the thesis we explore the life and death of packages to gain insight into how software collections change through time.

The second half of the thesis is about the architecture of open source software collections. Before software architectures were documented and studied in detail, they were used implicitly. Even when software is mainly built in an informal manner, good developers often adopted architectural patterns. Over time, these patterns become more explicit as people recognized similarities between how software is developed [35]. The same phenomenon is occurring in the packaging of open source software. Developers doing the packaging and maintaining are following implicit rules when it comes to integrating new or updated software into a software collection. This thesis takes a step towards formalizing those rules. For small applications, this task is simple; however, for larger applications, the packaging process can be very complex. This complexity can lead to bugs which can render the distribution unusable. Debian maintains a bug tracking system in order to fix packaging related bugs. In the past 10 years, there have been over 500,000 bug reports. Learning about how the packages are structured in a software collection will not dramatically reduce the number of packaging related bugs overnight, but an understanding of how packages interact with one another is expected to help.

Software architecture can be defined as elements from which systems are built, interactions among those elements, together with patterns that guide their composition [13]. In this thesis, we explore the software architecture at a granularity higher than individual systems. Packages are the elements that interact with one another to form the basis of software collections. The most apparent method of package interaction is through their specified dependencies. Packages commonly require the installation of other packages in order to run. These dependencies are listed in the package's corresponding control file and can come in a variety of flavors (explained in Chapter 3).

While there are a large number of open software collections that could be studied, we chose to examine Debian from an architectural point of view. Debian is a prime candidate to study since it is one of the largest and most mature software collections [28]. Moreover, to upload packages onto Debian, developers must go through an extensive certification process [18]. This is done to ensure uploaded packages will be of sufficient quality and to restrict upload privileges to only those developers who have proven their ability. These points make Debian a good representation of open sourced software collections and gives us some confidence to generalize our findings across other software collections.

1.1 Contributions

This thesis contributes to the body of knowledge of open source software collections and especially Debian. In Chapter 4, we expand upon existing work in software evolution to look at how Debian as evolved. Similar to other papers on evolution, the value lies in our observations. We extract facts and patterns about the system which have not been documented before. The information obtained forms a base from which numerous other research questions can arise. Software collection policy makers may also use the observed trends to adjust certain policies to optimize the quality and quantity of incoming packages. In Chapter 5, we explore and document Debian's architecture. We analyze how packages interact with each other by examining the package dependencies in detail. Much related work has been done to model and visualize software package dependencies. But to the best of our knowledge, our work is the first to analyze the architecture of an open-source software collection. Knowing the architecture of a software system aids in understanding as it provides a high-level overview of the system. We also document a set of packaging patterns and anti-patterns, which could be helpful for inexperienced developers looking to do software packaging. Mining this software repository and analyzing the data provides insight into the concept of an immensely sized software collection.

1.2 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 presents the related work in the areas of software evolution, software architecture and the various work done regarding software collections. Chapter 3 provides background information on Debian and definitions on the key terms used in this thesis. Chapters 4 and 5 present the core of the research, where Chapter 4 discusses the evolutionary aspects of Debian and Chapter 5 explains Debian's architecture. Chapter 6 presents some threats to validity along with future work. Finally, Chapter 7 provides the conclusion to this thesis.

Chapter 2

Related Work

Software evolution studies have been performed on a variety of systems in different contexts. Some of the first few empirical studies were carried out by Lehman on IBM's OS360 and other large industrial systems. He initially formulated three "Laws of Evolution": continuing change, increasing complexity and self-regulation. Later, the set of laws were expanded upon and now include eight laws [25][24]. These laws were meant for E-type systems which are described as monolithic systems produced by a team within a company. Open source software does not generally fit into this E-type mold and so others studies needed to be done to see how open source software evolves. Godfrey and Tu used Linux to explore the evolution of a large scale open source software project [16]. Instead of growing slowly like a typical large complex system, they discovered Linux is growing at a super linear rate. Other studies in this area have had similar findings [22][31][36]; more recently Deshpande and Riehle looked at 5000 active open source software projects to find that many grow at an exponential rate, which is even faster than previously thought [11].

Due to the immense size and open nature of Debian, there are a number of studies pertaining to this Linux distribution. For example, Boender, Di Cosmo, Vouillon, Durak and Mancinelli present some tools that aim at helping distribution editors with maintaining the huge package bases associated with these distributions. They aim to improve the quality by detecting errors and inconsistencies in an effective, fast and automatic way [7]. Others look for ways to detect failures more quickly to improve the overall quality of the system through the package management system [1][40]. The first major study on the state of Debian was performed by J. M. Gonzalez-Barahona et al in 2001 on Debian 2.2 [17]. The entire release was downloaded and analyzed. It was discovered that this version of Debian contained roughly 60,000,000 source lines of code (SLOC). Using the COCOMO model, they estimated that the cost of development was close to 2 billion dollars USD taking up

170,000 person-months. At the time, this was twice as large as Red Hat 7.1 and also larger than the latest Microsoft operating system. Programming languages were also analyzed and they found that over 70% of SLOC were written in C, followed by 10% in C++, 5% in LISP and 5% were Shell scripts. The remaining 10% were comprised of a large variety of different languages. Since then, Debian has grown tremendously.

Follow up papers measuring Debian 3.0, 3.1, 4.0 and 5.0 were subsequently written by J.J Amor [5][3][4]. These case studies are similar in nature and employed techniques and methodologies much like those of Gonzalez-Barahona [17]. These studies give a snapshot of the size of Debian for its major releases. They provide some notion that Debian is growing rapidly, but do not attempt to chart this growth in these papers. In a subsequent study titled "Macro-level software evolution", the results from these papers were combined to perform a longitudinal analysis of the size of Debian [18]. The authors concluded that the SLOC of Debian doubles every two years and noted that this growth is mainly due to the influx of new packages rather than existing packages getting larger. They also believed the growth may pose a significant risk if continued because the number of Debian developers is limited. To the best of our knowledge, the work done by Robles, Gonzalez-Barahona, Miehlmayr and Amor in MSR 2006 [33] and the subsequent journal publication in Empirical Software Engineering [18] are the only longitudinal studies done on a software collection. In their work, they only focused on the increasing rate of uploaded packages and the change in programming languages over time. Our study not only tracks the rate of incoming packages, but also removed packages, incoming/outgoing package maintainers, package lifespan, package bugs and package popularity. By examining these additional package attributes, we get more information on how the Debian distribution is evolving.

The idea of using packages to explore the architecture of large open source software collections has not been explored in great detail. However, ideas and techniques presented in other papers are used to as a springboard for our study. To start, let us look at the work done in the software architecture field. There are numerous books that describe the concept of architecture and architectural patterns developers can employ [35][6][8]. These books are centered on a select number of architectural patterns where they present modeling techniques, design, implementation and deployment. These books resulted from research done in the late 80s and early 90s. In this thesis, we provide a set of patterns based on how packages are structured. More recent research in software architecture has progressed to support emerging technologies such as the cloud [26] and mobile development [34][39]. This illustrates the fact that while general software architecture has been well documented, architecture of newer technologies and systems are always being discovered.

Package dependencies are especially interesting due to their complex interactions and size. The study of dependencies is not the same as the study of architecture; however, the two areas are closely related and we make use of Debian's dependencies in order to define its architecture. There has been a bit of work done in the field of package dependencies and one such study is from German, where he defines some metrics to quantify the dependencies of a given software application [14]. Another research paper in this area comes from German, Gonzalez-Barahona and Robles. In this work, they create a framework to model, extract, study and visualize application dependencies [15]. They introduce the notion of an inter-dependency graph (IDG) and apply it to Debian. The IDG is similar to the GIMP dependency graph in Figure 5.2, but their graph contains more information about the application. For example, depending on the type of package, nodes can be represented as circles, squares or diamonds. Similarly, they use a different set of edges to represent different dependencies (eg. optional vs required). Their dependency graphs are more detailed and convey more the information found in the control file. Abate, Boender, Cosmo and Zacchiroli introduce the novel notion of strong dependencies between software components. They argue that there is a different dependency graph to be studied to grasp meaningful relationships between packages [2]. A strong dependency graph represents the semantics of inter-package relationships, in which an edge between two packages is drawn only if the first cannot be installed without installing the second. One of the key use-cases for strong dependencies is to compute package sensitivity. They state that the higher the sensitivity of a package, the higher the minimum number of packages will be potentially affected by a change. Therefore the study of strong dependency and sensitivity directly helps with quality assurance and evaluation of upgrade risks. This idea is further expanded upon in a follow up paper [1].

Other studies have used network theory to help explain package dependencies in open source software collections [37]. Complex networks are an emerging research area in fields such as biology and sociology, but not generally used in software engineering. One can define a complex network as a graph with non-trivial topological features (features that do not exist in simple networks such as lattices or random graphs) [41]. LaBelle and Wallingford modeled Debian and BSD each as an unweighted simple graph G = (V, E) and then proceeded to analyze the resulting properties [23]. They concluded that software collections share properties typical to other real-world networks. In particular, they noticed a small-world effect, which means short geodesic path lengths and high clustering. The same small-world conclusion was reached by Sousa, Menezes and Penna in their study of Debian [10]. Packages are the building blocks in the majority of open source software collections and their interactions are defined by dependencies. Much of the work package dependencies gives a vague sense of architecture, but none of them go deeply into this topic. In our work, we leverage the previous research done on package dependencies to define a set of patterns that can be used to help explain the architecture of Debian.

Chapter 3

Background Information on Debian

Debian's purpose is to collect software packages and take them through a monitored maturing process. Users can choose to install different versions of packages with known levels of reliability. Debian started as a project in August 1993 by Ian Murdock because he felt there was a need for a distribution to be maintained in an open manner similar in spirit to Linux and GNU. His thoughts and reasoning behind establishing Debian are documented in the Debian Manifesto [28]. At the time, Softlanding Linux System (SLS) was the most popular distribution, but was bug-ridden and hard to maintain. Murdock wanted to improve this practice by designing Debian to be an open and modular system to ensure a high standard. This idea for free high quality software was further established in 1996 by Bruce Perens who replaced Murdock as project leader. During his tenure, he introduced the Debian Social Contract and the Debian Free Software Guidelines [30]. These documents describe the moral agenda of Debian and how developers should act when contributing to the project. The values of openness and quality were the foundation on which Debian was built. As a result, the processes for maintaining and releasing software packages within Debian reflect their core beliefs.

3.1 Debian Packages

For Debian to be as successful as it is today, volunteers have been tasked with the job of taking open source software and packaging it. The packaging process integrates each new package with Debian's existing infrastructure and enables simple installation for potential users. Packages in Debian come in two flavors: source and binary. A source package can generate one or more binary packages. The sources are what Debian developers work with and upload, while the binary packages are meant for the end user to download and install (Figure 3.1). Only the binary packages have relations and dependencies to and from other binaries. For the term packages, we generally refer to source packages in Chapter 4 when exploring Debian's evolution. In Chapter 5, we shift our focus to binaries when conducting research on Debian's architecture and so the term package will refer to binary packages.

To help with the task of packaging software, Debian has (an implementation of) a package management system. All package management systems contain three main parts: the package format specification, the package handler and the manager itself. In Debian's case, packages are specified in a .DEB format, the package handler is called dpkg and the package manager is called APT. Other package management systems are made up of other tools with their own strengths and weaknesses, but all accomplish the task of package maintenance. Although package managers are powerful, they are unable to automatically detect conflicts and dependencies to and from packages. This information must be manually specified in the control file stored in the DEB file.

The control file is the metadata that is associated with each package. It includes the inter-relationships describing the static requirements for the package to run. Requirements are listed in terms of other packages with possible restrictions on the version number. The list below summarizes the different relations between packages that can be specified.

- *Depends*: Packages specified under the depends field are absolute dependencies and therefore must be installed. Otherwise, the depending package will not be usable.
- *Recommends*: At times, a package may require another. The software to be installed could work, but not without limitations in functionality.
- *Suggests*: The software could be enhanced by the presence of other packages. For example, a package containing higher resolution fonts is suggested.
- *Enhances*: Similar to Suggests but works in the opposite direction. It is used to declare that a package can enhance the functionality of another package.
- *Provides*: The package may provide the functionality of another.
- *Conflicts*: Packages listed in this field cannot coexist with the package declaring the conflict.
- *Replaces*: A package may supercede parts of another package. Installed packages cannot be overwritten unless explicitly stated in this field. In certain situations such as when newer packages are renamed or when monolithic packages are split up, the

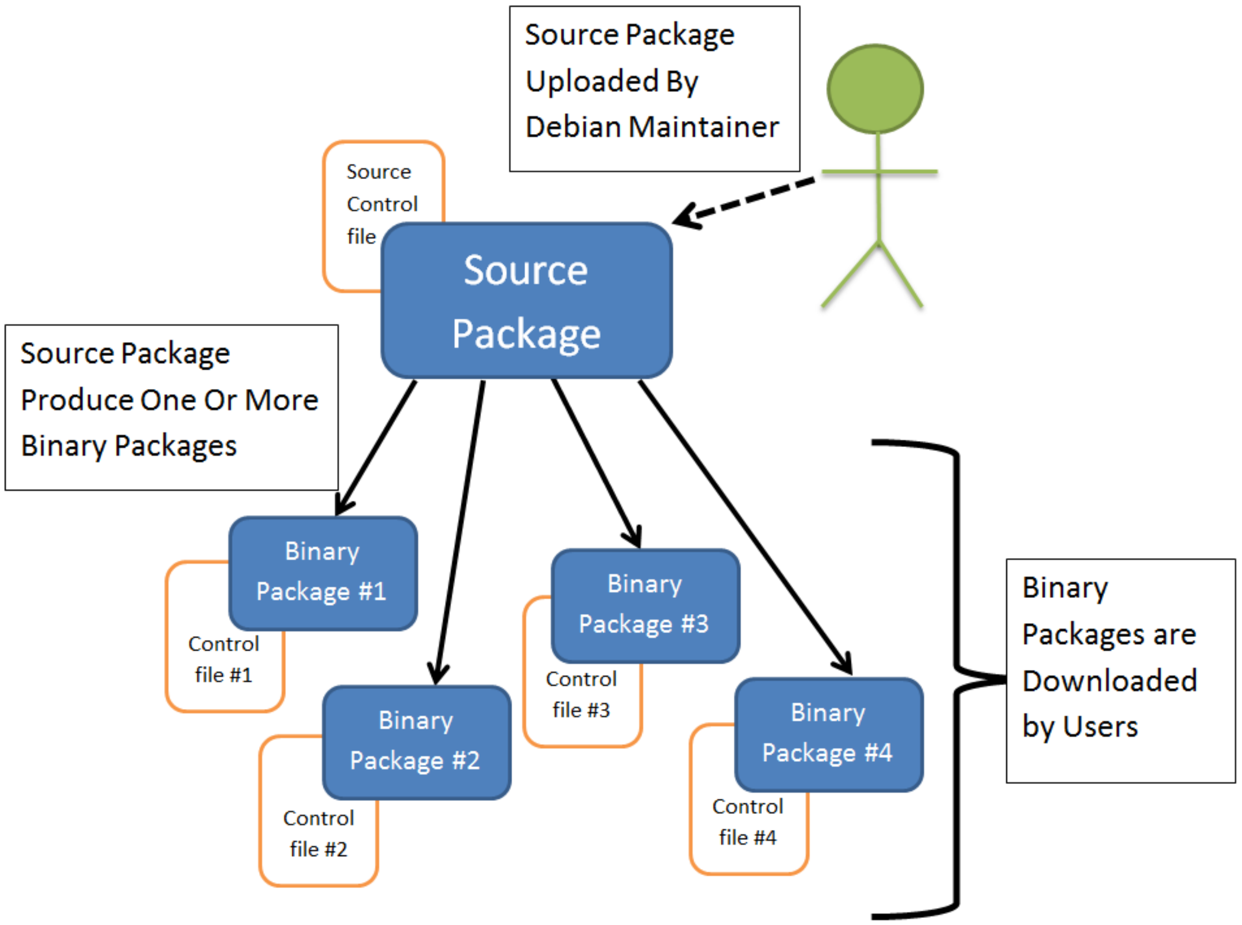

Figure 3.1: Debian Source and Binary Packages

older version must be replaced. Most often, this field is used in conjunction with the Conflicts relation.

The interaction between packages can be complex with a variety of different relations. This thesis will focus on the *depends* relation. The *depends* relation is by far the most common relationship and arguably the most important. The ability to manage dependencies is the basis of Debian and we explore it further in this thesis.

3.2 Lifecycle of Debian Packages

In order to explain the lifecycle of a source package, the process in which packages get uploaded and moved around in Debian must first be understood (Figure 3.2). Free and open source software is created to be used on a variety of platforms by people whom Debian calls the upstream developers. The people who package the software for Debian are called maintainers and are usually not a part of upstream development. Maintainers take the upstream source and through a series of steps create a Debian source. Each individual package that a maintainer creates can be identified through the package name and version. The version of a Debian source consists of the upstream version followed by a hyphen and then the Debian version. For instance, package `ABC 1.0-1` is the first Debian release of `ABC`. If the package has a bug which gets fixed, then the new version will be `ABC 1.0-2` since the upstream source stays the same. If the upstream source later gets updated to version 1.1 then a new package is released labeled `ABC 1.1-1`. We consider every unique package name and version to be an individual package with its own time of upload (birth) and removal time (death). Packages with the same name, but different versions will be defined as a package family in order to distinguish between the two concepts. For example, package `ABC 1.0-1` and `ABC 2.0-1` are considered two separate packages, but belong to the same package family.

Once the maintainer has done the necessary work to transform the upstream source into a Debian source and binary, the package can now be uploaded onto the Debian servers so that users throughout the world can download and install this particular piece of software. While anybody is able to create a package, only select members of the Debian community have upload rights. These people are the Debian developers, who have gone through a lengthy application process to achieve this title.

The Debian repository consists of three main distributions called: unstable, testing and stable. See Figure 3.2. When a package first gets uploaded, it will appear in the unstable distribution. After 10 days without any release critical bugs, it will flow down into the testing distribution. The package does not get deleted from unstable, but rather a pointer to the package is made in testing. After a period of roughly 1.5 years, the testing distribution becomes frozen. In this state, no new packages from unstable can flow into testing unless they include release critical bug fixes. These bug fixes are manually checked by the release team to make sure no other bugs are introduced. Around six months later and once the count of release critical bugs reaches zero; the frozen testing distribution will be released as stable. At this point, a new version of testing will be forked from unstable and the community will once again start working towards the next stable release.

The distribution that Debian recommends its users install is the stable distribution, since it is the most mature with the smallest number of bugs. The down side is that the packages found in stable can be quite outdated and so users who prefer up-to-date software and do not mind the occasional bug choose to install the unstable distribution. The testing distribution is less buggy than unstable, but when a package in testing does break, it could take weeks or even months before there is a solution available. This situation occurs when the unstable version is continually replaced. The ten day grace period is never reached and the latest version never gets to opportunity to move out of testing.

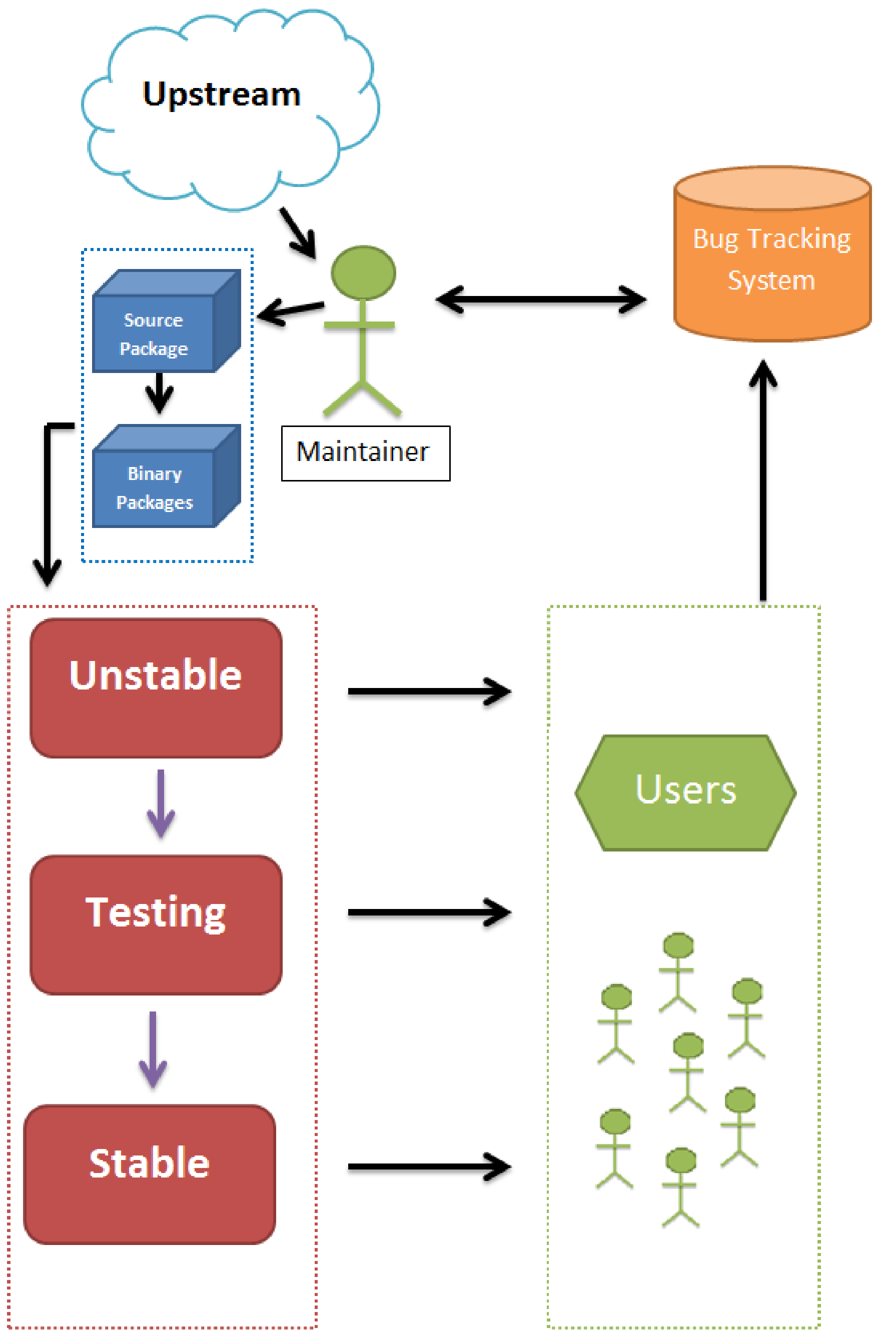

Figure 3.2: Flow of Packages in Debian

Chapter 4

Life and Death of Software Packages

In this chapter, we explore source packages from their inception to when they are removed or replaced. By studying the life and death of all the packages involved with the Debian software collection, we are able to get a sense of how the system is evolving. Figure 4.1 overviews the different stages a package goes through once it gets uploaded to the Debian repository. Each package starts out in the unstable repository and if there are no issues, it will proceed to testing and then finally stable. At any point in time, a package could get removed or replaced which would signal death. Besides studying the lifecycle of packages in detail, we also examine different attributes that are produced during a package's lifespan. These are: maintainer information, bugs, age and popularity. Looking at the attributes for each individual package does not tell us much about the software collection, but when exploring the attributes through the entire set of all uploaded packages, we learn a lot about how the collection is changing. For example, we can determine the number of new maintainers that join and contribute to Debian on a yearly basis. Or, the rate of incoming bug reports compared to previous years. Studying the evolution of Debian opens the doors to many other research questions. It can also determine weaknesses that can be addressed by those in charge of the collection.

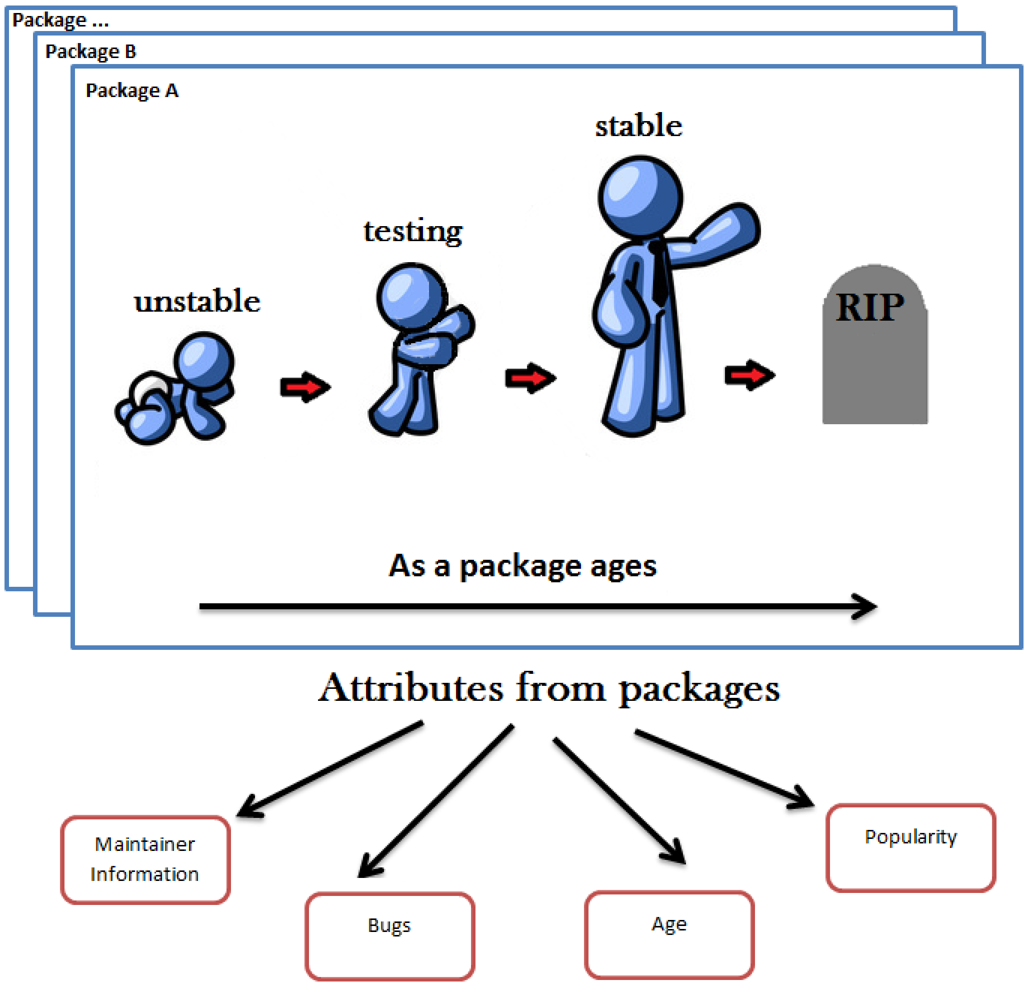

Figure 4.1: Package Lifecycle with Four Package Attributes

4.1 Methodology

This study was performed using the Ultimate Debian Database (UDD) [29]. The UDD was created to solve the intrinsic difficulty in correlating distribution data to package and maintainer metrics. The authors of the UDD proposed it has a QA tool for maintainers and also as a research tool for those who wish to data mine Debian. Currently, only a couple other studies have made use of the UDD [21][9], as explained in Chapter 2. Java code was written to extract the content from the database and to perform the necessary processing on the data. The results from this thesis come from the version of the UDD downloaded on February 1, 2012.

4.2 Birth of Packages

Debian is a large distribution with over 22,000 source package families available for download in its current stable distribution. But on a package to package basis, how many are being uploaded into Debian yearly? We define the birth of a package as any new package uploaded onto Debian; this includes both updates to an existing package or a new package that has never appeared before. Figure 4.2 summarizes our findings where we noticed that since 2004, the number of packages being uploaded (added to Debian) per year have stabilized at around 30,000. Since 2001, there have been at least 25,000 packages uploaded/year. This means that the birth rate of new packages within Debian has been roughly the same for the past 10 years.

An important point in Figure 4.2 is the big jump in upload activity from 2000-2001 when package uploads went from just over 10,000 to 25,000. We hypothesize that this increase is the result of a couple of factors that occurred in late 2000. Firstly, Debian made major changes to the archive and release management software. New "package tools" were introduced to help developers with the packaging process. The testing distribution was introduced for a relatively stable staging area for the next stable release. Secondly, developers began holding an annual conference called DebConf with talks and workshops for developers and technical users. We conjecture that this conference attracted a number of new people who then contributed to Debian.

Figure 4.4 breaks down the numbers in Figure 4.2 by using a more fine-grained time scale. Each column represents one month rather than one year. By presenting the data in this fashion, we learn more about when developers are uploading packages. Certain periods of time produce a greater number of new packages than others. Figure 4.4 exhibits noticeable hills and valleys: periods where upload activity are higher or lower than average. At a quick glance, there does not seem to be any pattern as to why this occurs. Upon closer inspection, there does appear to be a correlation between the hills and valleys to Debian's release cycle; see the lines that point to valleys in Figure 4.4. Each dip in the graph occurs when Debian's testing distribution is frozen. This is when no new packages from unstable can enter testing and thus will not be in the next stable distribution. When the new stable version of Debian is released, the number of new packages arriving into the system shoots back up.

Development of software upstream is independent of Debian, yet when Debian is frozen, the upload rate of new packages is about half as much the usual rate (Figure 4.4). We are not quite sure why this is the case, but developers could feel less motivated by the fact that any new packages they create will not make it into the testing and stable distribution for a long time. Another explanation could be that developers are spending their time fixing bugs during this freezing period and therefore are too busy to create new packages. Further research may determine more precisely the source of these hills and valleys.

Figures 4.2 and 4.4 show the total number of individual packages uploaded. We now ask: What is the percentage split between new package family births and package births that are updates to a pre-existing one? New package families in Debian are graphed in Figure 4.3. It appears that Debian is receiving an ongoing influx of new package families. Comparing the numbers to Figure 4.2, we determine that 7.5% of all uploaded packages are new ones, while the other 92.5% are updates to a pre-existing package. In the most recent year, there were over 2800 new package families that entered Debian. This represents a 49% increase over the average rate of 1885 new packages per year. The year before that, only 1791 new packages were uploaded, so it makes it difficult to predict how many new packages to expect next year. But over the past three years, the rate of new incoming package families has been increasing. This means the selection of packages that a Debian user can install from is increasing.

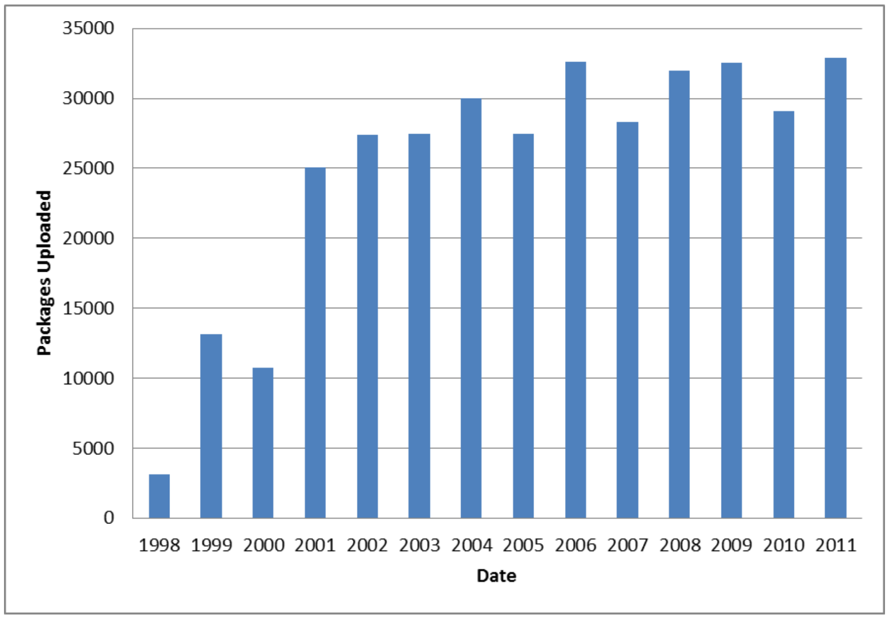

Figure 4.2: New Packages Uploaded By Year

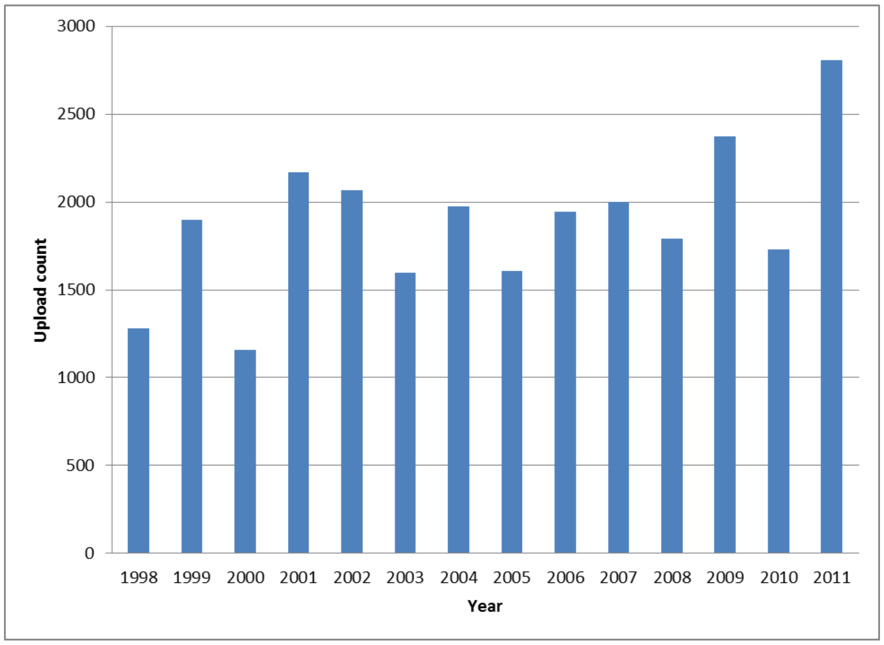

Figure 4.3: Upload Rates For Original Package Families

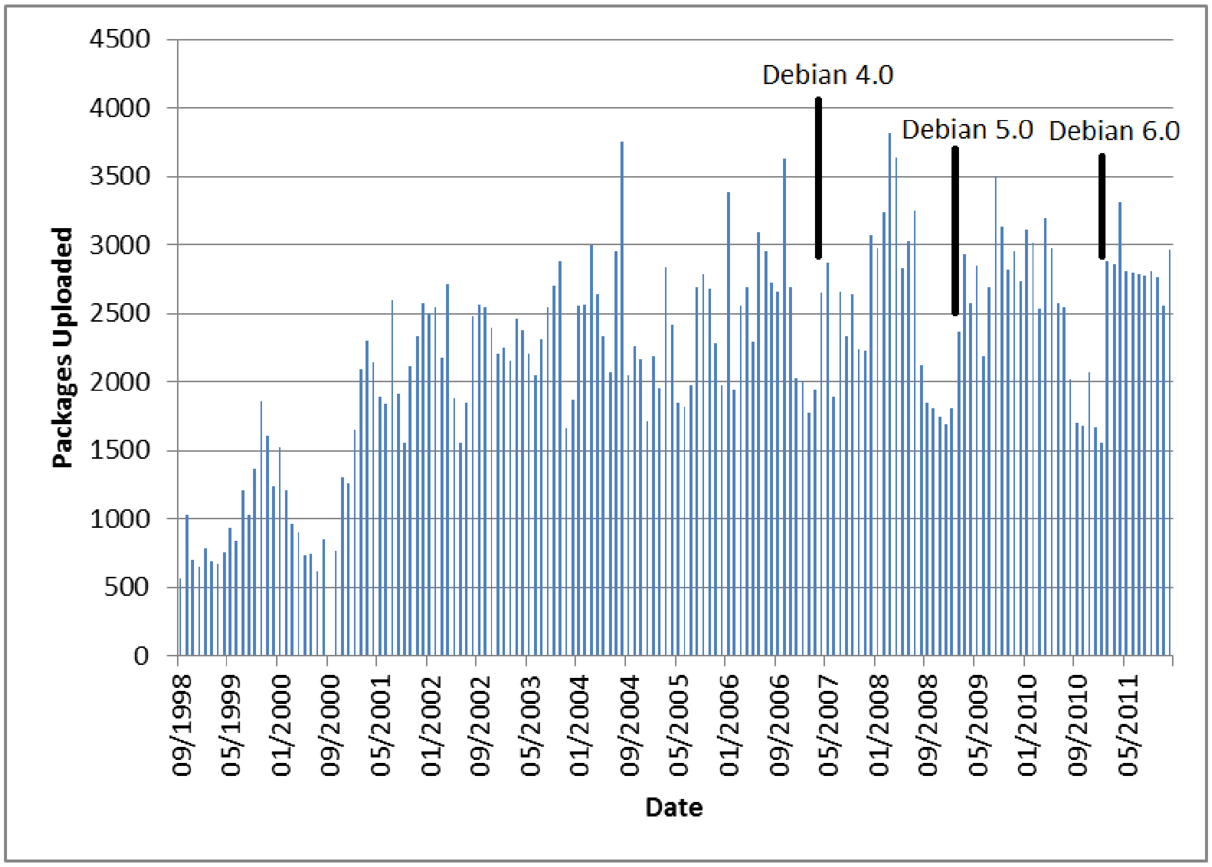

Figure 4.4: New Packages Uploaded By Month

4.3 Death of Packages

Software is not usually immortal and thus does not tend to live on forever. Rather, software is constantly changing; it can be an updated version with bug fixes and new features, or a brand new piece of software that renders an older one obsolete. Therefore, package death can occur in two ways in Debian: First, through a forced removal by the ftp master, or secondly through replacement when a newer version is uploaded.

We will first examine package death through forced removal. Packages are not removed from Debian unless there is a request with a stated purpose. These requests normally come from the maintainers of the package or the QA team for reasons such as bugs, lack of commitment to the package or obsolescence. In addition, there is a script called auto-cruft which automatically sends a removal request if it notices an anomaly in the system in terms of dependencies or naming. These motives are documented and the removal is done by the ftp master.

Figure 4.5 presents the number of source package deaths (through removal) per year for the three main distributions (unstable, testing and stable). The number of packages removed appears to be fairly steady on a yearly basis. One would assume that as more packages accumulate in the system, we should see an increase in the removal of older and obsolete packages. However, the data suggests otherwise. This means that for the most part, once a package gets uploaded to Debian, there is small chance of it being completely removed outright from the system. In other words, packages have the potential to live very long. Table 4.1 shows the breakdown of where source packages were removed. The majority of source deaths occur in the unstable distribution which means once a package flows into testing and stable, the chance of it being removed is very slim.

Besides an obvious removal from the system, software packages are also considered to die when a newer version is updated. We noted earlier that 92.5% of all new package uploads are replacing an older version. While packages have the potential to live for a long time; this does not happen often since the packages are constantly being replaced by newer versions. For example in 2011, if we subtract the number of new package family uploads to the total number of uploads, we discover that 30,077 packages were replaced that year. This means that 30,077 older packages were deleted and removed from Debian. Are certain package families skewing this number with very rapid releases or are all packages updated at a fairly regular interval? We will take a look at this in the next section which discusses package age.

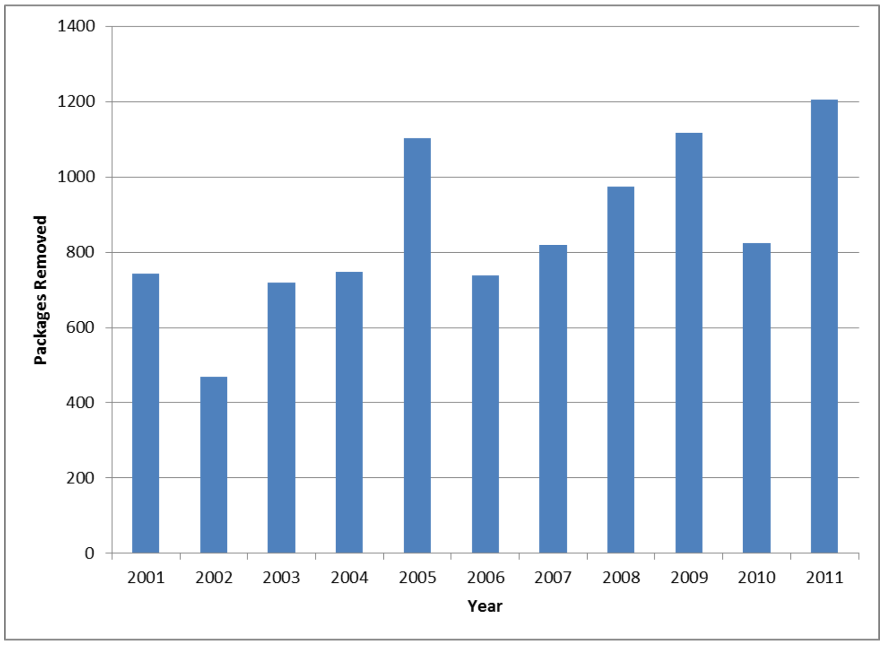

Figure 4.5: Forced Removal of Packages Per Year in Unstable, Testing and Stable

Table 4.1: Breakdown of Package Deaths by Removal From 2001-2011

Debian distribution	**Number of removed packages**
Stable	115
Testing	64
Unstable	8908

4.4 Package Biography

4.4.1 Package Age

With the information of when packages enter and exit the system, one useful attribute to come from the data is package age. The age of a package is one of the four attributes that helps tell the story of each individual package. Examining the entire set of all packages helps us to learn about how Debian evolves.

Package Lifespan

A long life span could mean that the package is mature and stable, while a shorter life span can signify that a package is gaining new features and therefore less stable. Figure 4.6 takes all the packages uploaded since 1998 and displays them in descending order according to how long they live. Figure 4.6 shows that many packages have a very short lifespan. For example, 23172 or 7.01% of all packages get replaced/removed (die) after only one day and 12485 (3.78%) are gone after two days. For this reason, we say Debian source packages have a high infant mortality rate. Almost half of all packages die within their first month of existence. This suggests that developers are very aggressive in releasing software with minimal time spent testing. Table 4.2 provides an overview of the age on all the packages that have died in Debian. It reveals that 88.15% of packages live less than one year. We conclude that a number of Debian source packages have very short lifespans.

It follows that, Debian packages are updated quickly. It could mean that packages get frequent updates from the upstream developers. It could also mean that packages are found to be buggy and a fix is uploaded immediately. Either way, this is a sign that rapid development is occurring. The one risk of such fast package turnover is the introduction of system breaking bugs; which can render the system inoperable and unable to boot. However, users of Debian *unstable* are prepared for this possibility and the *stable* distribution almost completely mitigates this risk with its extensive testing period.

Current Package Age

The above section explored the lifespan of every source package to be uploaded onto Debian. What about the packages in Debian today? The population pyramids in Figures 4.7 and 4.8 present the age of packages within the three different Debian distributions as of February 1st 2012. Figure 4.7 compares the age of packages in *testing* to *unstable.* At first glance,

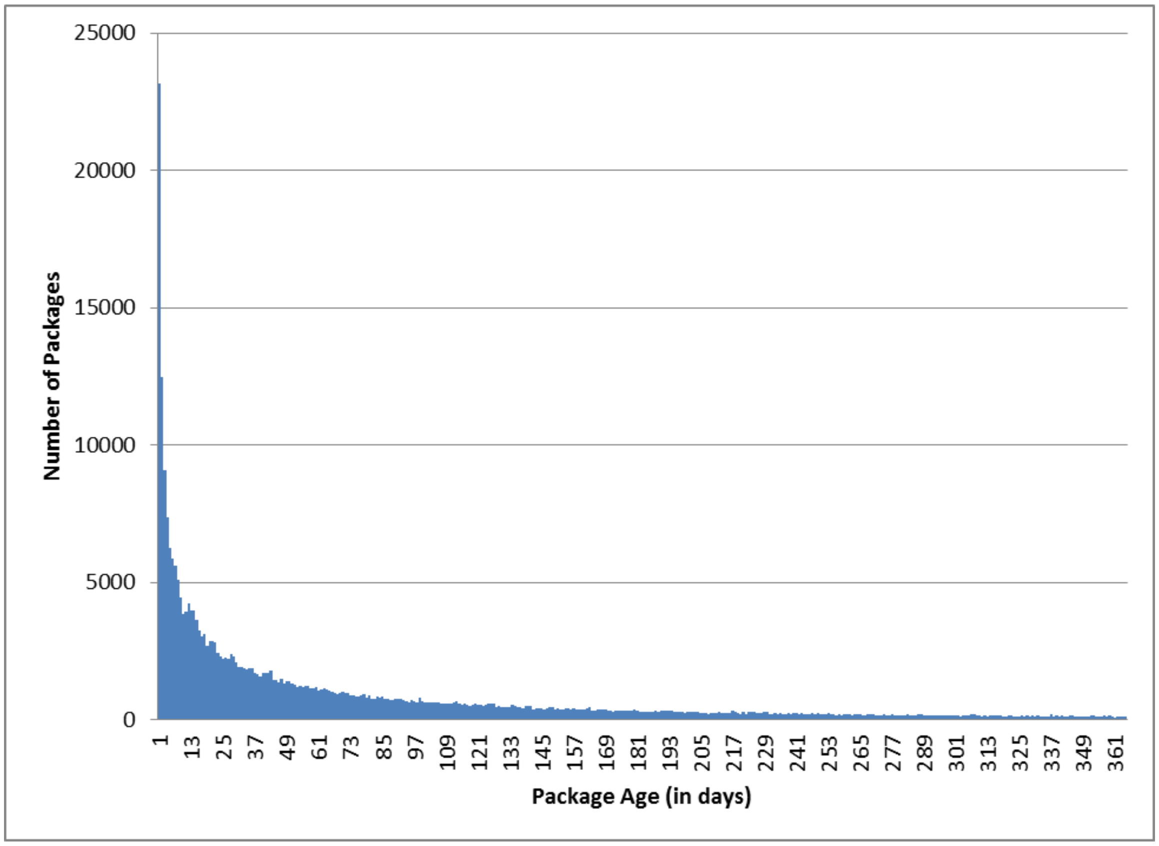

Figure 4.6: Lifespan of Debian Source Packages Under One Year

Table 4.2: Lifespan of all source packages

Age (years)	0-1	2-3	3-4	4-5	5-6	6-7	7-8
Package Count	291005	20485	7160	3590	1711	1354	970
Percentage of total (%)	88.15	6.2	2.17	1.09	0.52	0.41	0.29
Age (years)	8-9	9-10	10-11	11-12	12-13	13-14	13-14
Package Count	818	881	762	654	189	430	125
Percentage of total (%)	0.25	0.27	0.23	0.2	0.06	0.13	0.04

unstable and testing seem nearly identical. However, there is a difference near the bottom which shows that a greater percentage of *unstable* packages are younger than those in *testing*. This makes sense since *testing* should mirror *unstable* closely when it is not in its frozen state. Moreover, only packages that reach 10 days old will be made available to *testing*. This explains why *unstable* contains a greater percentage of very young packages.

The shape of the population pyramid resembles a triangle with a wide base and a narrow top. This indicates that the population is quite young and coincides with the data regarding package lifespan. While we know that most packages have a very short life, this does not mean that all the packages are constantly being updated. In fact, the population pyramid tells us that 11% of packages in unstable are younger than one month. Comparing this dataset with the data on package lifespan, we now know that Debian developers update a particular package many times over a short time interval. After the rapid updates, there will be an extended break before the package gets updated again. It is plausible to assume that a developer might try to get the latest upstream version onto Debian as soon possible while spending the next couple of days improving how the package is organized and fixing any bugs that may appear.

The contractions and expansions in the population pyramid correlate with when uploads occur. In Figure 4.4, we noticed that there are fewer package uploads when Debian is gearing up for its next stable release. One major contraction from the bottom occurs on February of 2011, which was when Debian 6.0 was released. Another contraction came February 2009 when Debian 5.0 was introduced. This figure once again shows that Debian developers do not like to upload packages when the testing distribution is frozen. For the packages that were uploaded during this time, there is a good chance that it has been updated more recently. For these reasons, not many packages introduced from this time period remain.

Figure 4.8 shows the current age distribution of packages in *stable* compared to *testing*. Unlike how packages flow from *unstable* to *testing*, *stable* only gets released approximately every two years. Since the *stable* distribution was released one year ago from when we gathered the data, it makes sense that all packages are at least one year old. Packages in *stable* are not as up to date as those in *testing* and *unstable*. Installing the *stable* distribution of Debian means that as time goes on, the distribution will become more and more out of date. Even installing stable on the first day of release does not guarantee the latest packages because the frozen period prohibits new packages from entering. In contrast, the *unstable* distribution contains mainly new packages that are up to date with the latest features from the upstream developers. Figure 4.8 does a good job visualizing the central difference between testing and stable as it shows exactly how much older the packages in stable are compared to testing.

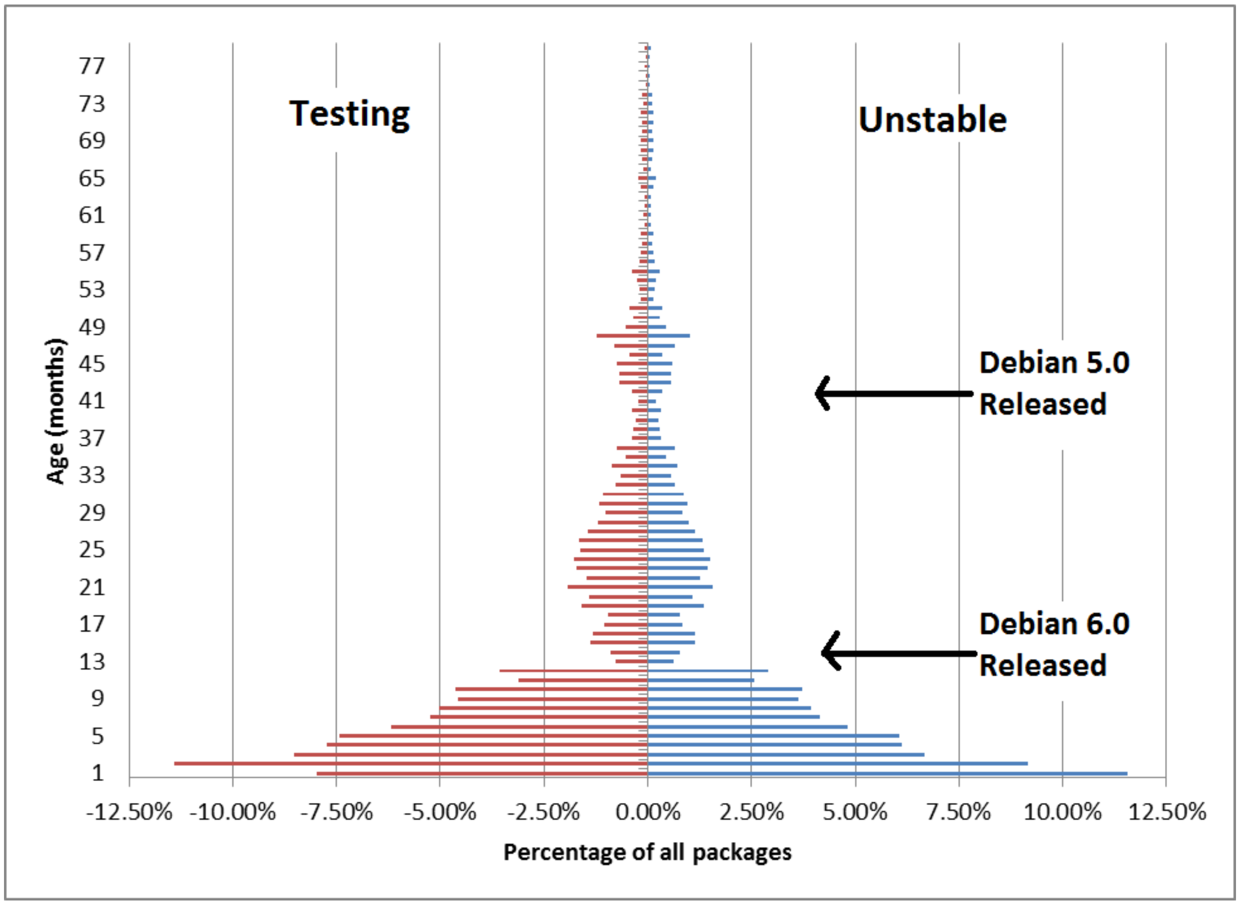

Figure 4.7: Age of Unstable vs. Testing as of Feb.1, 2012

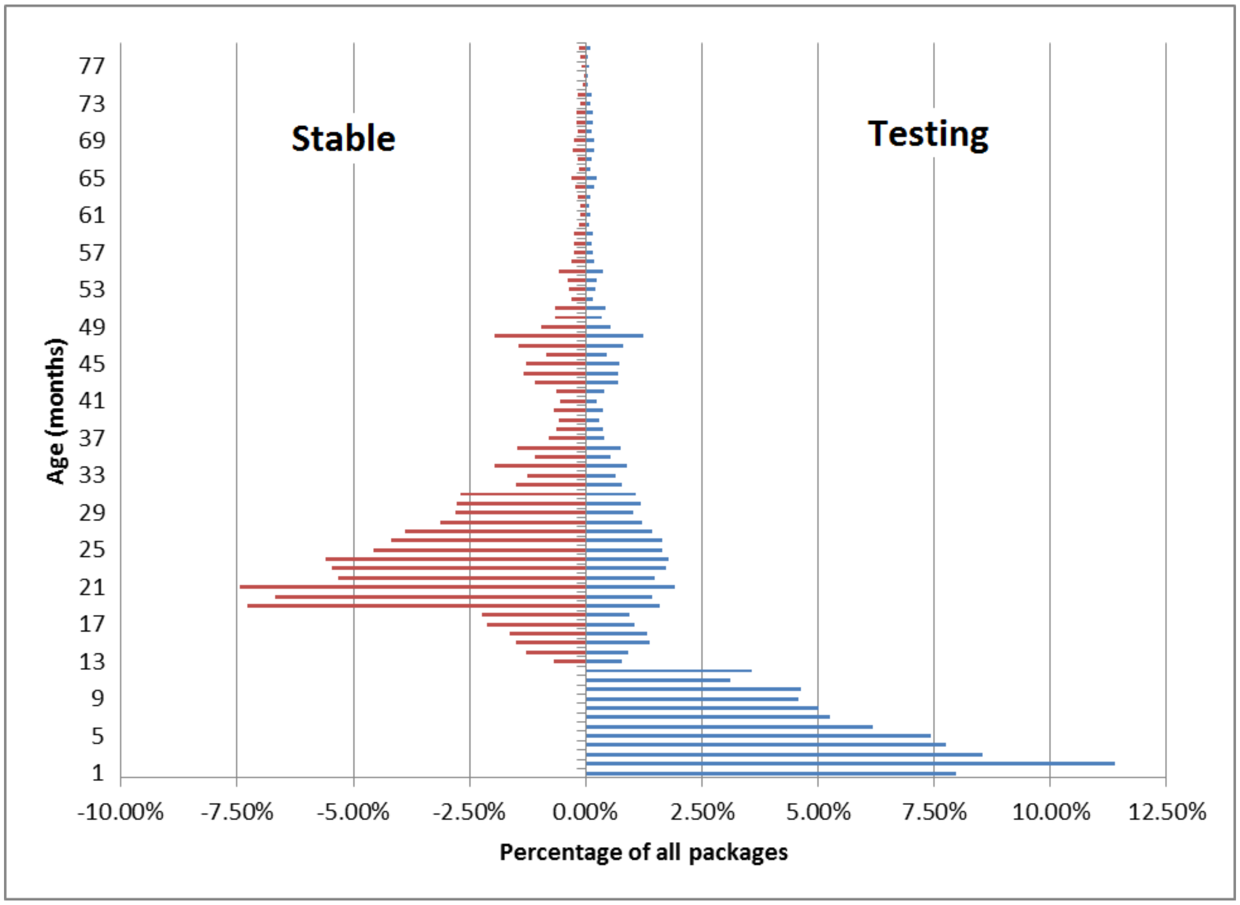

Figure 4.8: Age of Testing vs. Stable as of Feb.1, 2012

4.4.2 Debian Package Maintainers

Package age is just one part of the package biography; another attribute that we examine are the people who maintain the packages: the Debian developers or maintainers. There have been a number of studies regarding maintainers. One paper looks at how volunteer work affects quality assurance using Debian as a case study [27]. Another one performed by Robles, Gonzalez-Barahona and Michlmayr examines how Debian developers differ when compared to developers in a corporate environment [32]. They concluded that there are no formal ways of forcing a developer to assume any given task, but voluntary efforts seem to be quite stable. The mean life of volunteers in Debian was found to be larger than in many software companies, which has a clear impact on the maintenance of the software.

All Debian packages must have the name and email contact of its maintainer embedded in the package metadata. Most maintainers use their @debian email address when creating a package; so we assume that each unique email recorded maps to a separate maintainer. In order for Debian to grow and sustain itself, there must be a healthy number of developers working on the system. Figure 4.9 charts the number of new maintainers on a yearly basis. There is an average of 360 new maintainers per year over the last ten years. The number of new maintainers seems to be trending down; but it is hard to draw conclusions on the consequences of this type of downward movement. We know that the number of packages being uploaded annually is relatively steady and there are no sharp declines in maintainer activity. Therefore no apparent negative consequences are seen, but if this trend continues into the future, then we know that Debian is having a problem attracting new developers. Moreover, 2001 was a good year for Debian with an explosion in package upload counts (Figure 4.2). To facilitate the man power required to produce so many packages, there was also a huge influx of maintainers to Debian that same year. This is evident in Figure 4.9, which shows that the count for 2001 more than doubled the previous year's new maintainer count.

We have the data for the rate of new incoming maintainers, but what about the rate of maintainers leaving Debian? This is a more difficult question to answer since Debian developers are all volunteers and are not required to declare retirement. Thus we try to answer this question from the perspective of maintainer inactivity. Figure 4.10 shows that within the past year, 1793 developers created and uploaded a package. We will assume that these are the active developers of Debian. In the year before that, 372 developers became inactive since the last package they uploaded was over a year ago. This type of measurement is not precise because there is no telling when a developer might become active again, but they do give a good idea of the number of maintainers who stop working on new packages. Over the past ten years, the average number of maintainers who have

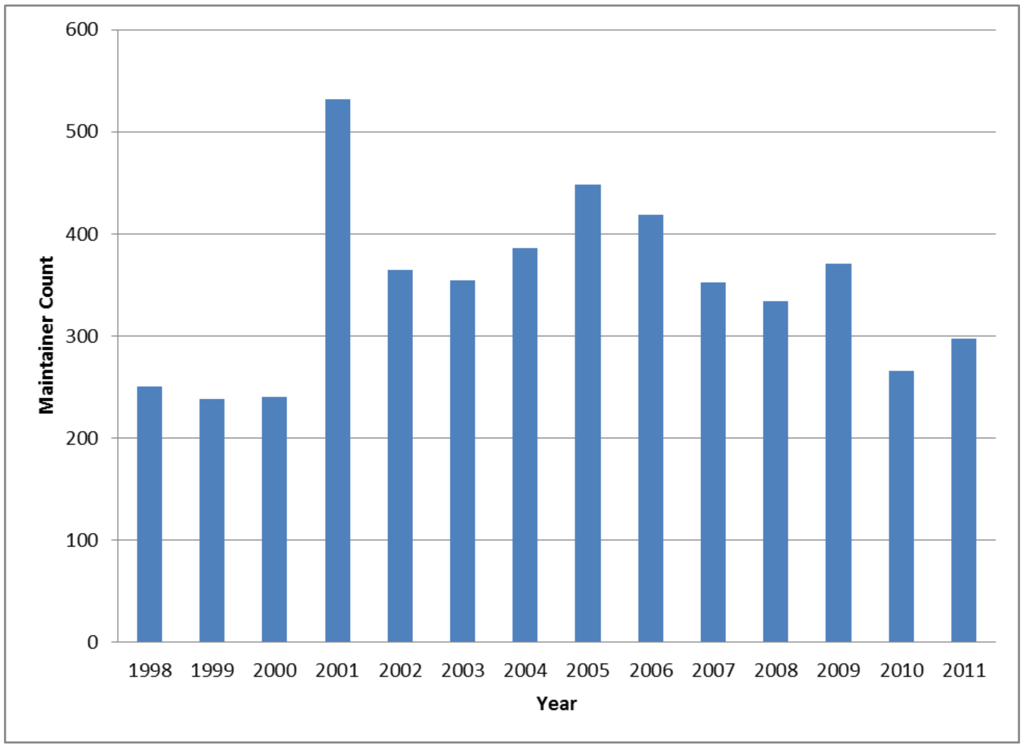

Figure 4.9: New Incoming Debian Maintainers

become inactive per year is 290. Comparing this with the influx of new maintainers, we have a net gain of 70 maintainers each year.

We know that there is an increasing base of active Debian maintainers, yet the number of packages produced per year has been quite stable. There is no discernable correlation between maintainer count and upload count. In order to figure out why, we take a closer look at the number of packages each maintainer uploads in Figure 4.11. The graph in Figure 4.11 is very top heavy; a small number of maintainers upload a tremendous number of packages. Figure 4.12 offers a zoomed view of the number of package uploads performed by just 1% of the Debian developer population. This group of 49 maintainers is a fraction of the nearly 4900 maintainers that have contributed to Debian, yet they account for over 27% of all uploaded packages. With such a high amount of activity, these maintainers are likely a team of people working together to produce the packages. The maintainer team working on the Perl packages is the busiest with 9996 packages uploaded. It would be interesting as future work to study how these groups function to see how and why they update packages at such a rapid pace.

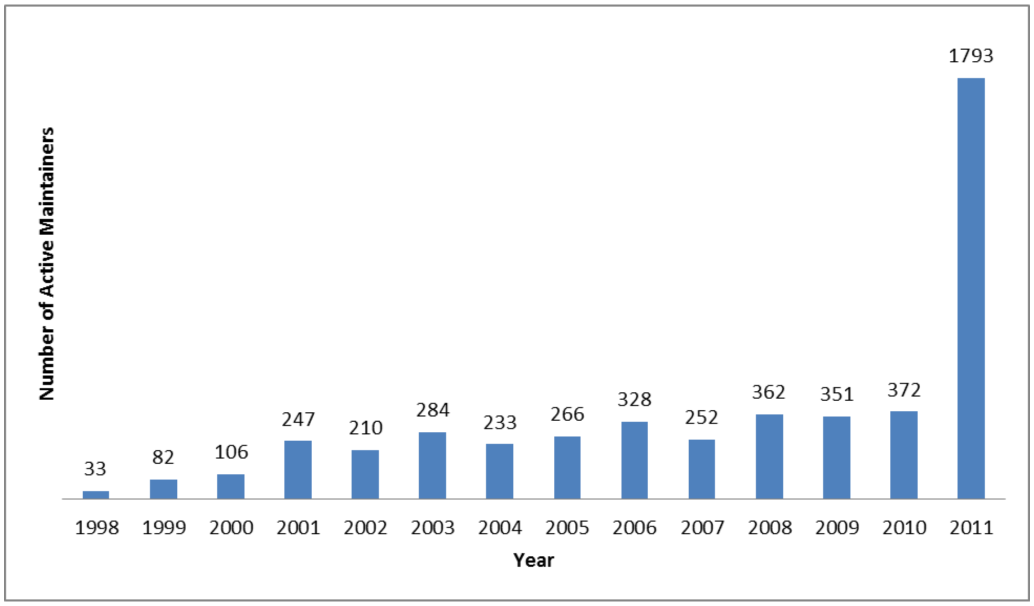

Figure 4.10: Last Recorded Activity From a Debian Developer

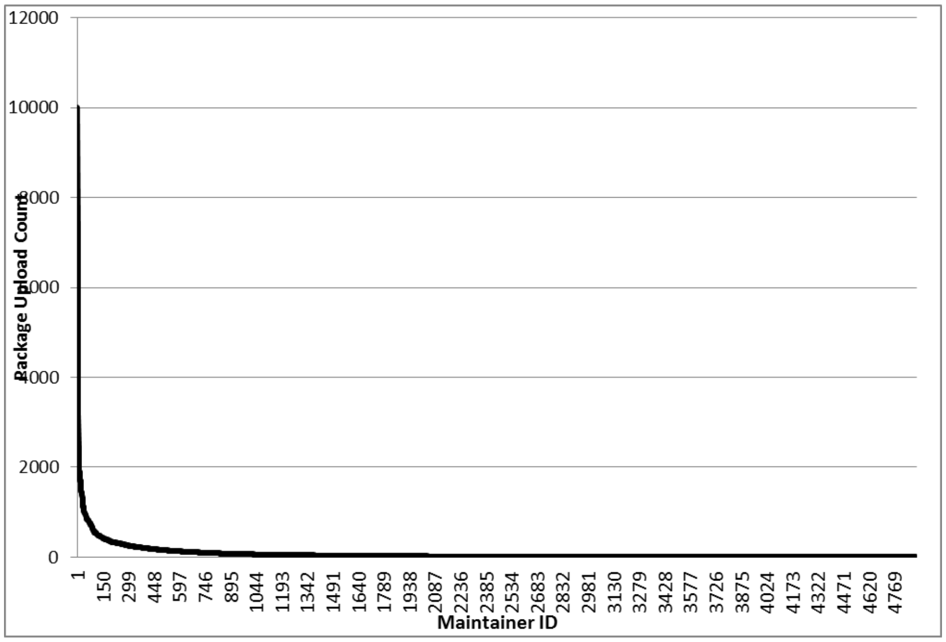

Figure 4.11: Total Package Uploads by Maintainers

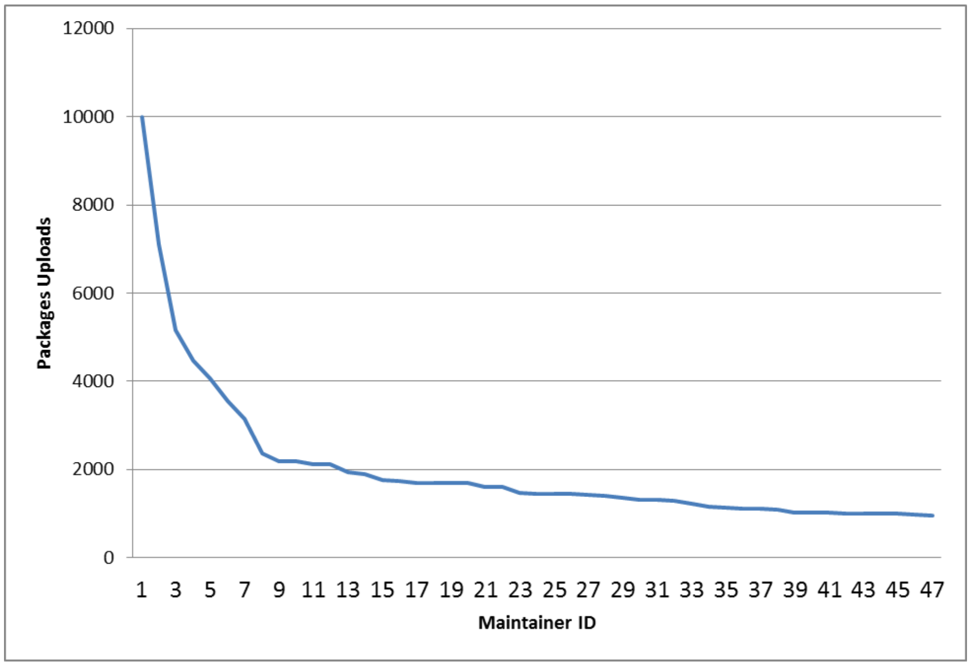

Figure 4.12: Package Uploads From Top 1% of Contributors

4.4.3 Packaging Bugs

During a package lifespan, another key attribute from the package are bug reports. Bug reports are handled through the Debian bug tracking system (BTS). The BTS consists of a set of scripts which maintains a massive database. As of February 2012, there have been over 630,000 bugs filed. The primary purpose of the BTS is to get packaging bugs noticed and fixed. If a bug has an upstream error, then it will get forwarded to the upstream developers who wrote the source code. Surprising, only 35,000 of the 630,000 bug reports have been forwarded which means the remaining 595,000 bugs are related to the Debian version. Figure 4.13 provides a look at the bugs reported per year. We noticed that since 2004, there has been a decline in bugs reported per year. A downward trend could just mean that less bugs are being reported, but it could also signify that developers are getting more experienced with creating packages and are producing higher quality packages overall.

The BTS automatically archives all fixed bugs if there is no longer any activity within the bug report after 28 days. Currently, 547,000 of the 630,000 bug reports are archived and cannot be opened again. Unfortunately the UDD loses some of the bug report data once it gets archived. More specifically, we are no longer able to tell from which distribution

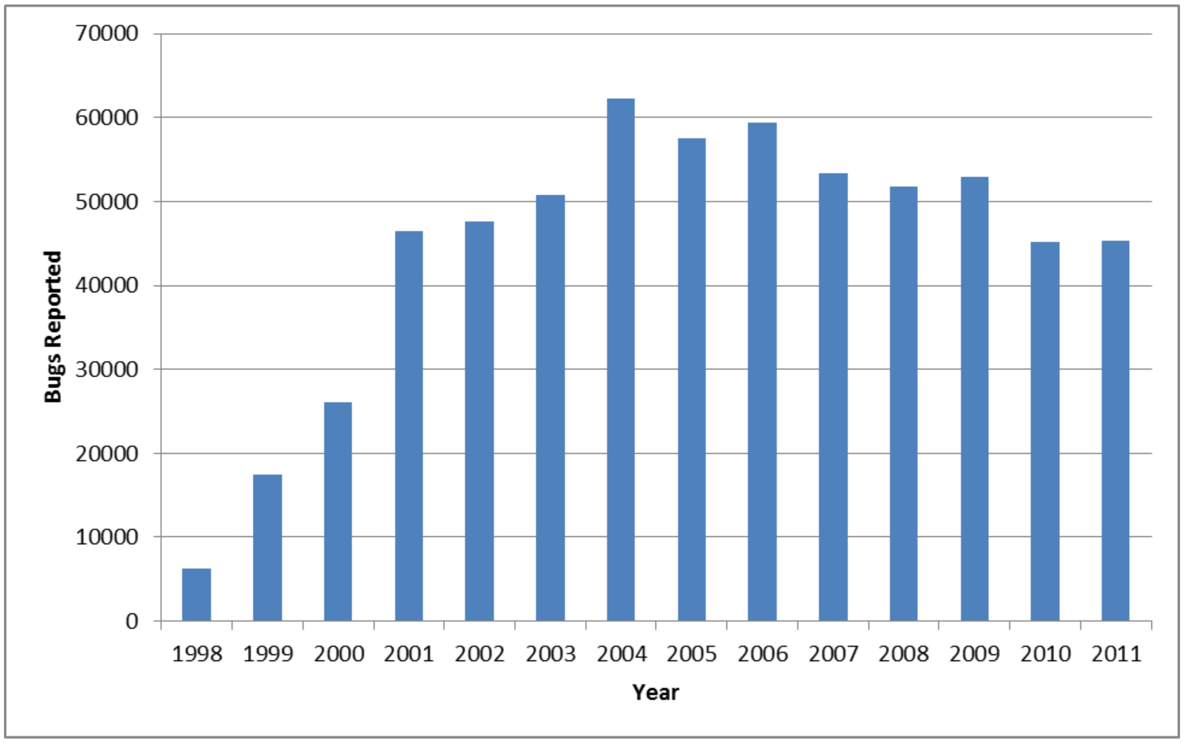

Figure 4.13: Bug Reports by Year

the bug report was filed for. The active bug reports still contain this piece of information and so we can chart the number of bugs across the three distributions. It is widely believed that the *stable* release of Debian is solid while *unstable* is more suitable for those who don't mind some instability in their system. Figure 4.14 provides the numbers of the active bug reports based on which distribution they are affecting. It is surprising to see that there is not a larger difference between the three distributions. One explanation could be that we are not using the archived bug reports. We know that the lifespan of unstable packages are usually quite short and that could also be the case for bugs that only affect unstable. This would mean that all of these reports that affect unstable would have already been archived because a newer version of the package would have replaced the older version. This is only a hypothesis, but could offer some insight into why the number of active bug reports is similar across the different distributions.

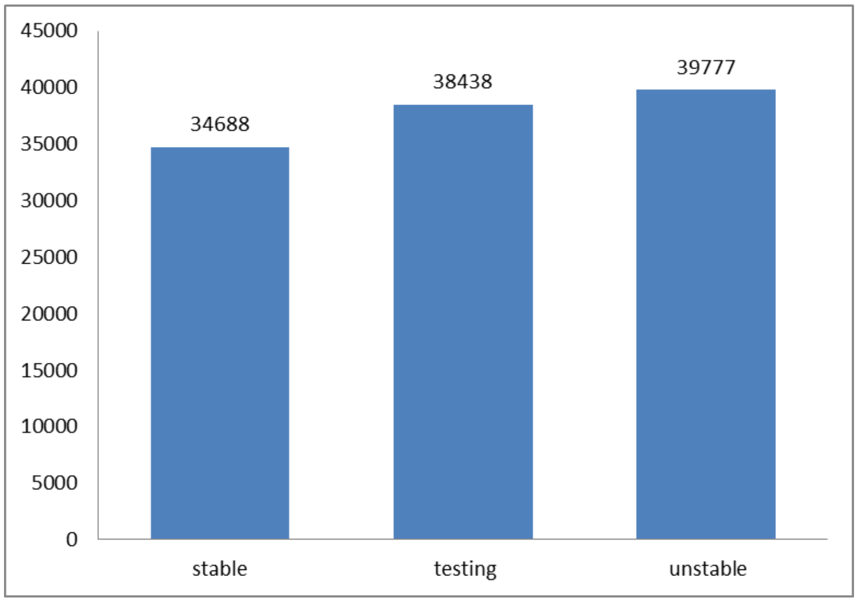

Figure 4.14: Active Bug Reports by Distributions

4.4.4 Package Popularity

The people at Debian like to keep track of the more popular package families so that they can include them on physical media to distribute to users. The method they use to gauge popularity is through the popularity contest package that users are asked to install. This package works by running a daily cron job that records new package installations, recently used packages and recently updated packages. The data is then uploaded onto the Debian servers once a week. Unfortunately only the present popularity data is available and we were unable to obtain the historical popularity information on the package families. It would have been interesting to chart any rises and falls in popularity compared to factors such as updates and bugs. Regardless, the data on the current popularity of packages is still valuable.

There have been two published papers that analyze package popularity in great detail in order to try to relate it to bug count. The first, performed by Davies, Zhang, Nussbaum and German wanted to determine a relation between bugs and popularity, but were unable to do so [9]. However, they remain optimistic that a correlation can be identified through future work. In a separate study, Herraiz, Shihab, Nguyen, and Hassan concluded that only very popular packages contain a high level of defects and that there is a popularity bias; defects will only be discovered if people use the software [21].

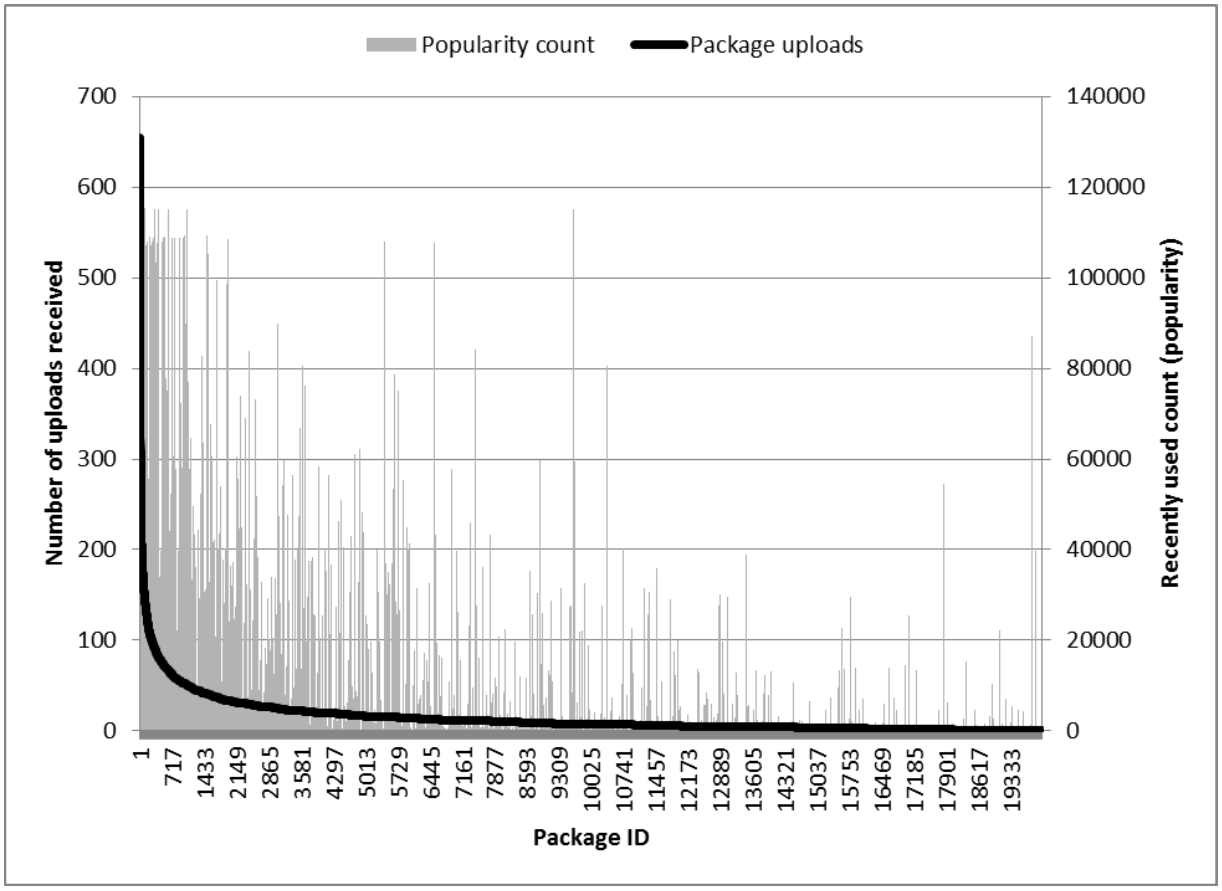

Figure 4.15: Popularity vs. Upload Count

We try to make use of popularity from a different angle by trying to correlate it to package updates. Figure 4.15 shows that there is some correlation between the number of times a package is updated and its usage count. A calculation of the Spearman correlation coefficient gives a value of 0.37 which means that there is some positive association between the two, but a deeper analysis needs to be completed before we can make firm conclusions. A previous study shows that Debian bugs highly correlate with package updates [9], which makes sense because new code often means new defects. This creates an interesting dynamic between popularity, updates and bugs. Could it be that it is not always popularity that is causing higher bug counts, but rather the frequent updates? More work needs to be done to determine how these three variables relate to each other and whether we can identify a cause and effect rather than merely discover the existence of a correlation.

4.5 Summary

In this chapter we examine the evolution of Debian as an example of a large scale software collection. A longitudinal study was performed in order to analyze the life and death of Debian source packages. By looking at software collections from the perspective of their individual parts (package biography) and their behavior over time, we gain an understanding of how the collection evolves as a whole. With the birth rate of packages, we noticed that they have been stable at around 30,000 package uploads per year. Apparently, Debian developers are conscious of when the next stable release will occur and will shy away from creating new packages when the *testing* distribution is frozen. When examining package death, we observe that the majority of packages do not get removed outright, but instead are replaced by a newer version. The average life span of a package is short with a high chance of infant mortality. The large number of incoming and outgoing packages suggests that Debian is evolving at a very rapid pace. Packages are consistently getting updated and each day brings numerous changes to the system as a whole. However, this can be the cause of defects which creates system instability. To solve this problem, Debian releases a *stable* version every two years which is free of all known release critical bugs. To help us further understand the evolution of this vast software collection, we review four attributes resulting from the lifespan of source packages. These are package age, package maintainers, package bugs and package popularity. Package age tells us that of all the packages uploaded to Debian; over 88% do not live to be more than one year old. We also looked at the age of packages in *unstable*, *testing* and *stable* distributions separately and found that *testing* mirrors *unstable* except for the very new packages. While packages in *stable* remain old until a new release is made. Investigating the people who upload and maintain packages reveals that the number of active maintainers grows every year. It also reveals the fact that a large volume of package uploads belong to a very small subset of maintainers. Packaging bugs were shown to be on the decline since 2004. Finally, we learn that package popularity has a small correlation with update rate. Popularity plays a role in how the individual parts of a software collection evolve, which in turn affects how the collection as a whole evolves. In this chapter, we present numerous trends about the Debian software collection. We learn that software collections grow in size regardless of how large they have already become and existing packages are constantly updated to provide the user with the latest software.

Chapter 5

Architecture of Debian

The previous chapter explored the evolution of Debian by examining source packages. This chapter will look at how the binary packages interact with each other to understand the architecture of Debian. Binary packages contain dependencies on other binaries; this is how we determine the interactions among packages. To effectively describe Debian's architecture, we came up with a set of patterns that can be used to classify the packages. Figure 5.1 gives an overview of the different package patterns. At first, packages can either be classified as part of the *layered* pattern or the *hermit* pattern depending on the number of incoming and outgoing dependencies. Once found to be a part of the *layered* pattern, the package can be further classified into a specific layer. Two automatic classification techniques are also presented to provide the number of packages belong to each pattern and layer.

Packages that belong to the *application layer* are different from the other layers. The dependencies among packages within this layer can provide a lot of information about how a particular application is structured. The different ways of structuring applications is described through five subpatterns. These subpatterns are design choices made by package maintainers. By introducing these patterns, we can provide inexperienced maintainers with information about patterns to use and ones to avoid. Finally, this chapter will also explore other ways that packages can interact with each other. At the code level, research by others has been done to show that release history can help identify coupling among software elements [12]. We extend this idea to software collections to see if there are hidden coupling among packages that are not already expressed as dependencies.

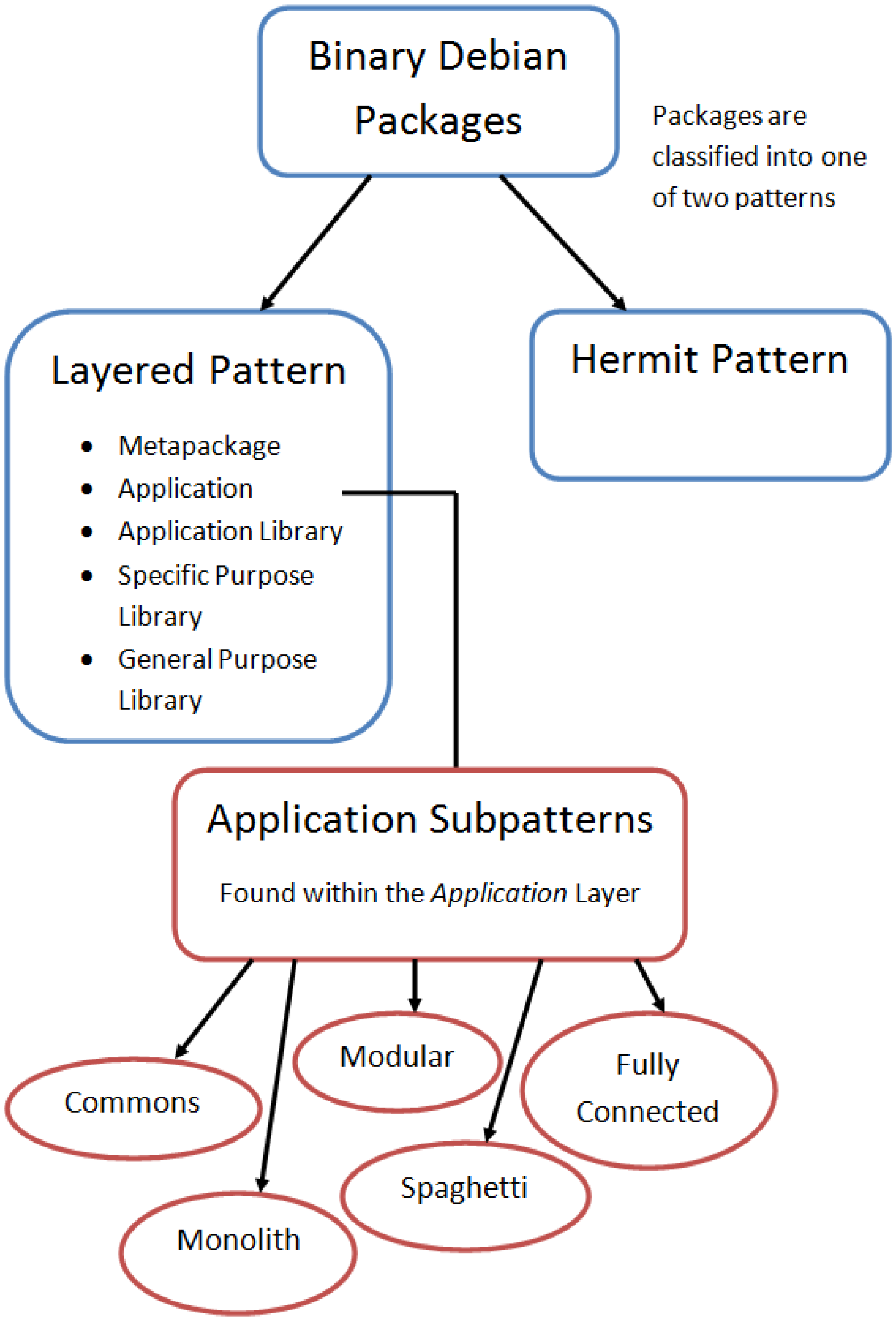

Figure 5.1: Overview of Package Patterns

5.1 Visualizing Debian Packages

Debian is a huge system filled with a large number of packages. The nature of the dependencies in Debian lends itself well to visualizations in the following way. Each package represents a node in a graph and each dependency represents an arrow to produce a directed graph. Drawing the directed graph helps to create a firm intuition of how Debian packages interact with each other. To manually draw and organize over 30,000 packages with 120,000 dependencies is close to impossible and so we relied on software to help with the visualization. We take the dependency data from each package and then consolidate the data into a single file. The data was fed into LSEdit from the SWAG lab at the University of Waterloo. LSEdit is a graph visualization tool for viewing, manipulating, querying, laying out and clustering of large graphs [38]. However, due to the sheer volume of packages and dependencies; the output was incomprehensible.

After this unsuccessful attempt at visualizing Debian, we decided to scale down and look at individual applications and their dependencies. A tool called debtree developed by Frans Pop extracts all the dependencies for a particular package then then outputs a graph. These graphs are large and messy, but much more manageable than trying to look at the entire set of packages at once. Figure 5.2 is an example of a directed graph illustrating the dependencies required for the photo editor GIMP. The top most node is the GIMP package. It has dependencies to a set of packages where the names also begin with GIMP. This leads us to suspect that these packages have a close relation with GIMP's functionality. At the bottom of the graph, common packages such fontconfig or x11-common show up. At first, the graph seems extremely large and difficult to decipher, but looking up the functionality of each package involved provides insight into why these packages are required to run GIMP. Visualizing packages and detailing the functionality of their dependencies allows us to understand the inner workings of what makes the package run. This procedure was applied to a variety of other packages. After examining 20 other packages in detail, we noticed they all shared one of two basic architectural patterns (explained in the next sections).

5.2 The Layered Pattern

From Figure 5.1, we see that packages can be classified into the *layered* pattern or *hermit* pattern. Packages that belong to the *layered* pattern are the individual pieces that contribute to the overall application. We found that it is possible to sort a package contained in the *layered* pattern into one of five layers: metapackage, application, application library,

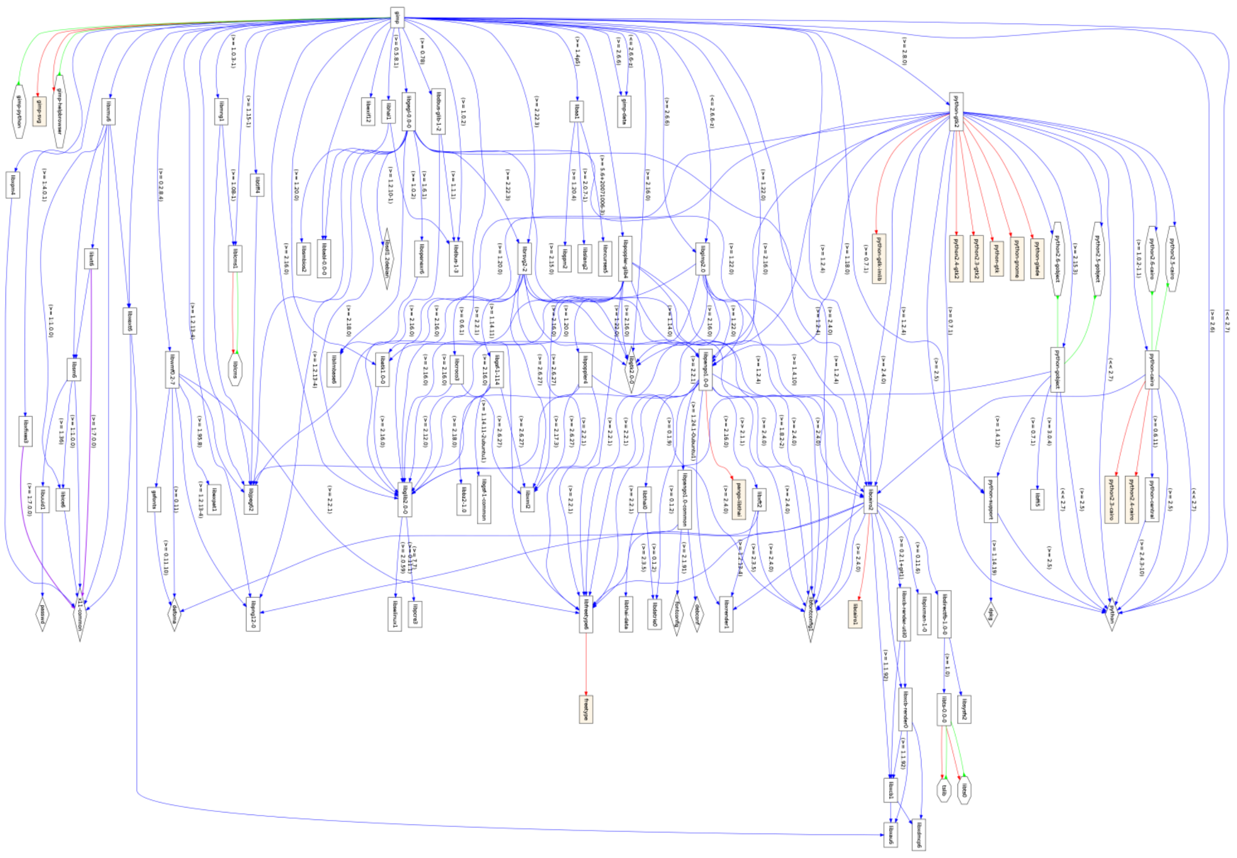

Figure 5.2: Dependency Graph of the GIMP Package

specific purpose library and general purpose library (Figure 5.3). The arrows in Figure 5.3 represent the allowed dependencies between each of the layers. We will proceed to describe how these groups are layered. At the top we have the metapackage layer; which only exists for the very large applications. A metapackage depends on a collection of applications in the same software suite such as a desktop GUI like gnome, or a game collection pack. A metapackage is a package which doesn't have any files to install, but depends on other files to make sure they are installed. Since metapackages are the top layer, they normally do not get referenced by any other package.

Packages that belong to the application layer are normally individual applications that end users would use. Application packages can possibly have dependencies to each of the three layers below it. Just below the application level are application libraries.

These are application specific libraries that are mainly suited to be referenced by the associated application packages. When Debian maintainers are packaging software, a couple of factors will motivate them to split up the application level package into one or more application library packages. For instance when there are multiple applications in the same software suite, often there will be shared functionality. Instead of duplicating this functionality into various application packages, maintainers can opt to create separate library packages that are installed separately. Another reason why maintainers want to create an application library is if they feel it might be of some use to other packages. There may be API calls or other functionality that new packages can build upon. Next in the hierarchy are specific purpose libraries. These are widely used libraries that provide a specific functionality. For example if the application uses XML, it is likely to reference a package called `libxml2` in order to parse XML documents.

At the bottom layer we have general purpose libraries. These are the packages that provide the basic functionality required by various applications and may be referenced by the many other packages. Packages such as `libc6` which contains the standard C libraries or fontconfig which provide system-wide font configurations are examples of general purpose libraries.

In order to more fully illustrate the layered pattern of Debian packages, the popular application Open Office will be used as an example. Open Office is an office productivity suite similar to Microsoft Office. When one installs the `openoffice.org` package, there are 43 other packages that must be installed to satisfy 165 dependencies between them. Listing all the dependencies and classifying each of the packages based on the layered pattern would be too lengthy. Instead, we will use Open Office to provide examples of packages that fit in each layer as shown in Figure 5.4.

At the metapackage level, we have `openoffice.org`. This package contains (depends

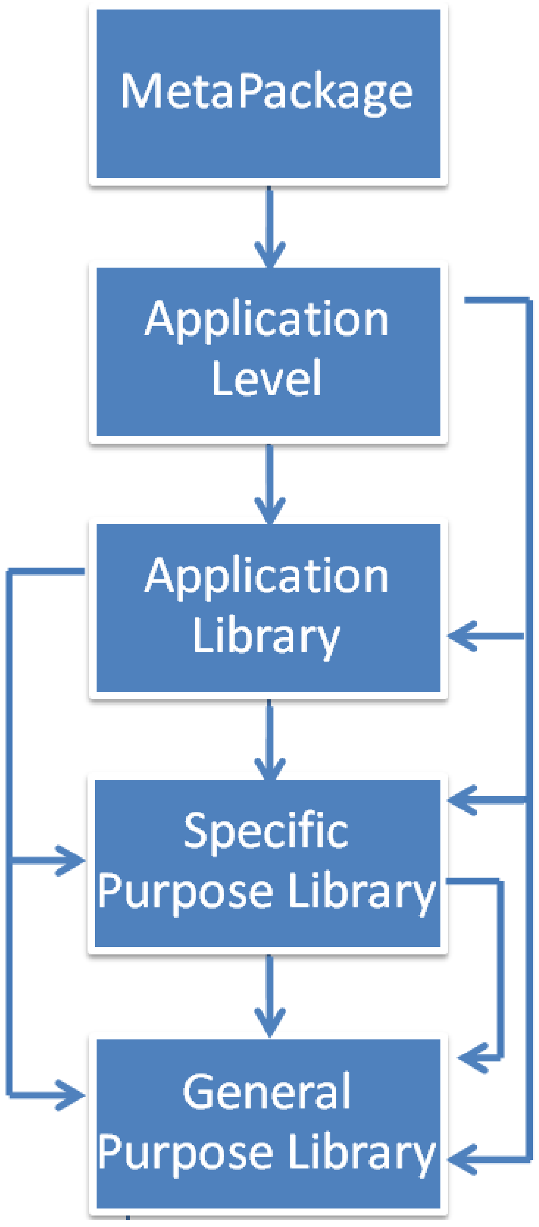

Figure 5.3: Layered Package Pattern

on) the entire office productivity suite. Some examples of the dependencies from `openoffice.org` are Writer, Draw and Calc. Writer is a word processor similar to Microsoft Word, Draw is a paint too used to create graphics and Calc is spreadsheet application. Each is shown in Figure 5.4 in the second layer. These application packages depend on `libwpd8c2a`, `libwpg-0.1-1` and `libhyphen0`. `Libwpd8c2a` helps handle the wordperfect format, `libwpg-0.1-1` imports/converts wordperfect graphics and `libhyphen0` is a hyphenation and justification library. For the specific purpose libraries, we have `libcurl3-gnutls` which is a portable multi-protocol file transfer library, `libhunspell-1.2-0` which is a spell checker with a morphological analyzer and `libgstreamer0.10-0` which is a media streaming framework that can process sound and video. In general, the application libraries are the packages which are only referenced by their associated application level package, while specific purpose libraries can be referenced by a larger number of applications. For example, `libhunspell-1.2-0` is also used by the package "mine", a comprehensive text editor.

The general purpose libraries normally deal with lower level complexity and are widely depended upon. In the Open Office example, we have `libc6` which contains the standard C functions. `Libuuid1` generates 128-bit unique ids and `zlib1g` is a compression zipping library. This example using `openoffice.org` provides a deeper look into the content and role of each layer.

5.3 The Hermit Pattern

Besides the five package layers, there is an additional group of packages that do not fit the layered model. We label these as the *hermit* packages. For a package to be classified as a *hermit*, it must have no dependencies to other packages nor dependencies from any package. Many packages that belong to this group contain only data or text. However, it could also be the case that the package is a lightweight application. In fact, a good number of packages in Debian do not contain a single dependency or reference. Hermits are isolated packages that do not interact with other the other packages of the system. For example, the package `openoffice.org-dev-doc` is a hermit as it contains no dependencies or references. This package is merely documentation on how to use the `openoffice.org` SDK.

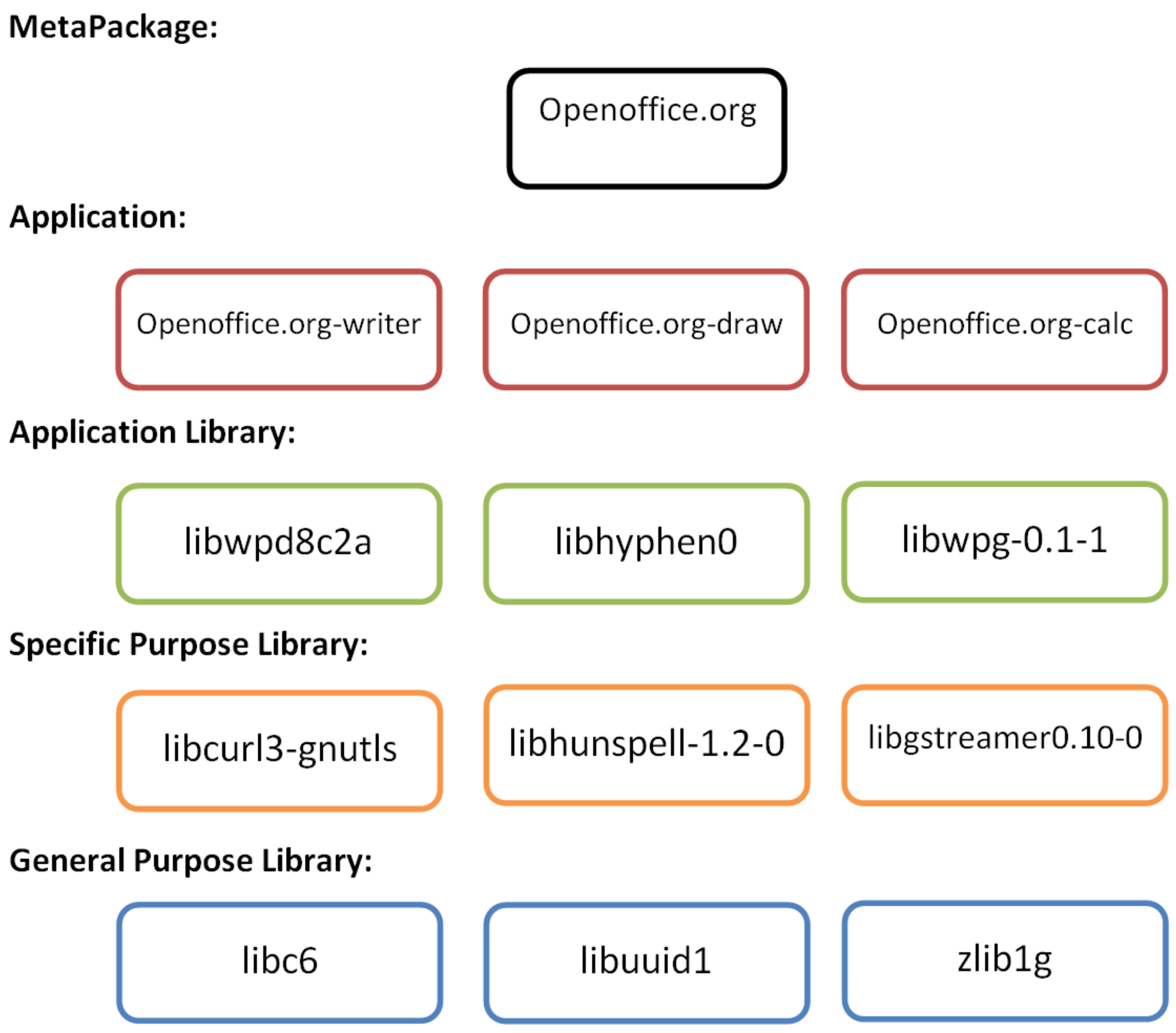

Figure 5.4: Layered Patterns of Open Office

5.4 Automatic Classification of Package Layers

We have described the *layered* pattern and the *hermit* pattern to show how packages are organized. Now we will explore the overall composition of Debian. In order to tackle this problem, we investigate the hermits and the five layers in more detail; more specifically, the makeup of each layer and how they interact with one another. We take the entire set of Debian packages and assign each package to a layer or as a hermit. Doing so provides an overview of Debian's structure. For instance, if the majority of packages are general and specific purpose libraries, we could categorize Debian as a bottom heavy software collection. Alternatively, if we discover that most packages are at the application level, then Debian would be more of a top heavy software collection.

In Figure 5.4 we manually classified packages into one of the five layers by using the description of the packages. But to classify over 30,000 packages by hand is not feasible and so we sought out a way to accomplish this task automatically. Throughout the process of manually classifying, we noticed many of the same general purpose libraries were referenced over and over again. At the same time, packages at the application library level and above were seldom referenced. This observation led to the counting of the *In, Out, Transitive In* and *Transitive Out* degree of all the Debian packages. The *In* degree of a package is the number of other packages that directly depend on it. The *Out* degree counts the number of direct dependencies to packages. *Transitive In* counts the number of packages that directly or indirectly depends on it. For example, if Package A depends on Package B and Package B depends on Package C, then Package C would have an *In* degree of 1, but a *Transitive In* count of 2. *Transitive Out* refers to the total count of packages that are required to be installed in order to satisfy all dependencies. Now that we have the list of all packages along with their *In, Out, Transitive In* and *Transitive Out* count, we use two methods to classify the packages. The first method is through simple sorting and filtering. This is an ad-hoc procedure with the following steps:

1. General Purpose Libraries:

   We first sort the list of packages based on the *Transitive In* count since we observe that general purpose libraries are widely referenced. At around the 500 count, packages no longer resemble general purpose packages and so we decide to set the threshold at this point. Table 5.1 lists a few examples of *Transitive In* counts for general purpose libraries. `Libc6` has a count of 19,724, `libuuid1` is 595 and `zlib1g` is 11056. When sorting all the packages based on *Transitive In* count, we notice that the high counts are all general purpose libraries. At around the 500 count, the packages we inspected no longer seem to function as a general purpose library, therefore we set the threshold at this number.

2. Hermits:

   Now, we begin to identify *hermits*. These are the isolated packages and so packages with a combined In and Out degree of 1 were tagged. We set the threshold of the *In* and *Out* degree to 1 instead of 0 is because small lightweight applications still depend on the standard C libraries in order to run. These applications with the single dependency are still considered hermits since there is no meaningful dependency structure.

3. Metapackage and Application Level:

   With the hermits out of the way, the packages that have an *In* count of less than 5 and a *Transitive Out* count of greater than 30 will be considered a part of the metapackage layer or application layer. Applications normally tend to have few references, but depend upon many other packages. Metapackages and applications are at the top layer and therefore depend on application, specific and general purpose libraries to function. Hence, these top layer packages will always have a larger *Transitive Out* count than library packages. We also decide to combine the metapackage and application level packages as one group because metapackages are not encountered as often. Moreover, it is almost impossible to distinguish the difference between these two groups based on the *In/Out* degree.

4. Specific Purpose Libraries

   To filter for specific purpose libraries, we look for packages with an *In* degree of greater than 15 or a *Transitive In* count of over 100. From applying our previous three steps, the packages we are left with now also have a *Transitive In* count of less than 500 and a *Transitive out* count of less than 30. This fits the description of a specific purpose library because they are less often referenced than by general purpose libraries and normally has a small *Out* degree.

5. Application Libraries

   The remaining packages are now labelled as Application libraries. These packages have an *In* degree of less than 15 and a *Transitive Out* degree of less than 30. Application libraries tend to have small *In* and *Out* degrees due to their limited use.

Overall, the described procedure is ad-hoc and only done to provide an estimation of the size of each package layer. Table 5.1 summarizes the *In, Out, Transitive In* and

Transitive Out degree count for the packages listed in Figure 5.4. Most of the manually classified packages also appear to be classified to the same layer using the ad-hoc method. The exceptions are `openoffice.org-writer` and `libhunspell-1.2-0` which is labelled as "incorrect" in Table 5.1. Both `openoffice.org-writer` and `libhunspell-1.2-0` were mistaken as application libraries. From the Open Office package examples, the ad-hoc method was able to correctly classify 10 out of the 12 packages giving us an accuracy of 83.3%. Manually identifying package layers and comparing it to our auto classification techniques is one way to validate our algorithms. We can also tweak and refine our ad-hoc algorithm if we knew the correct layer every package belonged to. Since manually identifying package layers is time consuming and the accuracy of our algorithms is not critical, there was not a significant amount of manual identification done.

Our second technique used to group the packages was to apply a clustering algorithm to the entire set of Debian packages. Weka is an open source tool that contains a collection of machine learning algorithms [20]. We use one of its available clustering algorithms help us classify packages into layers or as hermits. The k-means algorithm was selected to perform the clustering. K-means clustering is an unsupervised machine learning algorithm for partitioning a dataset into k separate clusters where instances (of the dataset) in each cluster are similar to each other [19]. The similarity measure in this particular algorithm is defined as the Euclidean distance between two instances. In our case, each instance consists of the *In, Out, Transitive In* and *Transitive Out* degree. The reason why we chose k-means is because it is a simple and efficient algorithm to find the different behaviors (clusters) in a dataset. The use of this algorithm is able to effectively group different packages together based on their *In, Out, Transitive In* and *Transitive Out* degree. There are other algorithms and techniques to consider, but the simplicity and power makes k-means an attractive choice. We set the cluster number parameter to 5 to correspond to the number of layers and then we let the algorithm run. The result of the k-means clustering with 5 clusters is shown in Table 5.2. The output produced from k-means is 5 clusters of packages that the algorithm deems to be most similar to one another. Weka does not output which cluster a package belongs to. Therefore to extract the cluster that k-means assigns to each package, some modifications to the source code had to be made. The added source code prints out all the package instances and their corresponding cluster in a text file.

From examining the connectivity of the packages in each cluster, we notice a common theme in each cluster similar to our ad-hoc procedure. Using a similar style of reasoning to the ad-hoc method, package layers are assigned to clusters as follows. Packages that have a high *Transitive In* degree were clustered together and we therefore labeled this cluster as the layer for general purpose libraries. The cluster where the majority of packages exhibited zero or very little connectivity is labeled as the hermits. Application layer packages were matched up with the cluster that had a mostly high *Transitive Out* counts and a low *Transitive In* count. The two remaining clusters were chosen to be either the application libraries or specific purpose libraries. The cluster with the higher *Out/Transitive Out* counts and lower *In/Transitive In* was labeled the application library layer. The cluster with the opposite characteristic was labeled as the specific purpose libraries. The characteristics of the clusters output by the k-means algorithm were each notably different, and thus made it easy to have each cluster represent a layer.

The numbers in Table 5.2 reveal Debian's composition in terms of package patterns (and layers). The most striking aspect is that over half of the packages in Debian are hermits. These are packages which are isolated from the rest of the group and do not contain any meaningful dependencies. On the other hand, the other half of Debian participates in a top heavy *layered* pattern. Of the packages that belong to the *layered* pattern, a majority are a part of the application level layer. This suggests that Debian contains a high number of application software as opposed to supporting packages. While library files are often referenced in Debian, there are not a large number of them.

Interestingly, the ad-hoc method and the k-means clustering produced similar results as seen in Table 5.2. However, one difference lies in the specific purpose library count as there are nearly 10 times more found with the ad-hoc procedure. The way that the clustering algorithm works could explain why the k-means cluster that represents the specific purpose libraries is small in size. The algorithm clusters similar packages together, therefore many of the specific purpose libraries that we found in the ad-hoc method were considered to be more similar to the packages in the other clusters. Regardless, this breakdown provides an estimate of the composition of Debian and the kind of packages available to the user.

5.5 Application Subpatterns

Dependencies from the application packages on certain libraries are linked automatically; saving the Debian maintainer time from having to specify them manually. Scripts can be used to detect the set of packages required to fulfill the library dependencies of a program [28]. The minimum set of Debian packages to satisfy the requirements are found by using the list dynamic dependency (ldd) command in Linux. The Shlibs scripts in Debian creates managed maps to translate the output into a list of dependencies. The dependencies from the applications to the specific and general purpose libraries are not as interesting to study as dependencies within the layers themselves because they are more a product of the code and less a product of the maintainer's design. For instance, dependencies between the

Table 5.1: *In, Out, Transitive In* and *Transitive Out* Degree Example

Package Name	**Out**	**In**	**T. Out**	**T. In**	**Ad-hoc Classification**
Application					
openoffice.org-writer	11	28	122	44	application lib. (incorrect)
openoffice.org-draw	8	2	121	20	Application
openoffice.org-calc	8	1	125	17	application
Application Library					
libwpd8c2a	3	12	4	71	application library
libhyphen0	1	2	3	85	application library
libwpg-0.1-1	3	6	4	33	application library
Specific Purpose Lib					
libcurl3-gnutls	7	101	26	312	specific purpose library
libhunspell-1.2-0	3	7	4	500	application lib. (incorrect)
libgstreamer0.10-0	3	84	8	370	specific purpose library
General Purpose Lib					
libc6	1	10414	2	19724	general purpose library
libuuid1	2	76	17	595	general purpose library
zlib1g	1	1638	3	11056	general purpose library

Table 5.2: Debian's Package Composition

Package Group	**Ad-hoc method**	**k-means clustering**
Hermit	11483 (50.9%)	12473 (55.3%)
Application Level	6225 (27.6%)	8240 (36.5%)
Application Library	2016 (8.9%)	1426 (6.3%)
Specific Purpose Library	2602 (11.5%)	276 (1.2%)
General Purpose Library	233 (1.0%)	145 (0.6%)

metapackage and application levels reveal how a particular application is packaged for Debian. Trying to find patterns of design through the dependency text is difficult; thus making some form of visualization necessary. We previously tried visualizing Debian's dependencies, but dependency graphs such as GIMP from Figure 5.2 make us doubt the utility in doing so. However, we hypothesize that the visualization would be more effective if we only visualize dependencies to and from the *application* layer. A large number of dependencies are ones to the library packages. Without these dependencies interfering in the visualization, the dependency graph will appear much less cluttered. Unfortunately there is no way to display all the dependency graphs of the *application* layer packages in this thesis due to the sheer number of them. Instead, we describe the trends and patterns in packaging design from the study of these dependency graphs. We noticed that a majority of packaged applications follow a good design principle since they allow for customizable installations and efficient updates. However, there were also few poorly designed packages that caught our attention. In the latest unstable release of Debian, we have noticed that some of these packages no longer exist and have been replaced by packages that are designed differently.

The *layered* and the *hermit* pattern describe how packages are structured in relation to the overall software collection. In contrast, the patterns described in this section deal with only those at the *application* level (Figure 5.1). These patterns are the manifestation of how the Debian maintainer designed and laid out the particular application. Hence, we will call these subpatterns and anti-subpatterns. Subpatterns are general solutions to packaging software, while anti-subpatterns demonstrate the weaker methods used to package software.

5.5.1 Application Subpattern: Commons

The first of the five application subpatterns we present is the Commons pattern. This pattern occurs when the maintainer factors out parts from one or more packages that are reused. For example in Figure 5.5, `xpilot-ng` is an application with two different clients called `xpilot-ng-client-x11` and `xpilot-ng-client-sdl`. Instead of duplicating code in each of the clients and related packages, it is factored out into a separate package called `xpilot-ng-common`. Saving space is important and sometimes necessary for embedded or other low resource machines. Often times, these package names end in "common", hence we label this the Commons pattern. Figure 5.6 demonstrates the same pattern in a different context. `Abiword-common` factors out the "architecture independent" parts from abiword. This means that `abiword-common` does not need to be compiled separately for each of the different chipsets and thus saving Debian the extra costs associated with distributing and

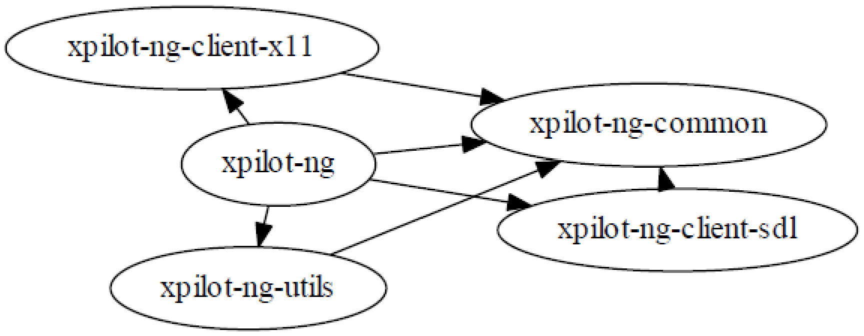

Figure 5.5: Commons Subpattern Example 1

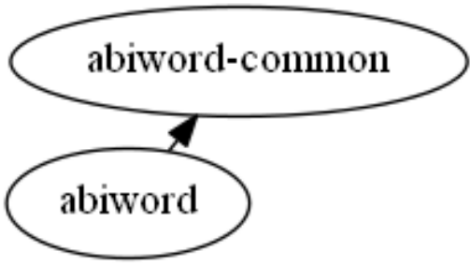

Figure 5.6: Commons Subpattern Example 2

storing more data. The idea behind the commons pattern is similar to creating a library package, but on a smaller scale. For instance, library packages are normally intended for a wider audience and require special packaging practices to make sure their effects on all dependent packages are error free [28].

5.5.2 Application Subpattern: Modular

Another subpattern identified is the modular subpattern. In this pattern, the application is split into multiple parts, where each part is largely independent from one another. When there are frequent updates to the application, Debian maintainers need to make sure that the new additions to the application can be applied individually. Figure 5.7 illustrates the dependency graph of a turn-based tactical strategy game called Wesnoth. Each of the packages that `wesnoth-all` depends on represents a single campaign (game level). When the game is updated with a new campaign, the package containing the campaign will be

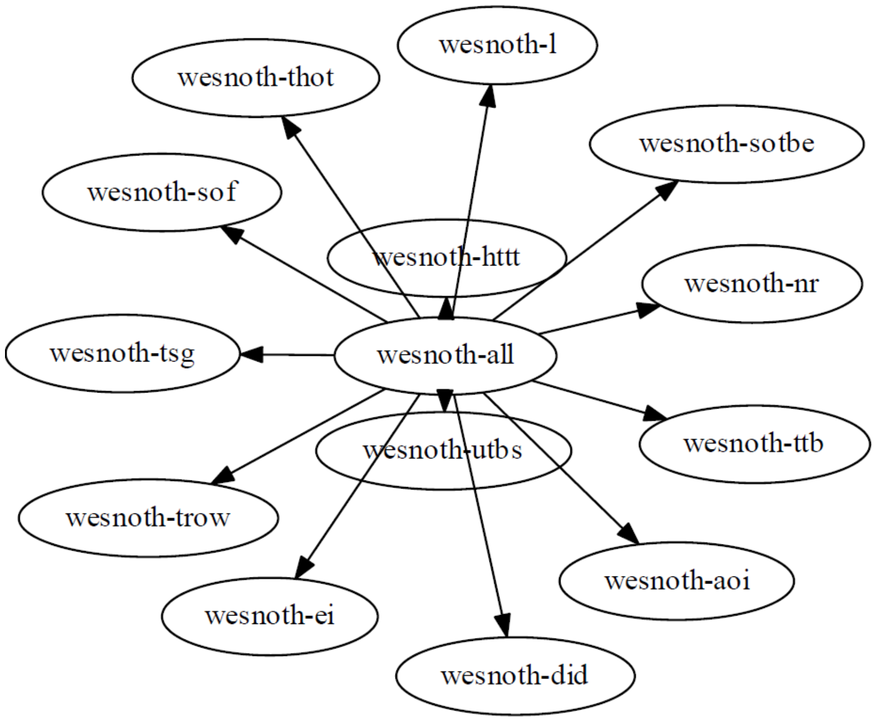

Figure 5.7: Modular Subpattern

installed without much modification to the existing Wesnoth packages. If Wesnoth was not modularized, each update would delete the existing version of Wesnoth. The updated version would install the exact same files again but with the extra campaign included. In general, it is important for maintainers to modularize the application they are packaging whenever possible.

5.5.3 Application Anti-Subpattern: Spaghetti

It is uncommon to see the dependency graph of an *application* layer package get as cluttered as the example in Figure 5.8. A graph such as this one with many inter dependencies is not common, but does occur occasionally. This design constitutes as an anti-subpattern because it is one that developers should avoid. When this spaghetti-like pattern occurs, the high number of interlacing dependencies make it is hard to see why particular packages

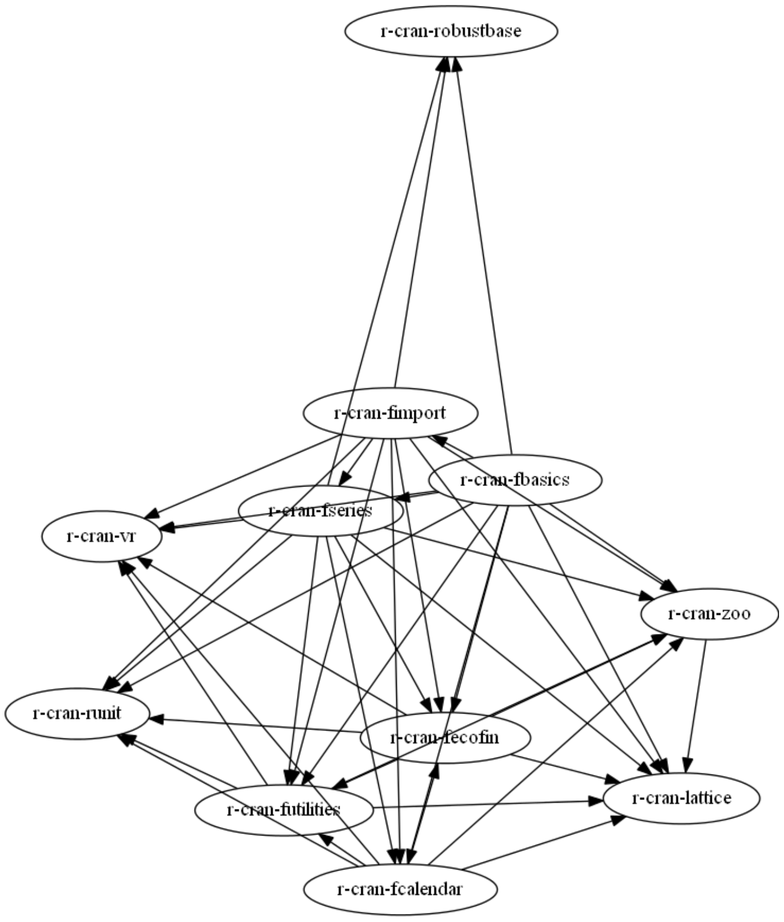

Figure 5.8: Spaghetti Anti-Subpattern

are referenced. This makes managing dependencies difficult because the maintainer needs to have a good understanding of the functionality of each package. Maintainers do not initially plan to have spaghetti dependencies. But over time the structure of the package may evolve to this cluttered state through a series of updates and revisions. At some point, the maintainers may need to restructure the packages.

5.5.4 Application Anti-Subpattern: Fully Connected

Dependency graphs in which all the packages of the application reference one another are usually not good design. The packages of `phpgroupware` in Figure 5.9 constitute an

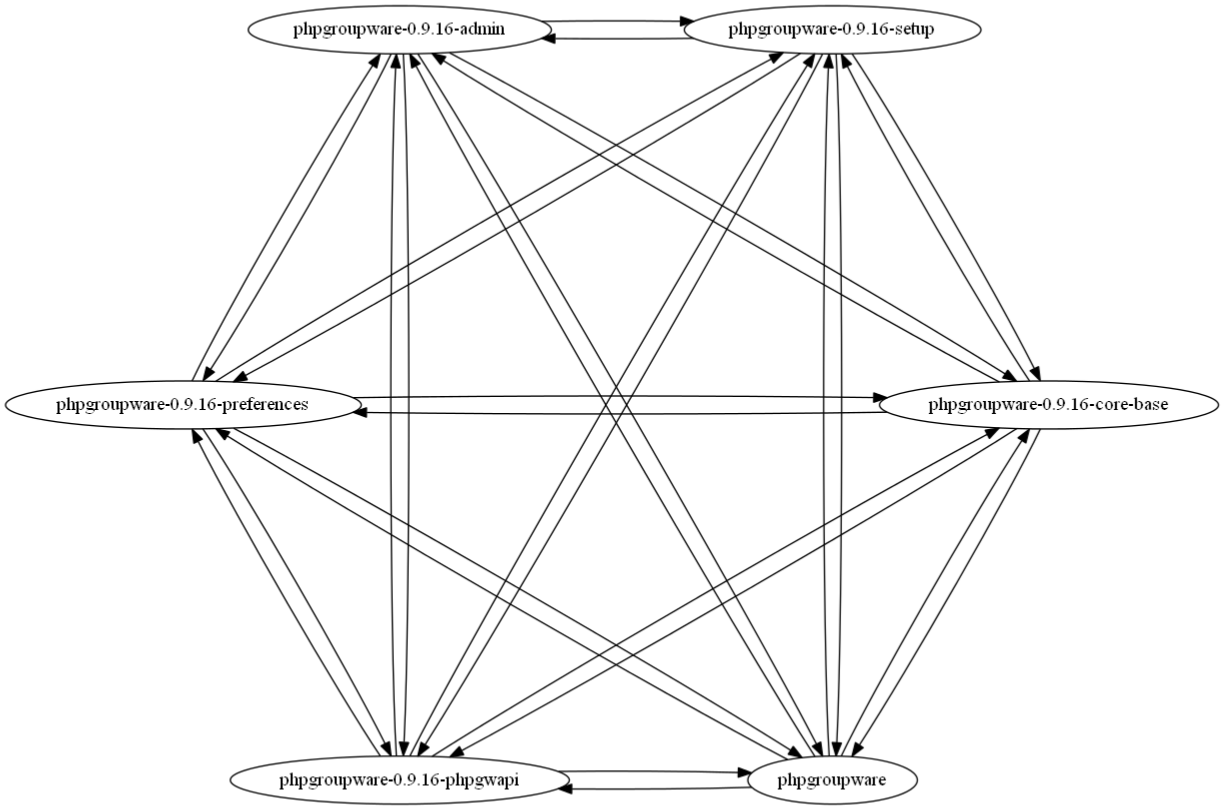
phpgroupware-0.9.16-admin
phpgroupware-0.9.16-setup
phpgroupware-0.9.16-preferences
phpgroupware-0.9.16-core-base
phpgroupware-0.9.16-phpgwapi
phpgroupware

Figure 5.9: Fully Connected Anti-Subpattern

example that exhibits full connectivity. If every package in the application depends on every other package, it should just remain as a single package. From the package description, each `phpgroupware` *application* package has a unique function and it appears that this application is modular; however, the dependencies render this modularization unsatisfying because the installation of one package requires the installation of the rest. When the dependency graph is completely connected as the one in Figure 5.9, it may serve as a warning that a redesign is appropriate. Splitting up an application into multiple parts in this case is unnecessary and may cause confusion to future maintainers.

5.5.5 Application Anti-Subpattern: Monolith

There is one more anti-subpattern that we address, but it cannot be visualized like the others, which makes it difficult to recognize. This is the Monolith subpattern and it applies to large application level packages. From the Commons and the Modular subpatterns; splitting up a larger package into smaller ones is a good idea and leads to an improved structure. Large stand-alone packages goes against this philosophy and is therefore why it is an anti-subpattern. However, not every stand-alone package is a Monolith, since there are many Debian packages that are lightweight and small. Applying the Commons or the Modular pattern to these small packages would not yield any significant returns and thus the package should remain on its own. For larger scale applications, splitting up the package should be the job of the maintainer. It is not good practice to keep a large application as one monolithic binary package. While monoliths are difficult for people not involved with the package to notice, the maintainers in charge of the package should know when Monolith anti-subpattern presents itself.

5.6 Correlation By Package Co-Change Analysis

System architecture is an extensive topic and the more we know about how the elements interact with one another, the better our understanding of it will be. So far in our analysis of Debian's architecture, we have focused our efforts on the written dependencies in the package control file. There is a possibility that software collections contain hidden dependencies between its packages that are not documented. Co-change analysis via release history is another method that has been shown in the past to unravel system architecture. Zimmermann, Diehl and Zeller describe how revision history can tell us which parts of the system are coupled by common changes [42]. The idea is that common changes of entities indicate evolutionary dependencies. Whenever Part A is altered, Part B is altered as well. The more frequently entities change at the same time, the stronger the coupling. The information about how and why a system evolves is rooted in the revision history. Using revision history as additional knowledge source allows for an improved assessment of software architecture. Along the same lines, Gall, Hajek and Jazayeri pointed out that there are hidden dependencies within software [12]. These dependencies are not exposed by code or documentation, but rather through a software developer's intuition and knowledge. The dependencies are expressed when the developer makes changes to the entities he or she is working with. More specifically, when a change of a certain type to one entity is committed, corresponding changes are always made to another set of entities. They call these dependencies logical dependencies, rather than the syntactic dependencies found in code. All syntactic dependencies are logical ones, but finding the logical dependencies which are not syntactic is what we are after.

The paper by Gall, Hajek and Jazayeri detected more coarse-grained coupling, while the paper by Zimmermann, Diehl and Zeller looked into more fine-grained coupling. Instead of just examining changes in the file, they also extract the functions, methods and attributes changed. Instead of just solely looking at the date of the file changed/uploaded, they also took into account the author, location, content and rationale. This finer-grained analysis allowed them to better understand the commonalities and anomalies between the software elements. Using these attributes as input into their ROSE prototype, they were able to test it on a number of different systems. For example, the GCC compiler contained 92,948 program entities and was found to have 3,424,012 dependencies.

We explore something similar for Debian. The dependencies found within the control files provide insight into the structure of Debian, but are there hidden dependencies that enable us to gain more knowledge about Debian's architecture? We answer this question by applying the data found in the package release history. We are able to access this data via the Ultimate Debian Database. Using the UDD, we are able to get release history as far back as August 1998. We extract the data for each package uploaded onto the Debian server from August 1998 to February 2012. In total, there were 354,668 entries and 25,674 unique source packages.

Similar to the studies described above, we use the concept of co-change in order to determine coupling between packages. Our algorithm for determining coupling between Debian packages lacks the data required to perform a more fine grain analysis like the study done by Zimmermann, Diehl and Zeller. But our analysis should give us an idea on the effectiveness of using release history to determine package coupling. Figure 5.10 illustrates how we implemented our co-change analysis. In this example, we have the release history for four different packages: A-1, A-2, B-1 and C-1. A-1 and A-2 are coupled together, but currently unbeknownst to us. We loop through all the packages and place a one hour time window on each. This means that packages that are uploaded 30 minutes ahead and before the current package will be grouped together. In Figure 5.10 the package A-2 groups together A-1 and B-1. Now all other occurrences of A-2 are searched for and their groupings based on the one hour time window are also noted. Once that is complete, the common packages between all the groupings are related by co-change to the current package we are examining. For package A-2, the only other co-changed package to appear in all the groupings is A-1. This procedure is repeated for all other packages to give us a list of co-changed groupings.

The example provided is very basic and when applied to the set of Debian packages yields far too many co-change dependencies. In order to get more useful results, certain thresholds need to be in place. The first threshold we can change is the timing window. A larger the window provides a greater number of coupled packages, but lower accuracy. On the other hand, if the window is too small, then we will miss a number of correlated

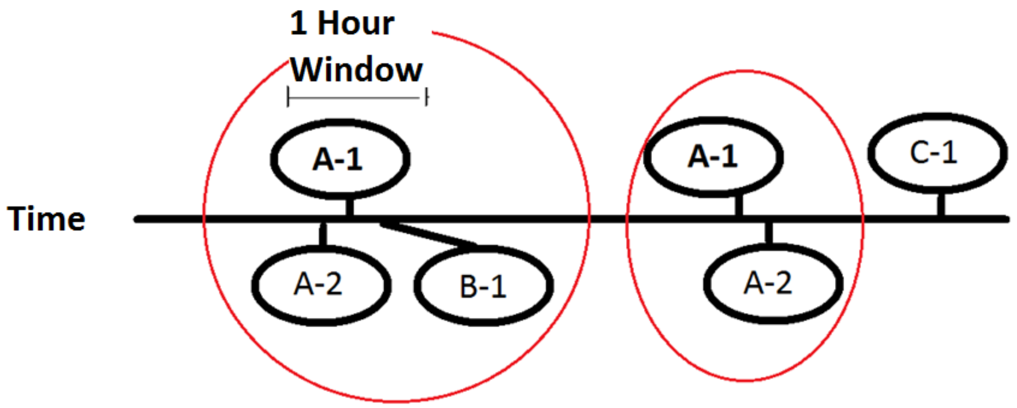

Figure 5.10: Package Co-Change Example

packages. In the end, we decided to go with a window of one hour to give 30 minutes for each package upload. Certain source packages are quite large and can take a long time to upload. If we made the window any smaller, then we would have missed these correlations. The second threshold we change is the minimum occurrence number. In Figure 5.10, only 2 version of A-2 were uploaded. Are two occurrences enough to say that the packages are correlated? In order to ensure that the grouping is not a coincidence, we were set the minimum occurrence number to 4. The last threshold deals with the matching ratio between the different groupings. Do all groupings need to contain the package in order for it to count as a correlation? Or can we set a threshold that will identify package correlation? For instance, if package D-1 has an occurrence of 15 and package D-2 was found in 13 of those occurrences, can we say that these two packages are correlated? Setting the matching ratio at 100% match greatly reduces the number of correlation because there are times when correlated packages are not always uploaded at the same time. We decided to go with a 75% match since that ensures some correlation, but also gives some leeway for the times where the packages are not uploaded together. The purpose of performing co-change analysis was to see if there were any hidden dependencies in Debian and how would it affect our view of Debian's architecture. Therefore, the method we used to determine these thresholds was through observation and testing. While a more mathematical and scientific approach could have been done, we do not believe it would have affected our conclusions on the effectiveness of co-change analysis for Debian.

5.6.1 Results From Co-Change Analysis

Overall, we discover that there were 202 source packages that had some form of correlation with other source packages by co-change analysis. This is a very small number considering we analyzed 25674 unique source packages (package families). This equates to just 0.787% of source packages having a co-change correlation. Since there are such a small number of correlated source packages, it is difficult to use this limited amount of information to further help further discover the architecture of Debian. The use of revision history helped to identify many dependencies in regular software projects. However, when it comes to software collections, it is not as effective. This is an interesting fact since it suggests that source packages are rarely worked on simultaneously. Rather, Debian maintainers focus their efforts only on one source at a time. For the source packages where a co-change correlation was found, what kind of packages are they? We will provide a couple examples of source packages that were discovered from the completed analysis. For instance, `Pdlzip` and `clzip` were found to be correlated through co-change analysis. Both are lossless data compressors based on the LZMA algorithm. The binaries they produce have no dependencies on each other or through any other package. `Pdlzip` is the public domain version and contains a simpler implementation, while `clzip` is the C language version of `lzip` intended for embedded devices. These two source packages are evidently related, and we would not have known without doing the co-change analysis. Another example is with `ruby-actionpack-2.3` which helps with web request routing and `ruby-actionmailer-2.3` which is a framework for designing email service layers. In this case, the binaries from `ruby-actionmailer-2.3` are dependent on `ruby-actionpack-2.3`. Therefore the co-change analysis did not expose a new correlation. Among the 202 source packages we found to have a co-change correlation, not all of them revealed new hidden dependencies. Overall, the number of co-changed correlated packages is minimal and not high enough to impact our current understanding of Debian's architecture.

5.7 Summary

The goal of this chapter is to determine the architecture of open-source software collections. There is a huge user base of open-source software and the number of new packages available is growing at a staggering rate. Yet amidst this popularity, there is an insufficient understanding on the types packages uploaded and how they interact with each other. In this chapter, we present the notion that packages either exist in isolation as hermits or belong to one of five layers in a *layered* pattern. These layers are: metapackage, application, application library, specific purpose library and general purpose library. Within Debian, over half of the packages are hermits, and therefore have no major dependencies on other packages. The other half partakes in a top heavy *layered* pattern. This means that the majority of packages are grouped into the application layer, with fewer at the library level.

The many interactions between each of the layers are what cause such a large dependency graph as seen in Figure 5.2. Classifying packages into their layers and only visualizing the dependencies within each layer allows for more comprehensible graphs. The visualization of packages in the application layer can be important because we get concise graphs that display how the application is organized. In this visualization, the automatic dependencies to the low-level libraries are removed and what remains is the packaging design from the developer who created the packages. We present the package design principles in the form of two subpatterns and three anti-subpatterns: commons, modular, spaghetti, fully connected and monolith. Experienced package maintainers are aware of the best practices when packaging software. But by documenting these patterns, we hope to help new developers produce higher quality packages that are less prone to bugs and more readily upgradable.

We also conclude that the ways developers work with and upload source packages are fundamentally different than with pure source code. When source code is committed to a repository, there is usually logical coupling occurring between files which can provide insight into the system architecture. However with Debian, we noticed a very sparse amount of coupling among packages. Using the techniques devised in this thesis, the conclude that the recorded dependency information in the control file is currently the only reliable method to determine package correlation.

Chapter 6

Threats to Validity and Future Work

In Chapter 4, our work relied on the UDD as a source of information; mistakes within the UDD could affect the internal validity of our study. In the study of Debian package bugs by Davis, Zhang, Nussbaum and German, they noticed certain inconsistencies such as missing package data and missing bug reports [9]. They calculated the percent missing to be 0.7%. While this is a small value and should not affect our results, it is something to be aware of. Furthermore, we noticed columns in the *archived_bugs* table provided the wrong information; the *affects_stable*, *affects_testing* and *affects_unstable* columns were all recorded as 'FALSE' which is not true. There were also some discrepancies with the package upload dates. A small number of them had a date in the 1970s when Debian did not exist and others had a date set in the far future. These far-fetched dates comprised around 0.03% of our overall package data and were filtered out from our results. Though these issues may be cause for concern, the size of the UDD is enormous and other small inaccuracies should have a minimal effect on our results. As for external validity, we cannot generalize the trends found in Debian for every type of software collection. However, based on a quick survey of the amount of software available in other large software systems, we know that they also grow at a very fast pace.

One of our main concerns from the work done in Chapter 5 is that there may be more architectural package patterns than described. As mentioned, the *layered* pattern was discovered by closely examining 20 different applications. There is a possibility that there are other ways in which packages are organized. There may also be some bias which can cause the dismissal of other package patterns. One way to solve this issue is to gather another group of people to survey the applications within the software collection. An alternate method we would like to implement in the future is to do interviews with experienced Debian developers to get their insight into common patterns they use when creating packages. Another threat to validity is our method of classifying the Debian packages according to the *layered* pattern. We presented an ad-hoc and a k-means solution which we believe did a reasonable job at giving an estimate on the type of packages that are found in Debian. However, misclassifications were found, so it was not entirely accurate. For future work, we would like to define training and testing sets in order to verify the accuracy of our classification. With this in place, we can further refine our algorithm to produce better results.

Lastly, we would like to address the external validity of our study. The large size and high popularity of Debian makes it an important open source software collection. However, this does not mean the evolution and package patterns found in Debian translate to other software collections. From our knowledge of the other Linux distributions, we expect there is a high chance that the others are structured similarly. Based on a quick survey of the amount of software available in other large software systems, we know that they also grow at a very fast pace and therefore might have the same evolutionary pattern as Debian. But nothing is confirmed until we mine the software repositories of other software collections. For future work, we want to address this issue of our study and explore the evolution and architecture of other software collections and compare them to Debian. We would apply the same techniques presented here on data from other software collections. By noting similarities and differences, we hope to gain a better understanding of how large scale software collections evolve and its architecture.

In a different direction, we would also like to explore Debian at a deeper level. Instead of focusing on evolution and architecture distinctly, we would explore the evolution of architecture. We would try to discover how Debian's package composition changes as the system evolves. Do packages gain more dependencies? How often do packages get restructured and at what rate do the anti-subpatterns get resolved? Studying the evolution of dependencies and packaging patterns can tell us more about Debian and how software collections are structured.

Chapter 7

Conclusions

Debian is one of the largest coordinated software collections, because of this, there is value in improving the techniques and processes used to maintain it. Insight into the growth and organization of this particular software collection can be beneficial to all those involved with these types of systems. This thesis offers a first step for improvement of software collections by observing the current state of Debian and how it has reached this point. Our study targets packages from life to death and their attributes to see how the software collection is affected. The facts that we learned with this longitudinal study are as follows:

- The number of package uploads has been relatively stable at around 30,000 per year, but when the *testing* distribution is frozen, the rate of uploads drops significantly.
- Most packages are superceded by newer versions rather than being removed outright.
- The average lifespan of a package is short with 88% of all packages not lasting longer than one year.
- There are an average of 360 new Debian Developers that help contribute to the system each year.
- A majority of package uploads are performed by a small subset of Debian Developers.
- Packaging bugs are on the decline since 2004.
- Package popularity mildly correlates with its update rate.

In addition to exploring the evolution of a software collection, this thesis also describes its architecture. By using binary package dependencies, we derive patterns from the interaction of these elements. We discover the following traits about Debian:

- Packages either belong to a *layered* pattern or exist in isolation as hermits.
- The *layered* pattern contains five layers based on where they fit along a dependency tree graph.
- The *In, Out, Transitive In* and *Transitive Out* degree can be used to help identify which layer a package belongs to.
- Using an ad-hoc or a clustering auto-classification technique, we learn that around half of the packages in Debian are hermits. For those packages that belong to the *layered* pattern, the majority are application packages.
- Dependencies among application packages can be visualized to see how a maintainer designed the packaging of a particular application.
- Five application subpatterns are defined based off of the application level visualizations.
- There is no significant correlation between upload history and logical coupling between packages.

Overall, the data gathered in this thesis can be useful to two groups of people: (1) software collection contributors and (2) researchers:

1. Software collection contributors:

   A subset of Debian developers are in an administrative role and make decisions regarding policies and procedures. They could review our observations to see if they are in line with their vision for Debian. For instance, they may not like seeing a slowdown in package uploads when "testing is frozen". They may choose to implement a new process to encourage more uploads during this period of time. There could also be a case where the policy makers feel that the extremely short lifespan of packages is harmful for users and request that maintainers more thoroughly test their packages before officially uploading them.

2. Researchers:

   The data provided in this study is substantial and can it provide a springboard for many other research topics. There are a number of questions researchers can try to answer after learning about the information presented in this study. Some examples could include:

   - Which packages have lived for over 15 years and why? What role do they play within the software collection?
   - What type of package families are updated rapidly? Can we predict them and/or slow them down?

   As a software collection grows even larger, it becomes increasingly important to see how it has reached its current state and how the elements in the system are interacting with each other. Doing so allows for necessary adjustments to better accommodate the growth. Once this work is applied to other various software collections, strengths and weaknesses of each collection can be determined and backed up with empirical data. Policy makers can use this data to further improve how their software collection functions.

References

[1] P. Abate and R. Di Cosmo. Predicting upgrade failures using dependency analysis. In *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on*, pages 145–150. IEEE, 2011.

[2] 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*, pages 89–99. IEEE Computer Society, 2009.

[3] J.J. Amor, G. Robles, J.M. González-Barahona, and I. Herraiz. From pigs to stripes: A travel through debian. In *Proceedings of the DebConf5 (Debian Annual Developers Meeting)*. Citeseer, 2005.

[4] J.J. Amor, G. Robles, J.M. González-Barahona, and F. Rivas. Measuring lenny: the size of debian 5.0. Technical report, Downloaded 2010-12-10, available at http://gsyc.es/˜ frivas/paper. pdf, 2009.

[5] J.J. Amor-Iglesias, J.M. González-Barahona, G. Robles-Martínez, and I. Herráiz-Tabernero. Measuring libre software using debian 3.1 (sarge) as a case study: preliminary results. *CEPIS promotes*, page 13, 2005.

[6] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice*. SEI series in software engineering. Addison-Wesley, 1998.

[7] J. Boender, R. Di Cosmo, J. Vouillon, B. Durak, and F. Mancinelli. Improving the quality of gnu/linux distributions. In *Computer Software and Applications, 2008. COMPSAC'08. 32nd Annual IEEE International*, pages 1240–1246. IEEE, 2008.

[8] F. Buschmann. *Pattern-Oriented Software Architecture: A System of Patterns*. Number v. 1 in Wiley Series in Software Design Patterns. Wiley, 1996.

[9] J. Davies, H. Zhang, L. Nussbaum, and D.M. German. Perspectives on bugs in the debian bug tracking system. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 86–89. IEEE, 2010.

[10] O.F. de Sousa, MA de Menezes, and T.J.P. Penna. Analysis of the package dependency on debian gnu/linux. *Journal of Computational Interdisciplinary Sciences*, 1(2):127–133, 2009.

[11] A. Deshpande and D. Riehle. The total growth of open source. *Open Source Development, Communities and Quality*, pages 197–209, 2008.

[12] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Software Maintenance, 1998. Proceedings. International Conference on*, pages 190–198. IEEE, 1998.

[13] D. Garlan and D.E. Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4):269–274, 1995.

[14] D.M. German. Using software distributions to understand the relationship among free and open source software projects. In *Mining Software Repositories, 2007. ICSE Workshops MSR'07. Fourth International Workshop on*, pages 24–24. IEEE, 2007.

[15] D.M. German, J.M. Gonzalez-Barahona, and G. Robles. A model to understand the building and running inter-dependencies of software. In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pages 140–149. IEEE, 2007.

[16] M.W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Software Maintenance, 2000. Proceedings. International Conference on*, pages 131–142. IEEE, 2000.

[17] J.M. González-Barahona, M.A.O. Perez, P. de las Heras Quirós, J.C. González, and V.M. Olivera. Counting potatoes: the size of debian 2.2. *Upgrade Magazine*, 2(6):60–66, 2001.

[18] J.M. Gonzalez-Barahona, G. Robles, M. Michlmayr, J.J. Amor, and D.M. German. Macro-level software evolution: a case study of a large software compilation. *Empirical Software Engineering*, 14(3):262–285, 2009.

[19] G. Guojun, M. Chaoqu, and W. Jianhong. Data clustering theory algorithm and application, 2007.

[20] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I.H. Witten. The weka data mining software: an update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[21] I. Herraiz, E. Shihab, T.H.D. Nguyen, and A.E. Hassan. Impact of installation counts on perceived quality: A case study on debian. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*, pages 219–228. IEEE, 2011.

[22] S. Koch. Evolution of open source software systems–a large-scale investigation. In *Proceedings of the 1st International Conference on Open Source Systems*, 2005.

[23] N. LaBelle and E. Wallingford. Inter-package dependency networks in open-source software. *Arxiv preprint cs/0411096*, 2004.

[24] M.M. Lehman, D.E. Perry, and J.F. Ramil. Implications of evolution metrics on software maintenance. In *Software Maintenance, 1998. Proceedings. International Conference on*, pages 208–217. IEEE, 1998.

[25] M.M. Lehman, J.F. Ramil, P.D. Wernick, D.E. Perry, and W.M. Turski. Metrics and laws of software evolution-the nineties view. In *Software Metrics Symposium, 1997. Proceedings., Fourth International*, pages 20–32. IEEE, 1997.

[26] Xiao Liu, Dong Yuan, Gaofeng Zhang, Wenhao Li, Dahai Cao, Qiang He, Jinjun Chen, Yun Yang, Xiao Liu, Dong Yuan, Gaofeng Zhang, Wenhao Li, Dahai Cao, Qiang He, Jinjun Chen, and Yun Yang. Cloud workflow system architecture. In *The Design of Cloud Workflow Systems*, SpringerBriefs in Computer Science, pages 13–18. Springer New York, 2012. 10.1007/978-1-4614-1933-42.

[27] M. Michlmayr and B.M. Hill. Quality and the reliance on individuals in free software projects. In *Proceedings of the 3rd Workshop on Open Source Software Engineering*, pages 105–109. Citeseer, 2003.

[28] I. Murdock. Overview of the debian gnu/linux system. *Linux Journal*, 1994(6es):15, 1994.

[29] L. Nussbaum and S. Zacchiroli. The ultimate debian database: Consolidating bazaar metadata for quality assurance and data mining. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 52–61. IEEE, 2010.

[30] B. Perens et al. The open source definition. *Open sources: voices from the open source revolution*, pages 171–188, 1999.

[31] G. Robles, J.J. Amor, J.M. Gonzalez-Barahona, and I. Herraiz. Evolution and growth in large libre software projects. In *Principles of Software Evolution, Eighth International Workshop on*, pages 165–174. IEEE, 2005.

[32] G. Robles, J.M. Gonzalez-Barahona, and M. Michlmayr. Evolution of volunteer participation in libre software projects: evidence from debian. In *Proceedings of the 1st International Conference on Open Source Systems*, pages 100–107, 2005.

[33] G. Robles, J.M. Gonzalez-Barahona, M. Michlmayr, and J.J. Amor. Mining large software compilations over time: another perspective of software evolution. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 3–9. ACM, 2006.

[34] Tassia Serrao, Lucas M. Braz, Sergio Crespo C. S. Pinto, and Gisela Clunie. An architecture based on web services for mobile social software. In *Proceedings of the 6th Euro American Conference on Telematics and Information Systems*, EATIS '12, pages 413–416, New York, NY, USA, 2012. ACM.

[35] M. Shaw and D. Garlan. *Software architecture: perspectives on an emerging discipline.* An Alan R. Apt book. Prentice Hall, 1996.

[36] G. Succi, J. Paulson, and A. Eberlein. Preliminary results from an empirical study on the growth of open source and commercial software products. In *EDSER-3 Workshop*, pages 14–15. Citeseer, 2001.

[37] S. Sun, C. Xia, Z. Chen, J. Sun, and L. Wang. On structural properties of large-scale software systems: from the perspective of complex networks. In *Fuzzy Systems and Knowledge Discovery, 2009. FSKD'09. Sixth International Conference on*, volume 7, pages 309–313. IEEE, 2009.

[38] N. Synytskyy, R.C. Holt, and I. Davis. Browsing software architectures with lsedit. In *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*, pages 176–178. IEEE, 2005.

[39] Ji-Uoo Tak, Roger Lee, and Haeng-Kon Kim. Design of mobile software architecture. In Roger Lee, editor, *Software and Network Engineering*, volume 413 of *Studies in Computational Intelligence*, pages 133–146. Springer Berlin / Heidelberg, 2012. 10.1007/978-3-642-28670-412.

[40] C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner. Opium: Optimal package install/uninstall manager. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 178–188. IEEE, 2007.

[41] X.F. Wang and G. Chen. Complex networks: small-world, scale-free and beyond. *Circuits and Systems Magazine, IEEE*, 3(1):6–20, 2003.

[42] T. Zimmermann, S. Diehl, and A. Zeller. How history justifies system architecture (or not). In *Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of*, pages 73–83. IEEE, 2003.