A DEFECT-CENTRIC OPEN-SOURCE LIFE-CYCLE MODEL

By

Brandon Nuttall

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Computer Science

May, 2006

Nashville, Tennessee

Approved

Professor Stephen R. Schach

Professor Douglas H. Fisher

For Dad

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

To many, open-source software (henceforth OSS) is an enigma. Indeed, the common description of the OSS development process reads like a bit of dot-com mythology: roving bands of developers spread over four continents get their heads together, get organized, and cause millions of lines of enterprise-class code to fall from the sky. Given this description, any properly-cynical CIO should have reservations about trusting their enterprise to software developed in a manner that seems so egalitarian.

Furthermore, any project manager who has had the pleasure of trying to keep a software project on time and under budget knows how difficult it is to herd teams of developers toward a common goal. Hugely complicated books such as *The Unified Process* codify a development process that reflects industry best-practice; however, cursory glances at OSS development teams reveal communities that seem to have little respect for these rules. How then can OSS development projects thrive in a software ecosystem where many well-funded and well-staffed commercial software projects fail?

The key observation that resolves this dramatic tension is the observation that the OSS development process is not as chaotic as it seems at first glance. This work claims that an important component of the process is **defect-centric** – defined by the introduction and resolution of defects in the software. This paper then isolates and describes this process and examines the development process of a popular OSS product through defect-centric analysis.

This examination is split into several different sections. Chapter II examines canonical software development methodologies and addresses how they might (or might not) apply to open-source software development. Chapter III introduces and defines the defect-centric process of OSS development and sets forth hypotheses presented by the defect-centric model. Chapter IV introduces the Mozilla project and uses it as a case study to examine the defect-centric model. Finally, Chapters V and VI address future areas of study and conclude the work.

CHAPTER II

RELATED WORK

Software engineers have been creating models in order to better understand software development for decades. Their methods and results are applicable to the development models that they studied. However, the OSS development process differs from previously-studied development processes enough to make these previous attempts inapplicable. This section examines how that could be the case. After establishing this, this section then introduces the theoretical underpinnings of a new model, probed (but not rigorously described) by previous research.

Classical Models

In the beginning, software projects were simple and small, allowing a developer to have an accurate mental model of the entire system. As hardware platforms have become more powerful and programming languages more complex and expressive, software systems have grown to the point that it is impossible for a single software developer to understand the entire system. This problem of imperfect information has given rise to software processes like the Unified Process [Jacobson et al. 1998] and Extreme Programming [Beck 2000] that attempt to guide development in a way that reduces this problem's impact. Because this thesis models the OSS development process, it would make sense to first try to apply existing models and processes to this new problem

domain. However, as powerful and battle-tested as these existing processes are, there are at least two reasons why looking to these models is a mistake.

First, the Unified Process and Extreme Programming are **iterative** (work is done in a series of discrete steps) and **incremental** (future work is based on past work). Though these are also accurate descriptors of the OSS development process, iteration and incrementation for the OSS serve a fundamentally different purpose than they do for the Unified Process or Extreme Programming.

In the Unified Process and Extreme Programming, the driving purpose for an iterative and incremental development style is to mitigate the risk that stems from imperfect knowledge of the requirements of the software product [Jacobson et al. 1998]. Developers are usually not users of the product that they create; therefore, they should be supplied with documentation of how the product should function in the form of requirements or use cases. Often these documents are incomplete or faulty, a problem that is resolved by iterative and incremental intermediate products (or builds) that developers present to the client to solicit feedback. The developers then use this feedback to fine-tune their requirements documentation and ultimately their product.

Open-source software is developed differently. Many development efforts begin not for money or fame but to scratch an itch -- to solve a problem faced by the originator. The originator, and those who have the same itch to scratch, are almost always also users of the product and have intimate knowledge of the product's requirements. Here, risk management is no longer an all-important concern; rather, the iterative and incremental development process serves in part both to keep the community that grows around a software project together [Raymond 2001] and to optimize the defect resolution process

(which will be addressed in Chapter III). Neither of these purposes have an analog in the Unified Process or in Extreme Programming.

Furthermore, the Unified Process and Extreme Programming primarily concern themselves with pre-release development. In contrast, researchers have found that it is extremely hard, if not impossible, to get the OSS development engine started without an initial release of code, because without code it is impossible to attract co-developers to the project [Raymond 2001]. Additionally, rather than simply being a prototype or a proof-of-concept, this first code release must be a working product to attract users[1]. Using the classical definition of maintenance, this code distribution fits the definition of a release, development activities after that release being considered maintenance.

Origins of the Open-Source Software Development Process

To understand why the open-source software development process does not conform to existing models requires knowing of its origins. Richard Stallman, the Free Software[2] pioneer who authored the GPL (the GNU Public License, a commonly-encountered OSS license) and founded the FSF (Free Software Foundation), originated the formal concept of open-source software when he launched the GNU project in 1983 [Free Software Foundation 2005]. However, the development process employed by the FSF was similar to that found in contemporaneous proprietary software companies: being

1 Attracting more users invites more feedback, which can speed defect resolution. This is discussed in more detail in Chapter III.

2 To the chagrin of some, this thesis will use the term "open-source software" to refer to both open-source and Free software. For those readers who desire insight as to the difference between these two terms, the author invites you to read the official open-source definition at [Open Source Initiative 2005] and the official Free software definition at [Free Software Foundation 2004].

characterized by strong central control, high barrier to entry, and slow release cycles [Raymond 2001].

When Linus Torvalds and his Linux came on the scene in 1991, the FSF had completed every significant part of its GNU system but the kernel. Though Torvalds never thought that Linux would become the kernel of the GNU system, development ramped up rapidly and Linux soon eclipsed the HURD (the GNU project's kernel) in terms of features and popularity [Stallman 2001]. The key to Linux's initial success was not its technical merit (which, though substantial, was not extraordinary) but instead Torvalds's management style: being characterized by flexible central control, low barriers to entry, and rapid release cycles [Raymond 2001].

Describing Open-Source Software Development

It was this development style born from Torvald's Linux project that Raymond describes as being the seminal example of the OSS development process. Through the exploration of his personal experience with Linux and his own *fetchmail*, Raymond presents a set of proverbs that distill OSS development's conventional wisdom into their essence. The most important of these proverbs, known as “Linus's Law”, forms the conceptual base for the defect-centric model and is explored in more detail below. In contrast, most of the rest of the proverbs are little more than rules of thumb and are not useful when attempting to describe a formal model.

Many other authors have attempted to formalize OSS development practices. The skeleton of the Mozilla project, used in Chapter IV to illustrate a defect-centric model, is formally described in [Reis et al. 2002]. In that work, Reis explores the workers, CASE

tools, and low-level activities from which the Mozilla product and process are constructed; however, Reis does not address the higher-level workflows that drive the actual software development, a deficiency that this work addresses.

Another work that uses Mozilla as a case study is [Mockus et al. 2002], which explores the relative levels of participation for contributors to the Mozilla source code repository. In contrast, this work is concerned less with contributions to the Mozilla code base and more with contributions to Mozilla's Bugzilla system, a defect tracking database. Furthermore, the case study of Mozilla found in Mockus was published in mid-2002, a time when the browser was in a pre-1.0 state, making the observations found therein somewhat obsolete. An analysis similar to [Mockus et al. 2002] is found in [German Mockus 2003], which addresses the Evolution product; however, the Evolution product is much smaller in scope than Mozilla[3].

Linus's Law

One of the key questions to answer when trying to describe a development process is to describe how the development process sustains itself. Fundamentally, a development process converts fuel (the time investment of its contributors) into work (successful pieces of software). In classical development processes, the fuel could be such things as the pressure to deliver to the customer, or even money -- given enough fuel, the project continues; remove it, and the project must fail.

This thesis argues in Chapter III that defects are an important driver in the OSS development process. The most important insight into the effect of defects on the open-

3 Evolution is a Microsoft Outlook-style mail and calendaring application, whereas the Mozilla product includes a mail application, a calendar application, and a host of other applications and tools.

source development process is given in [Raymond 2001] in a proverb termed “Linus's Law” and its two corollaries:

Proverb 1: Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix will be obvious to someone.

Corollary 1: Given enough eyeballs, all bugs are shallow.

Corollary 2: Debugging is parallelizable.

These proverbs have three important consequences. First, they assert that there exist clearly-defined actors and roles in the OSS development process. Identifying the roles that particular contributors play is key to describing how actors work together to create software; these actors and roles are described in Chapter III.

Second, these proverbs assert that eyeballs, or project contributors, are effective in resolving bugs, or issues in the software. Previous work has cast doubt as to the applicability of this proverb to the OSS development process, as many claim that faults are not fixed by the many eyeballs the proverb would suggest, but instead by a small, select group of core contributors. This argument is flawed: according to Torvalds, the primary activity of the OSS development process is not fixing faults but is instead **fault characterization**, the act of finding faults through examination of a pattern of failures [Raymond 2001]. This distinction will be explored completely in Chapter III.

Third, these proverbs assert that OSS projects can effectively scale to many contributors. This scalability, asserted by the second corollary of Linus's Law, flies in the face of conventional wisdom as codified by Brooks's Law (in [Brooks 1995]) . Chapter III of this work introduces the concept of Raymond's **trace paths** and how they allow

open-source development projects to parallelize development and optimize the use of developer resources.

Open-Source Development Group Organization

Most researchers view the contributors to an OSS development project as being stratified rather than homogeneous. Raymond, when speaking of the structure of his *fetchmail* product and of other open-source products with which he is familiar, refers to the structure of an OSS project as having a project **core** consisting of a few major contributors, surrounded by a **halo** of comparatively many minor contributors [Raymond 2001]. Other works such as [Nakakoji et al. 2002] and [Ye Kishida 2003] agree with Raymond on the stratified nature of the contributors but disagree on the particular number of groups. Regardless of the number or nature of the groups, these works establish that analysis of a particular development activity should reveal that contributors are stratified. This is the major motivation behind the discussion in Chapter III.

CHAPTER III

MODELING OPEN-SOURCE SOFTWARE DEVELOPMENT

The open-source software ecosystem is widely (and wildly) diverse, encompassing a myriad of different projects and management styles. This thesis brings these disparate projects under a common banner by showing how basic assumptions about the structure of an open-source software development project can be fleshed out into a model which should be generally applicable.

Building Blocks

The first step to describe the open-source software development model (henceforth OSSDM) is to state the assumptions made. This thesis asserts that the following things are minimally true before the OSSDM can be applied to a software project:

- Work that is performed on the software project can be measured and quantified. A unit of work is a **contribution** to the project.
- Individuals contribute to the project. These participants are called **actors**. Work performed by actors can be grouped into distinct classes called **activities**.

Not all open-source software projects will fit these requirements; for example, if a project does not rigorously document its development activities, the OSSDM may not be applicable because work might not be measurable.

Development As Maintenance

The second step in describing the OSSDM is to present a framework for thinking about development activities within a project. One lens through which OSS development can be observed is Linus's Law, which describes work on an OSS project in terms of contributors, defects, and fixes, and as argued in Chapter II this image can be described in terms of classical maintenance activities. Consequently, OSS development activities should reflect to a large extent the three classical maintenance activities: corrective maintenance, perfective maintenance, and adaptive maintenance.

- The way in which the OSSDM handles **defects**, errors in the source code, is analogous to corrective maintenance.
- The way in which the OSSDM handles **enhancements**, or improvements to the product, is analogous to perfective maintenance.
- The way in which the OSSDM handles **adaptations**, or changes made to the product for the sake of making it work on a platform different from the one for which it was originally built, is analogous to adaptive maintenance.

Each of these three fundamental development activities will be examined in turn to give a complete accounting of the activities performed in the OSSDM.

Contributions, Actors, and Activities

The above activities, if performed effectively, should tangibly change the OSS product. The OSSDM should describe how these changes are measured. This measurement could be something as straightforward as LOC (lines of code) or as esoteric as an obscure, project-specific code-quality metric; regardless of the particular metric

chosen, this fundamental unit of work is a **contribution**. The people that perform this work are actors, and the work that is performed by an actor is that actor's activity.

Choosing appropriate metrics of contribution is difficult because all software projects are managed differently, making it important to choose metrics on a per-project basis. Regardless, a good metric should have two basic properties:

- The metric should be **applicable** -- an increase or decrease in the metric should reflect the increase or decrease of something tangible.
- The metric should be **discriminatory** -- applying the metric to a pool of contributors should divide the pool into groups of similar actors.[4]

Good choices for metrics include:

- Defect reports created per contributor per version, counted by tallying the number of defect reports created by a particular individual for a particular release of the software.
- Unique communications generated per contributor per version, counted by tallying the number of email, message board, or other communications participated in by a particular individual for a particular release of the software.
- Source commits per contributor per version, counted by tallying the lines of code a particular individual committed to a source repository for a particular release of the software.

Bad choices for metrics include:

- Karma points, counted by the number of pats on the head a particular individual received from the product's project manager for a particular release of the

4 For OSSDM projects with very few contributors, this guideline may be dropped; however, projects of that size would probably not be interesting enough to model.

software. (This metric is not applicable because a larger karma score does not necessarily result in something tangible for the software product.)

- Number of email addresses, counted by tallying the number of individuals registered for the project's defect tracking database. (This metric is not discriminatory because it does not divide individuals who contribute to the project from one another.)
- Raw lines of code, counted by counting the total number of lines of code in the software product for each version. (This metric is not discriminatory because it applies to the software product as a whole and not to the actors who contribute to it.)

The discriminatory nature contribution metrics makes the differences between actors readily apparent. For example, assume that for a particular open-source software project that the "number of defect reports submitted per contributor" metric is valid. If this is the case, it can rightly be said that the contributors that have contributed zero defect reports are not defect reporters, and the contributors that have contributed one or more are. The property of being a defect reporter refers to work that the actor contributes to the open-source software development project, which in our nomenclature is an activity.

Development Processes

Given the concept of actors and activities, and given a way to measure their effect on the software product, it becomes possible to describe and measure the processes by which the open-source software development process handles defects, enhancements, and adaptations, the three basic development activities of the OSSDM.

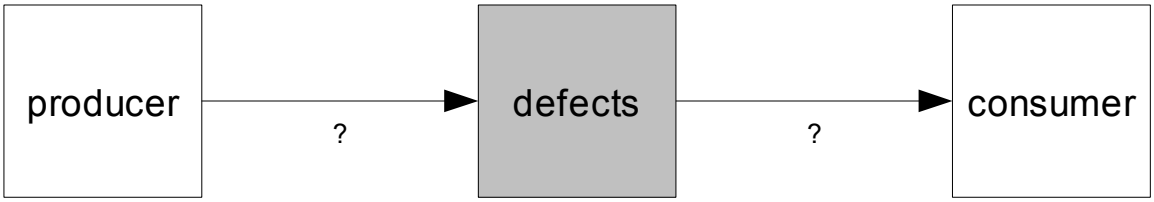

Figure 1: A Skeleton of the Defect Workflow

Defects

Errors in a software product's source code are the unsavory side effect of software's necessarily-imperfect human origin. Fundamentally, all defects in the source code must have a **producer**, or the vector by which the defect was exposed. In good software products, these defects are eventually removed from the product by the **consumer** (Figure 1).

Deconstructing the producer and the consumer allows the analysis of how defects are resolved by the various actors and processes in the OSSDM.

Deconstructing the Producer

From the point of view of the software project, a particular defect itself is not observable. What instead exists is a **defect report**, or an artifact that documents a particular perceived defect in the system. This defect report does not materialize from the ether; rather, it must be generated by a **tester** who observes it and documents it, bringing it to the software project's attention.

Furthermore, this perceived defect is not observed in isolation; rather, it is observed in one or more particular **releases** of the software product that the tester acquires and tests. In addition, the product release also did not come from nowhere – at some point, some actor must have decided to "bless" that release as a formal product

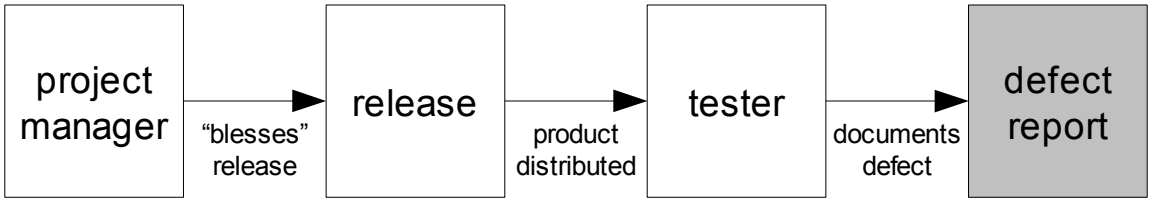

Figure 2: The Producer Component of the Defect Workflow

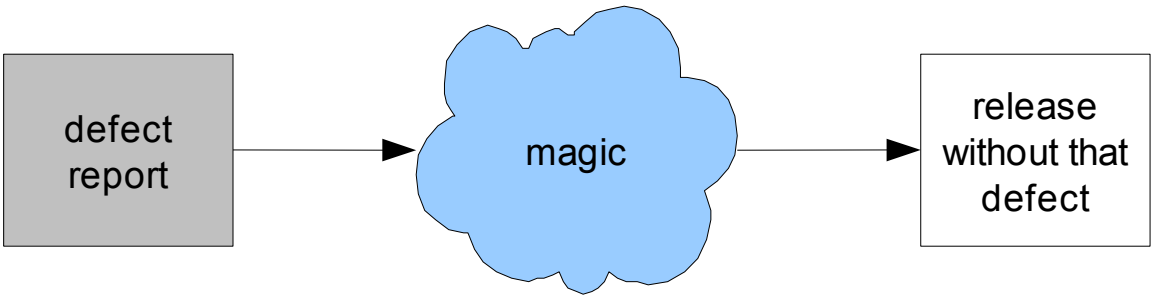

Figure 3: A Skeleton of the Consumer Workflow

release by packaging it and promoting it. This actor is a **project manager**. (The producer workflow is described in Figure 2.)

Deconstructing the Consumer

Just as a defect is produced, it should also be consumed. For a defect to be truly considered consumed, it must be removed from the product: a release of the product must be made with that defect missing (Figure 3).

The "magic" bubble represents the process of converting the knowledge documented in the defect report into a release of the software product without that defect. The first step is to determine whether the defect report documents a **fault**, or an observed error in the source code, or a **failure**, or a situation where the product's behavior deviates from the observer's expectations (Figure 4).

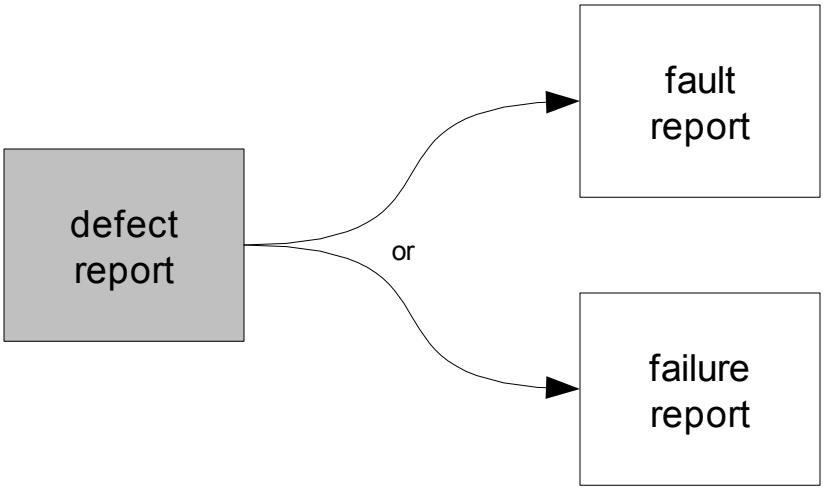

Figure 4: Sorting Defect Reports into Faults and Failures

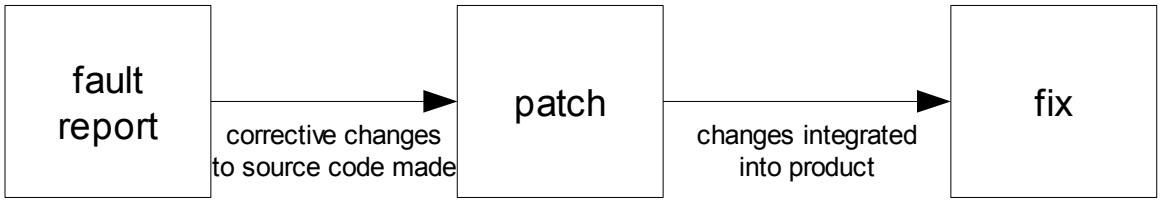

Figure 5: Resolving a Fault Report

Because a fault report describes an error in the product's source code, these errors can be resolved directly by determining exactly which corrections must be made to the project's source code. These changes are documented in an artifact called a **patch**. This patch, once accepted by the project manager and integrated into the source code of the product, becomes a **fix** (Figure 5).

A failure report is handled differently. Because a failure report does not describe an error in the product's source code, this failure must be traced to the source code to reveal its underlying fault. To borrow Linus Torvalds' term, the process by which a failure is mapped to a fault is called *characterization*[5] (Figure 6).

5 The text of Linus' law makes it clear that Torvalds considers the act of characterization and the fix as two separate activities – they may even be performed by different people!

Figure 6: Characterizing Failure Reports

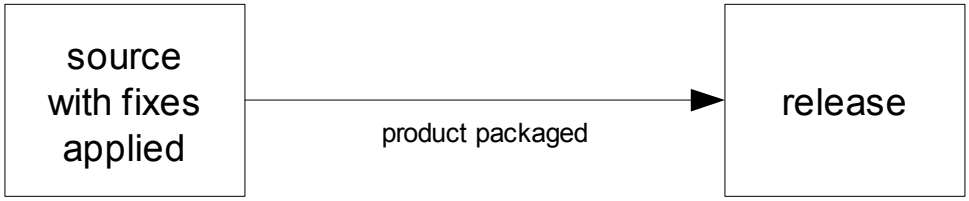

Figure 7: Packaging the Product

Once the defects (faults and failures) are fixed, those fixes are packaged into a release (Figure 7).

Bringing It Together

These work flows for handling defects can be connected together to create a single, coherent work flow for handling defects in the OSSDM. This work flow is called the **defect workflow** and is illustrated in Figure 8.

Enhancements

In addition to having defects, most open-source software products seek to enhance or refine their software offering, an activity analogous to perfective maintenance. New code that is added to a software product that adds a feature rather than resolving a defect is an **enhancement**. The process by which the OSSDM integrates enhancements is the **enhancement workflow**.

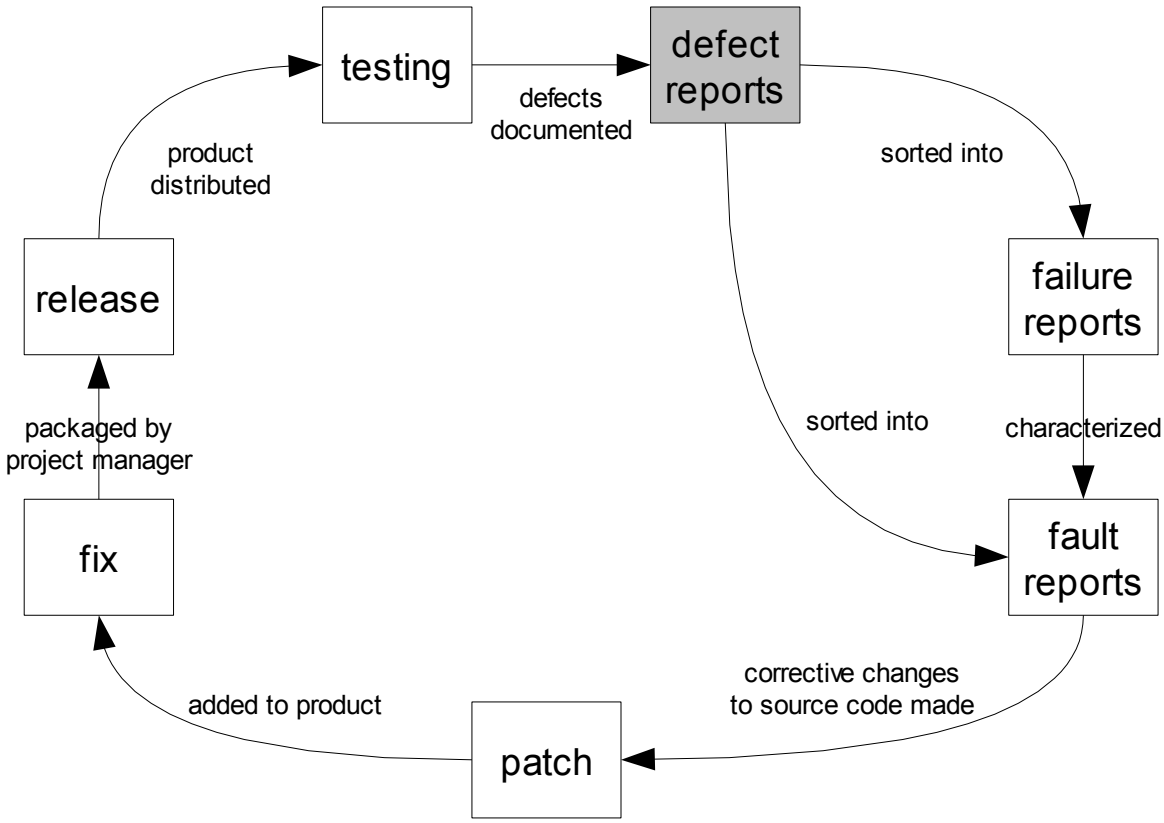

Figure 8: The Defect Workflow

Consider an arbitrary OSSDM project that has just released a version of its product with a new feature the next oldest release did not have. This new feature must have been added through changes to the product's source code, or **patches**. Similarly to how the OSSDM handles defects, these patches must have been integrated into the product's source code and then released (Figure 9). Similar to the defect workflow, the life of this enhancement can be split into two separate activities: the activities that occur before the patch is accepted (**pre-integration activities**) and the ones that occur after the patch is accepted (**post-release activities**) as shown in Figure 10.

Pre-integration, the development of the enhancement occurs outside of the OSSDM product's main source code tree. This development is therefore independent of

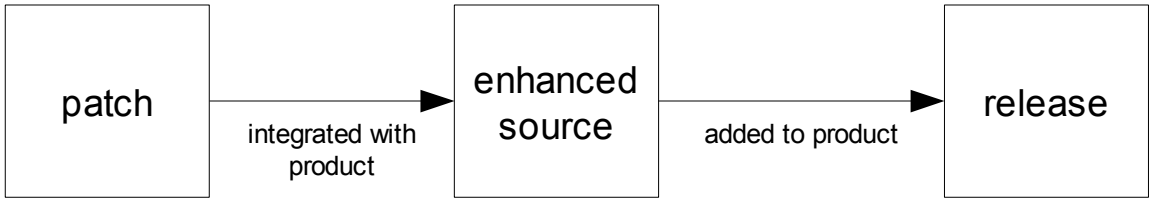

Figure 9: The Integration of an Enhancement

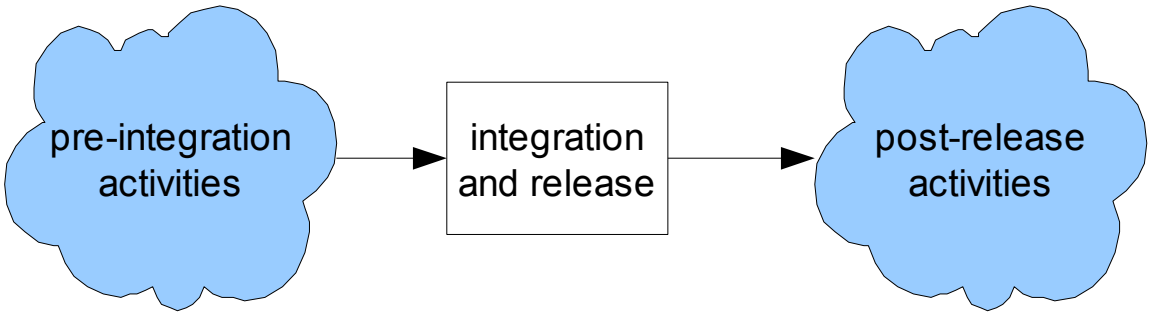

Figure 10: The Lifetime of an Enhancement

the continuing development of the rest of the OSSDM product. Because of this lack of mandated interaction, the OSSDM is silent on the exact method by which the enhancement is created; essentially, as long as the enhancement meets the standards set by the project manager, it does not matter by which process the enhancement was initially created.

Once the code is integrated into the OSSDM software product, that code and changes to it fall under the scope of the OSSDM. If no defects are discovered in this code, the code will live happily in the OSSDM product's source code repository and cost very little in terms of resources to maintain. If this code is ever changed post-integration, changes to that code are therefore maintenance and must fall in one of two classes:

- Corrective maintenance, or maintenance that fixes defects in the new enhancement. These defects are handled just like any other defect in the software product by the defect workflow.

- Perfective maintenance, or maintenance that extends the enhancement by adding new features. Because this type of maintenance in effect enhances the enhancement, its development activities are best described by the process detailed in this section.

Adaptations

Adaptive maintenance is maintenance that ports the software to a platform for which it was originally not designed. Maintenance performed under this umbrella is one of two types:

- Maintenance that corrects errors that prevent the software from working on the new platform. These changes fall under the scope of corrective maintenance and are handled by the defect workflow[6].
- Maintenance that modifies the product to take advantage of features available on the new platform that were not previously available. These changes fall under the scope of perfective maintenance and are handled by the enhancement workflow.

Applying Contribution Metrics

Contribution metrics are used to explore how contributors to an OSSDM project participate in the activities described above. These contribution metrics must divide contributors into groups that are significantly different, which is difficult to determine in general because all OSS projects are different. Therefore, setting a discrete line will

6 Theo de Raadt, project manager for the OpenBSD, noted with regard to porting OpenBSD to the SGI O2 architecture that "...every architecture we add has helped us find bugs in shared code that affects other architectures. [...] Some very scary and major bugs have been dredged out almost automatically." [De Raadt 2004]

cause difficulties, such as a case where an individual who contributed at level n would not be significant, whereas one who contributed at level $n+1$ would be.

Good arguments could be made for moving this line to include more or fewer people into the pool of significant contributors, so rather than trying to set hard thresholds for contribution metrics in general, it is more interesting to address the question of anomalous results. For example, consider an OSS project where for the last twenty versions defect report creation for a certain set of contributors (say the top 10%) remained relatively constant. Assume that for the next version, the data show that this value unexpectedly increased or decreased. If this value unexpectedly decreased, the value for the bottom 90% must have increased, a shift which could have been caused by an increase in knowledgeable non-core contributors to the software (among other things). If instead this value unexpectedly increased, this could signal a decrease in participation, or an increase in central control. Regardless, unexpected shifts in the values from version to version for contribution metrics indicate that something interesting has happened and warrants closer scrutiny.

Characterization and Trace Paths

Most of the activities described in the defect workflow and the enhancement workflow are straightforward in nature and need no further exploration. One activity, however, is more complex: the activity of **characterization**, or the mapping of failure reports to faults in the system. This characterization activity is documented by an artifact called a **trace path**.

To motivate this discussion, consider a hypothetical scenario. A particular browser has a defect in its HTML parsing routines that causes memory corruption when it is fed a particular bogus series of HTML tags. This overflow overwrites a small segment of the memory space the browser uses to store its DOM tree[7]. In turn, this DOM failure causes some Javascript-driven shopping carts to silently fail to submit orders correctly. Other testers find that a web-based email provider's website fails to load properly and causes the browser to use 100% of the host computer's CPU. This scenario has two interesting properties:

- One fault, the parsing overflow, leads to several different failures that do not appear to be related at first glance.
- The root fault, the parsing defect, is several steps removed from the failures that expose it, like layers of an onion.

The act of resolving this defect in involves three different activities. The first activity, performed by the tester, involves finding and documenting the software failures; this activity generates the original defect reports. In this example, the testers would use the product to be tested in their everyday browsing activities to spot defects. The second activity, performed by the maintainer, involves the actual source-level fixing of the faults that lead to the defects. The maintainer in this example would have access to the browser's source repository and write the code to fix the faults documented in the defect reports submitted by the testers.

The third activity, performed by the analyst, involves transforming the defect reports contributed by the testers into detailed fault profiles that can be processed by the

7 The DOM, or Document Object Model, is an API used to access documents structured using XML (or equivalent) programmatically. Programs typically use data structures such as trees to internally represent a DOM. For more information, see [Le Hegaret et al. 2000].

maintainers. In this scenario, the analyst examining the form submission problem might notice that the structure of the DOM tree for that site is corrupt using a DOM explorer tool. Upon further exploration, the analyst might find a sequence of HTML tags that causes such corruption. This process, akin in the above metaphor to peeling back the layers of a hypothetical onion, is a trace path. Formally, a trace path is the path discovered and traversed by analysts as they iteratively find the cause to software defects until they discover a defect that is a fault, not a failure.

Graphing Trace Paths

Because the act of following trace paths describes an iterative process (examining the cause of one failure reveals another failure, which is caused by another failure, and so forth) the relationship between faults and failures explored in this way can be modeled as a graph constructed in the following manner:

- A circle shape represents a failure, and a square represents a fault.
- A directed edge between two shapes represents an "is caused by" relationship. Note that a failure can be caused by either a fault or another failure, and that faults are always "first causes" and cannot cause other faults.
- Defects are labeled by unique numbers.

In this notation, a path that leads from a vertex with in-degree zero (representing a failure reported by the testers) to a vertex with out-degree zero (representing a fault) is a trace path.

Figure 11 presents a relatively simple scenario. Two failures, 1 and 2, have both been reported. These two failures are both caused by fault 3.

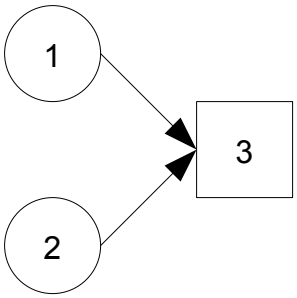

Figure 11: A Simple Scenario

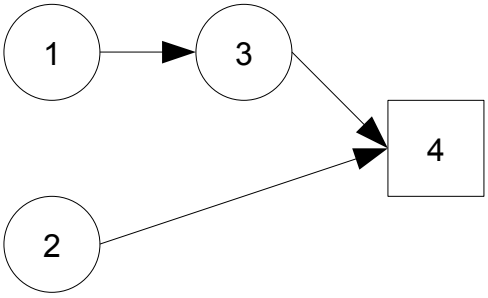

Figure 12: A More Complex Scenario

Figure 12 is a second scenario is a more complex case that models the example given in this section. Failure 1 (the shopping cart issue) and failure 2 (the webmail issue) are both ultimately caused by fault 4 (the overflow in the parsing routine). Note that the trace path leading from failure 1 to fault 4 is longer than the trace path from 2 because of the existence of failure 3, the corrupt DOM tree failure. An analyst starting from failure 1 would have to discover failure 3 before being led to fault 4.

"Debugging is Parallelizable"

The notion of trace paths also gives the tools necessary to show how Brooks's Law does not overly-burden OSSDM projects. Consider a large OSSDM project with many contributors and complex software. In such a system, the failures caused by a single fault might be documented in ten different defect reports that do not initially seem to be related. Assume that ten analysts begin to trace these failures back to their faults, and after one month they all discover and document the fault that caused them. This duplication of effort means that most of their work has been meaningless – one sufficiently-good defect report is as good as ten to a maintainer, who writes the code to fix the one defect that caused the ten failures.

Assume instead that all ten analysts began to trace these failures back to these faults, and after one week two of them had discovered and documented the defect that caused them. These fault reports were converted into fixes by a maintainer, which is rolled into the next release by the project manager that same day. The other eight analysts, rather than continuing to work with their old release, all download the new release and see if the failure they were originally analyzing still exists. If it still exists, they can continue to trace that fault in the old version or the new version. However, if that failure no longer exists, they know that it must have been fixed by some change made in the new version, allowing them to refocus their attention on another failure. While in the first example the time of ten analysts was used to fix the fault, in the second example eight of the ten analysts were able to move on after the first week. Analyst resources are conserved through this rapid release strategy, preventing the explosion of inefficiency that Brooks's Law would predict. However, releases that are too rapid come with a cost – the analyst must retest the original failure with each new release.

CHAPTER IV

A MOZILLA CASE STUDY

The model presented in Chapter III makes a series of claims that should hold true for any applicable open-source software project. This section presents the Mozilla project as a case study of a project that fits to the OSSDM. First, this section shows that the Mozilla project is observable enough to draw applicable contribution metrics. Then this section decomposes the claims of Chapter III and explores them using data gleaned from Mozilla's development.

Applicability of the OSSDM

For the OSSDM to be applicable to Mozilla, measurable contribution metrics must be derivable. Fortunately, the Mozilla project's history is laid bare in the form of Internet-accessible databases. The primary databases from which empirical measurements can be extracted are the Mozilla Bugzilla database and the Mozilla Bonsai database.

The Mozilla Bugzilla Database

The Mozilla Bugzilla database documents the development activities of the Mozilla team [Mozilla Foundation 2005c]. The atoms of the Bugzilla database are what

are known as **bugs**[8] (a report of a particular fault, failure, or enhancement request). These bugs have attached **commentary** (remarks made by other users or the Mozilla developers); one item of commentary attached to a defect report is a **comment**. Examples of data that can be extracted from this database include:

- Absolute number of defect reports created per contributor per release.
- Non-duplicate[9] number of defect reports created per contributor per release.
- Comments created per contributor per release.

The Mozilla Bonsai Database

The Mozilla Bonsai database is a database that provides access to the complete logs of the Mozilla CVS server [Mozilla Foundation 2005b]. Any addition, removal, or modification of source code in is logged by the Bonsai system. Examples of data that can be extracted from the Bonsai system include:

- Distinct number of patches submitted by a particular contributor per release.
- Number of lines of code submitted by a particular contributor per release.

The Producer

Showing that the Mozilla project is an OSSDM project requires showing that project managers create releases and that testers create defect reports. Testers will be identified first. There are at least two classes of a tester – one who contributes defect

8 The term bug is overloaded. For the purposes of this thesis, bug will always refer to a particular main topic entry in the Mozilla Bugzilla database. When referring to a particular failure or fault, we will use the term defect instead.

9 The term **duplicate** has a special meaning when applied to defect reports. This concept will be explored later in this chapter.

reports and also participates in other ways, and another who contributes defect reports but does not otherwise contribute substantially. This second group of testers, the **pure testers**, is identified by showing that a class of participants exists who contribute defect reports but do not contribute commentary. The contribution metric that shows this is calculated by adding the number of defect reports created per contributor to the number of comments created per contributor, and ranking contributors by their resulting score.

These contribution metrics are evaluated by processing Mozilla Bugzilla data. For each of the first sixty-three versions of Mozilla, the number of defect reports and comments created per contributor were tallied and contributors sorted in order from greatest level of contribution to least level of contribution, and then sorted into percentile buckets based on the contributor's relative level of contribution for that version. The level of contribution for each of the buckets were then converted into a fraction of the total contribution, and those fractions were averaged over the first sixty-three versions of Mozilla to produce Figure 13 (and Table 1, a detail of Figure 13).

Judicious application of the 80/20 rule is enlightening: the bottom 80% of participants (as ranked by total participation) contribute 51% of the defect reports but only 13% of the commentary. Furthermore, the bottom 20% of participants contribute a number of defect reports generally in proportion with the size of their class (19%) but contribute an insignificant number of comments. These data show a clear division between at least the top 80% (and probably the top 20%) of contributors and everyone else. This lower class of contributors make up the class of **pure testers**.

The other group of actors addressed in the producer side of the software development activity are the project managers. The dual responsibility of project managers, those of being caretakers and reviewers, are analogous to the Mozilla concepts

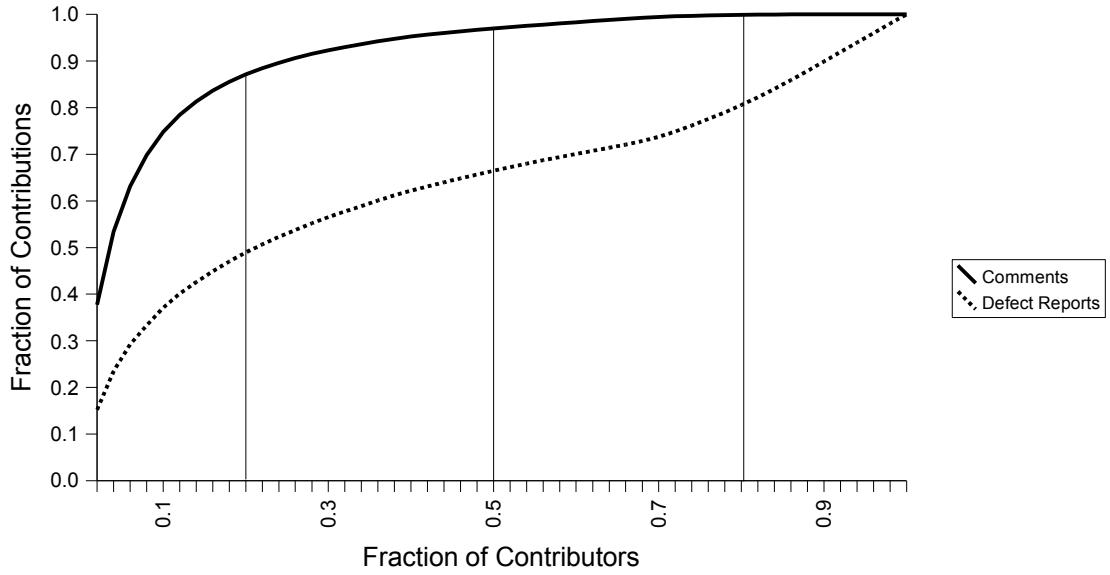

Figure 13: Cumulative Distribution of Contributions (Defect Reports and Comments) to Mozilla Bugzilla. (Number of Versions = 63)

Table 1: Detail of Figure 13 -- Cumulative Distribution of Contributions (Defect Reports and Comments) to Mozilla Bugzilla. (Number of Versions = 63)

Contribution Percentile	% Comments	% Defect Reports
Top 20%	87% ± 5%	49% ± 16%
Top 50%	97.0% ± 1.4%	66% ± 14%
Top 80%	99.9% ± 0.02%	81% ± 7%

of module ownership and super-review. To explain how this works, consider the Mozilla process of getting a patch into the product source as described in the *Hacking Mozilla* document ([Mozilla Foundation 2005a]). First, a patch for a particular module is reviewed by the owner of that module, or one of his or her peers. Then, if that patch affects in-process code (code that is not stable or otherwise well-known), the code is super-reviewed by a "strong hacker" to ensure code quality. Once this is done, the code can be checked in to Mozilla CVS.

Even if the process is spelled out in product documentation, the OSSDM requires a contribution metric; fortunately, the Mozilla project makes available a list of module owners, their peers, and individuals capable of being super-reviewers. Combined, these lists identify roughly 250 unique individuals that fill project management roles. Furthermore, the level of participation of project managers is countable because the Mozilla developer guide calls for code review to be documented inside a defect report. Assuming that all of the project managers participate in the project each release, the project managers would only make up 14% of the total contributors for each release on average[10]. This shows that, rather than many or all contributors being project managers, project managers are at most a small subset of all contributors.

The Consumer

The consumer side of the equation has two activities that have not been previously addressed: the characterization task, where defect reports are sorted into fault reports and failure reports, and the patch-creation task, where patches are made for defects in the source code.

To address the first activity, that of characterization, a metric must be found that uniquely identifies characterizers. Fortunately, the Mozilla Bugzilla system records artifacts of the characterization process: comments. During the course of defect resolution, contributors add commentary to defect reports. The nature of this commentary varies, but much of it has to do with whether a bug is a duplicate of another – any defect marked as DUPLICATE will have at least one comment stating that fact,

10 In actuality, many modules are stable or otherwise dormant; it would be rare for every project manager to contribute every release.

Figure 14: A Duplicate Bug in Mozilla

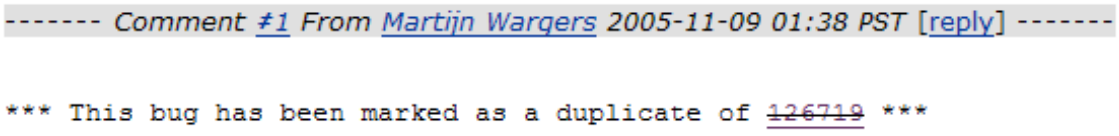

Figure 15: Duplicate Commentary

and as previously shown duplicate defect reports are artifacts of the defect resolution process (Figure 14 and Figure 15).

Because of the above, comments end up being a good indicator of a contributor's level of participation in the characterization task, making it possible to create a contribution metric to separate those contributors that perform the characterization task from those that do not. This metric itself is defined in terms of how many comments a group of contributors makes compared to the number that would be expected if every contributor contributed equally. For example, if every contributor contributed equally, any group of contributors that made up 10% of the total would submit 10% of the comments; this group would be assigned the value "1" to represent their level of

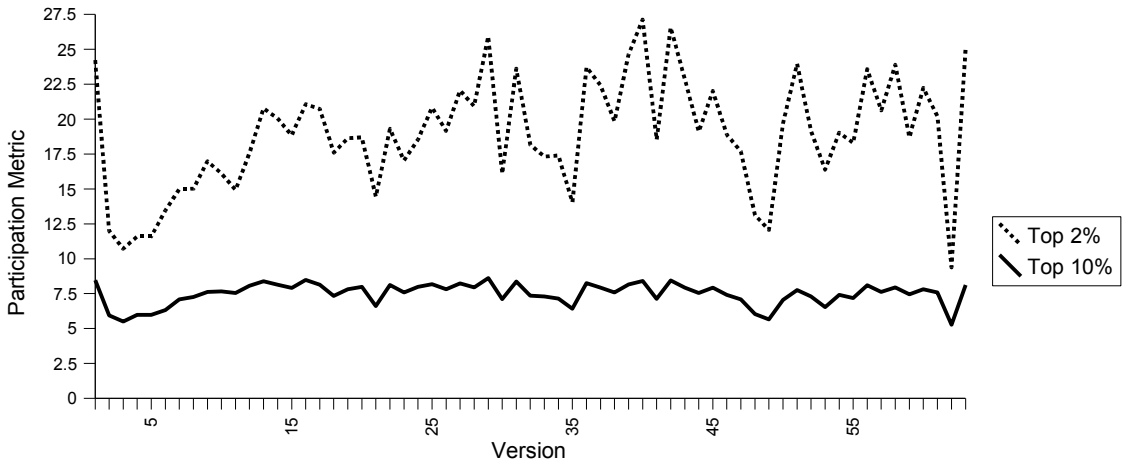

Figure 16: Mozilla Bugzilla Comment Participation of Top 2% and the Top 10% of Contributors. (Number of Versions = 63)

contribution. If instead that same group submitted 20% of the comments, that group would be assigned the value "2", as they performed twice as much work as baseline.

Figure 16 charts this metric for the top 2% and the top 10% of contributors to the Mozilla project. The data show that the top 2% and top 10% are wildly more productive than their numbers would suggest, with the top 2% producing an average of 18.88 ± 4.07 times more commentary than baseline would predict (37.8% ± 8.1% of the total commentary). The top 10% also fare well, registering 7.48 ± 0.80 on the same scale (representing a whopping 74.8% of the total commentary). This suggests that, just as bottom 50% and 80% of contribute submit a disproportionate number of defect reports, the top 10% and 2% of contributors perform a disproportionate amount of analysis of those defect reports and are the characterizer actors[11].

The other activity that makes up the consumer part of the equation concerns itself with creating patches that fix defects. The integration of these patches into the Mozilla

11 The large standard deviation in the level of participation for the top 2% of contributors, ±22%, is in part due to the small size of that group, consisting of only 35 ±18 individuals.

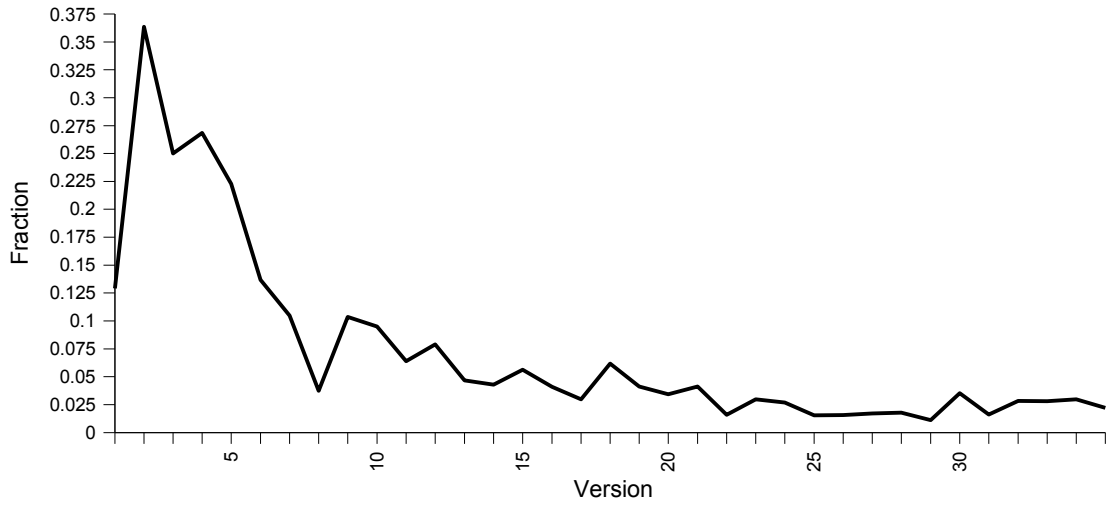

Figure 17: Contributors with CVS Access as a Fraction of All Contributors.(Number of Versions = 35)

source code tree is tracked by the Mozilla Bonsai system, which lists among other things each contributor's unique identifier and a summary of the changes made by the patch. The number of patches submitted by each unique contributor can therefore be counted, and a contribution metric derived.

Because source repository access in the Mozilla project is controlled, the contribution metric "contributors who have access to the Mozilla CVS per version" is applicable. Figure 17 shows the number of contributors with CVS access as a fraction of all contributors for each of the first thirty-five versions of Mozilla. From this figure it is clear that there are relatively few contributors with CVS access when compared to all contributors (75.49 ± 34.05 vs. 1681.4 ± 848.82), showing a clear division between those contributors with CVS access and those without. Furthermore, there is stratification even within the circle of contributors with access. Figure 18 with contributors ordered in decreasing level of contribution, is a typical example: in this

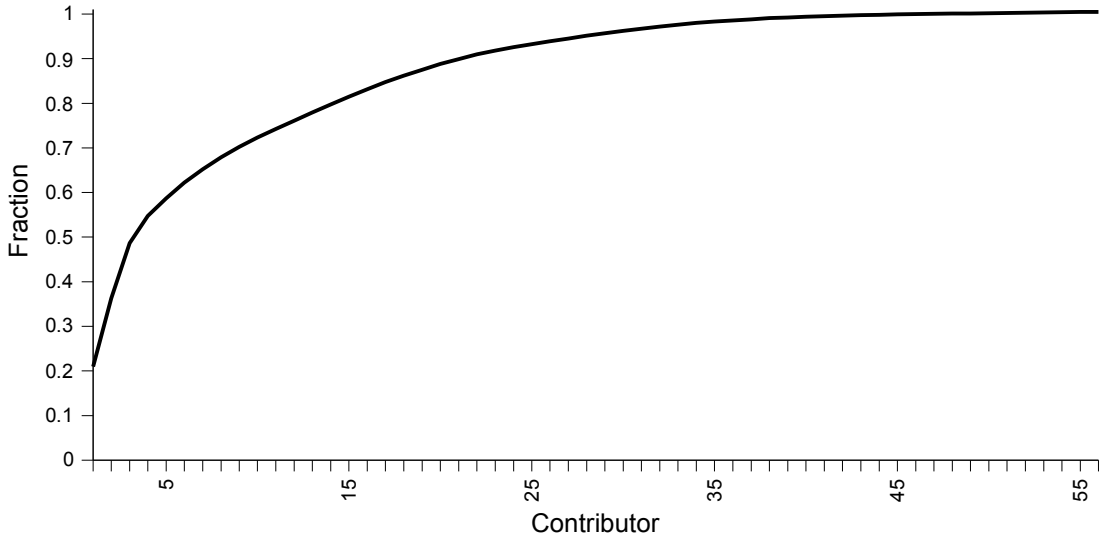

Figure 18: Cumulative Distribution of Contributions to Mozilla CVS. (Version 23, Number of Contributors = 56)

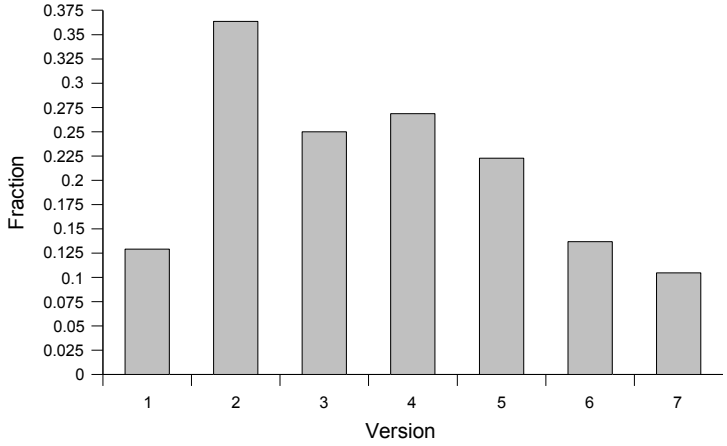

Figure 19: Fraction of Contributors with CVS Commit Access for Selected Versions of Mozilla

version, the top 20% of contributors made 71.8% of the commits to the source repository (11 contributors, 1419 commits).

A particularly interesting inconsistency in the Mozilla Bonsai data comes from data drawn from the first seven versions of Mozilla (Figure 19). For these versions, an average of 21% of the contributors had CVS access, while the average for the next

twenty-eight versions is only 4%. A justification for this sharp difference is the Mozilla project's unusual origins. Generally, OSS projects add developers as they add features and attract more users to their software product. Mozilla, on the other hand, leaped from the womb fully-formed, with a full complement of professional engineers on Netscape's payroll tasked to encourage its growth. This imbalance was rectified as the Mozilla team created releases for its users to test and provide feedback on, which in turn resulted in better software, which in turn attracted more users, and so on.

Trace Paths, Characterization, and Duplicates in Mozilla

The Mozilla project thoughtfully provides a wealth of information through which to explore the characterization process. This chapter has previously asserted that the number of comments created by a contributor is a good predictor of that contributor's participation in the characterization task. This assertion is justified by an analysis of duplicates and their effect on the Mozilla product's development.

Informally, the notion of a duplicate is the notion of repetition – for example, two Mozilla defect reports that are mere copies of one another can be considered duplicates. This particular sort of duplication is frowned upon because of its wasteful nature. However, other forms of duplication are both necessary and desired because they can have positive side-effects on the development of the software product. Formally, a **duplicate** is a defect report that documents a fault that has been previously reported, or that documents a failure whose parent fault has been previously reported. Conversely, a **nonduplicate** is the defect report that duplicates are considered to be duplicating.

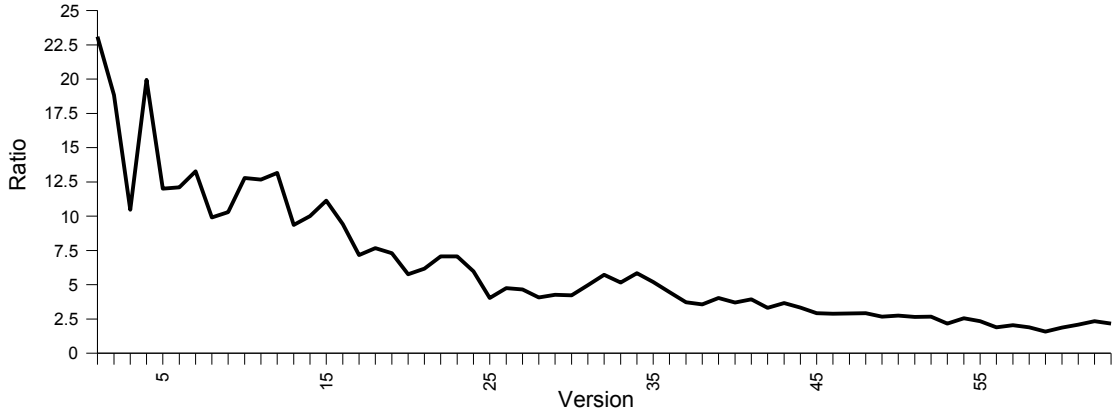

Figure 20: Ratio of Duplicate Defect Reports to Non-Duplicate Defect Reports for Versions of Mozilla. (Number of Versions = 63)

It would a waste of time to examine duplicate defect reports if they did not significantly affect defect report processing; it is telling then to note that the Mozilla project takes the problem of duplicates seriously enough to deeply integrate sophisticated duplicate tracking in its Bugzilla defect report tracking tool. Analysis of these data show the extent of the duplicate problem: in later versions of Mozilla, five defect reports were resolved DUPLICATE for every two that were not; in older versions this ratio could be twelve to one or worse (Figure 20).

Understanding Mozilla Duplicates

Clearly, duplicates present a question for the Mozilla project that must be addressed. Fortunately, the framework presented in Chapter III gives a way by which they can be understood. As has already been addressed, a trace path is an artifact that graphs the interaction effects between faults and failures. During the process of

characterization, these trace paths are traversed by analysts, who resolve the defect report to be either a fault report (and therefore a nonduplicate) or a failure report (and therefore necessarily a duplicate). When resolved in this way, defect reports in Mozilla's Bugzilla system are tagged with the defect report of which they are a duplicate and marked as DUPLICATE. A side-effect of the tagging process is that a comment is generated that records the ID of the tagger and the time and date of the tagging.

Furthermore, the DUPLICATE tag is applied to what we have identified as two distinct classes of duplicate defect reports in the Mozilla project. The first class, the **temporal** class, is where a defect report is a mere copy of one that had been documented in the past. The second class, the **causal** class, is where report documents a failure that has an "is caused by" relationship with another fault or failure. Given these two classes of duplicates, our definition of a duplicate report can be made more precise:

- A **shallow duplicate** is a defect report whose relationship with its mate or mates is one of mere repetition.
- A **deep duplicate** is a defect report that seems unique upon first examination but actually is not.

Trace Paths in Mozilla

Because shallow duplicates are simply time-shifted copies of an original report, they are relatively uninteresting outside of the amount of time contributors use to identify them. Deep duplicates are a more interesting case because the causal relationship between deep duplicates; the causal relationship between deep duplicates suggests applying the graphical language described in Chapter III to trace paths in Mozilla. This

graphical language is much like the one previously described, with the following modifications:

- A number in a shape uniquely identifies a defect report. A single particular defect may be reported more than once (i.e., may have shallow duplicates); these are omitted for clarity.
- A letter in a shape uniquely identifies a defect that has not been documented.

Figure 21 is a simple example of this notation. Failures 1 and 2, which are both documented and are not shallow duplicates, both have their root cause in undocumented fault *a*. Figure 22 is a more complex case. Failures 3 and 4 are both known and both have their root cause in fault 5, which has been documented However, failure 3 is caused by an intermediate failure, *b*, which is in turn caused by the root fault 5. A characterizer resolving the trace path beginning at 3 would have to discover and document failure *b* before he or she could know that fault 5 is the root cause of failure 3.

Figure 23 shows a trace path from Mozilla Bugzilla. Bug #52798 documents a page reflow error that occurred when a GIF image meeting certain specifications was present on the rendered page. This defect caused the GIF image in question to flicker and the browser to consume more CPU than normal. This bug has fourteen duplicates, ten shallow and four deep. The deep duplicates in this case are the ones that are judged qualitatively to add to the knowledge base regarding the source defect report rather than being mere duplicates -- #56015 and #56082 are the first to document that the defect can cause Mozilla to consume 100% CPU, while defect report #65614 documents a test case where the failure is triggered without the use of HTML table tags. We also remark that Bug #52798 documents a failure, not a fault, meaning that the bug is conceptually a duplicate of an undocumented defect report, represented by *a* in the graph.

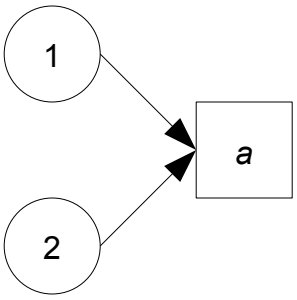

Figure 21: A Simple Scenario

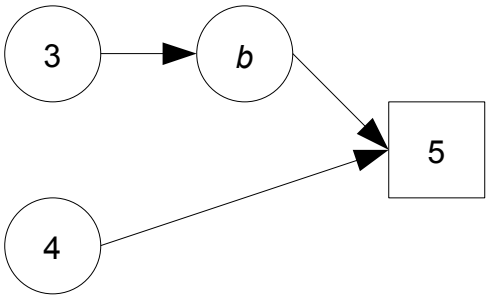

Figure 22: A More Complex Case

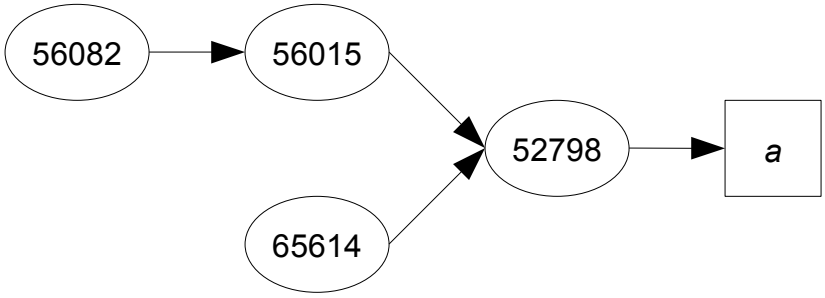

Figure 23: A Trace Path from Mozilla Bugzilla

Trace Paths and Duplicate Resolution

Often, reported defects that developers are trying to resolve simply vanish when a new version of the software is released. This happens because the defect report with which the developers were working was a deep duplicate of a fault that was resolved by another developer in the interim rather than a non-duplicate. (In effect, multiple developers were working on multiple trace paths of the same fault at the same time.) Given our definition of duplicate resolution, defect reports that become obsolete after a new version of the software is released are also successfully resolved as duplicates.

The comment shown Figure 24 illustrates this. Mozilla Bugzilla bug #67196 documents a failure that caused a table to not be rendered correctly on a particular website. The defect was explored and thought to be related to a problem with the browser's CSS parsing routines, but this avenue was not immediately explored and the

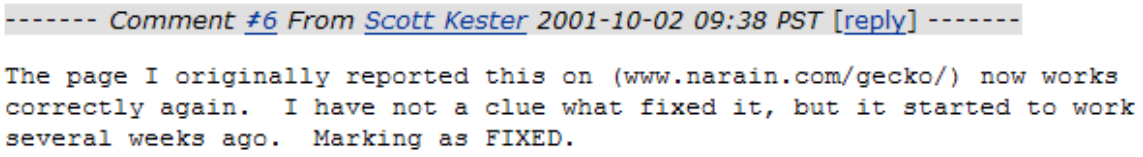

Figure 24: A Comment from Bug #67196

bug went dormant for a time. When the defect was retested at a later date, it was no longer reproducible, meaning that it was fixed by some patch integrated into the product in the interim. Furthermore, it is almost impossible to determine exactly which patch (and the bug that documents it) fixed this defect because many patches are integrated into each version of Mozilla, and many versions of Mozilla were released between the time in which the bug was opened and the contributor made the comment in Figure 24.

This connection between trace paths, duplicates, and duplicate resolution is further explored in Chapter IV.

CHAPTER V

DISCUSSION AND CONCLUSIONS

Defects are the enemy of software reliability. Because of this, it can seem counterintuitive to use defects as a lens through which to view software development. However, it is the obsession that classical development models have with reducing both the number and severity of defects that makes closer analysis of the defects themselves attractive. It is this, coupled with the easy access to data afforded by OSS projects, that makes the defect itself a powerful analytical tool.

This tool is used in Chapter III to document a defect-centric style of development. Though this style is described from a fundamentally different viewpoint than classical models, its validity is corroborated by the fact that this model and classical models share important structural components such as actors and activities, just with different scaffolding. Furthermore, Chapter III also describes a framework for understanding cases where the data do not fit the model by showing how its predictions are contingent on projects following Torvald's OSS development philosophy.

Chapter IV uses different views of the data published by the Mozilla project both to analyze claims of Chapter III and to plumb the depths of the Mozilla project itself. Most data show a remarkable level of stratification between roles, as predicted by the defect-centric model, the notable exception happening during a unique time in the Mozilla project's history..Furthermore, the parallelization of debugging using trace paths

presented in Chapter III and explored in Mozilla in Chapter IV shows how every actor in an OSSDM project must be cognizant of how defects affect every step of development.

CHAPTER VI

FUTURE WORK

Open-source software and OSS software engineering is a relatively new and open research field. This section touches on several areas of future research suggested by this work.

The Failure of Open-Source Projects

An interesting consequence of the model of defect handling presented above is that it can be construed to make some predictions as to what would cause the OSS development process to fail. Basically, if any link in the workflow presented in Chapter III were to fail, the ability of the OSS development process to function would be greatly hampered. Potential sources for failure include:

- Being too difficult for reporting users to document defect reports and communicate them to the developers. This would cause the project to starve through lack of feedback.
- Being too difficult for developers to process feedback and identify which defect reports are duplicates and which correspond to real faults in the source code. This would cause the project to starve through the lack of developer resources.

These sources for failure could be used to create a set of metrics for determining empirically whether an OSS product has "failed."

Duplicate Tagging Behavior

As time passes after a defect report has been submitted to Mozilla's Bugzilla system, contributors develop a certain familiarity with the failure or failures the defect report describes. This growing familiarity should tend to make the pool of contributors that could accurately tag the defect report as a DUPLICATE increase over time. Conversely, if a defect is not well known, characterization predicts that work on that defect will proceed in parallel and will be documented in several different Bugzilla bugs, which should eventually be marked as duplicates of one another as their causes are divined. This suggests that in a graph of a non-duplicate and its shallow duplicates should be wide and shallow, while a graph of a non-duplicate and its deep duplicates should be narrow and deep. If these assertions are valid, then it should be possible to analyze the Mozilla Bugzilla duplicate data and quantitatively determine which duplicates are shallow and which are deep (as opposed to the qualitative analysis performed in Chapter IV).

Optimizing Release Cycles

Chapter IV shows in part that duplicates can be resolved simply through the release of a new version of the software. What is interesting is that the frequency of releases strongly affects the number of duplicates that can be resolved in this manner. Releasing too frequently frustrates developers by making the platform a moving target for resolving trace paths. Similarly, releasing new versions of the software too infrequently could cause developer resources to be squandered by having developers follow trace paths that are actually duplicates for a longer time than they would have otherwise.

This suggests that there should be a happy medium between releasing too frequently and too infrequently; this is interesting because of the different schedules OSS projects use for creating releases of their software. For example, Mozilla creates nightly releases that developers use to see if faults have been resolved. However, this strategy is not practical for projects like KDE[12], as it is not reasonable to expect your reporting users to install a new version of their desktop environment on a daily basis to test for defects. Finding the optimal release cycle for a project such as KDE that would conserve the most developer resources without frustrating testers could result in a significant productivity boost.

12 KDE is the K Desktop Environment, a "network transparent contemporary desktop environment for UNIX workstations." [K Desktop Environment 2005]

APPENDIX A

MOZILLA VERSIONS

This table lists the vital statistics of the versions of Mozilla used in this paper. Each row in the table corresponds to a single release of Mozilla's development branch.

- **AV**: Absolute version of this Mozilla version release, beginning with number one (m3).
- **V**: Canonical version name of this release; the name that the Mozilla project assigned to this release.
- **Release Date**: The date of this release of Mozilla, as recorded from the Mozilla release information pages.
- **Days Released**: The number of days between the release of the previous version of Mozilla and this version of Mozilla. Essentially, the number of days that the previous version was the newest version of Mozilla.

Table 2: Mozilla Vital Statistics

AV	Version	Release Date	Days Released
1	m3	03/19/99	N/A
2	m4	04/15/99	27
3	m5	05/05/99	20
4	m6	05/29/99	24
5	m7	06/22/99	24
6	m8	07/16/99	24
7	m9	08/26/99	41
8	m10	10/08/99	43
9	m11	11/16/99	39
10	m12	12/21/99	35

AV	Version	Release Date	Days Released
11	m13	01/26/00	36
12	m14	03/01/00	35
13	m15	04/18/00	48
14	m16	06/13/00	56
15	m17	08/07/00	55
16	m18	10/12/00	66
17	0.6	12/06/00	55
18	0.7	01/09/01	34
19	0.8	02/14/01	36
20	0.8.1	03/26/01	40
21	0.9	05/07/01	42
22	0.9.1	06/07/01	31
23	0.9.2	06/28/01	21
24	0.9.3	08/02/01	35
25	0.9.4	09/14/01	43
26	0.9.5	10/12/01	28
27	0.9.6	11/20/01	39
28	0.9.7	12/21/01	31
29	0.9.8	02/24/02	65
30	0.9.9	03/11/02	15
31	1.0rc1	04/18/02	38
32	1.0rc2	05/10/02	22
33	1.0rc3	05/23/02	13
34	1	06/05/02	13
35	1.1a	06/11/02	6
36	1.1b	07/22/02	41
37	1.1	08/26/02	35
38	1.2a	09/11/02	16
39	1.2b	10/16/02	35
40	1.3	12/02/02	47
41	1.3a	12/13/02	11
42	1.3b	02/10/03	59
43	1.3	03/13/03	31
44	1.4a	04/01/03	19
45	1.4b	05/07/03	36
46	1.4rc1	05/29/03	22
47	1.4rc2	06/17/03	19

AV	Version	Release Date	Days Released
48	1.4rc3	06/24/03	7
49	1.4	06/30/03	6
50	1.5a	07/22/03	22
51	1.5b	08/27/03	36
52	1.5rc1	09/17/03	21
53	1.5rc2	09/26/03	9
54	1.5	10/15/03	19
55	1.6a	10/31/03	16
56	1.6b	12/09/03	39
57	1.6	01/15/04	37
58	1.7a	02/23/04	39
59	1.7b	03/18/04	24
60	1.7rc1	04/21/04	34
61	1.7rc2	05/17/04	26
62	1.8a1	05/20/04	3
63	1.8a2	07/14/04	55

APPENDIX B

SOURCE CODE

This appendix contains source code listings for the scripts used to acquire and analyze Mozilla data. All of these programs use Python, a cross-platform scripting language ([Python Software Foundation 2005]).

BugzillaTools.py

BugzillaTools.py consists of a collection of four classes used to encapsulate searching Bugzilla. There are two query objects and two search objects; properties of a particular defect report are scraped from the Bugzilla page using regular expressions and are stored in instance variables of the BugzillaBug class.

```
#
# BugzillaTools.py
# Utilities for building, getting properties of, and executing Bugzilla
queries.
# Brandon Nuttall
# b.nuttall@vanderbilt.edu
#

import urllib2
import re
from Counter import Counter

class InvalidBugURLError( Exception ):
    def __init__( self, value ):
        self.value = value
    def __str__( self ):
        return repr( self.value )

class BugzillaQuery( object ):
    def __init__( self ):
        self.url = ""
    def __doQuery( self ):
```

```
        """
        Execute the query for this query object
        """
        data = ""
        for line in urllib2.urlopen( self.url ):
            data = data + line
        return data

class BugzillaSearchQuery( BugzillaQuery ):
    """
    Encapsulates a Bugzilla query
    """
    def __init__( self, startDate, endDate ):
        self.startDate = startDate
        self.endDate = endDate
    def doQuery( self ):
        """
        Executes the Bugzilla query encapsulated by this object
        """
        self.url =
"http://bugzilla.mozilla.org/buglist.cgi?query_format=&short_desc_type=
allwordssubstr&short_desc=&long_desc_type=substring&long_desc=&bug_file
_loc_type=allwordssubstr&bug_file_loc=&status_whiteboard_type=allwordss
ubstr&status_whiteboard=&keywords_type=allwords&keywords=&bug_status=RE
SOLVED&bug_status=VERIFIED&bug_status=CLOSED&resolution=FIXED&resolutio
n=DUPLICATE&emailassigned_to1=1&emailtype1=exact&email1=&emailassigned_
to2=1&emailreporter2=1&emailqa_contact2=1&emailtype2=exact&email2=&bugi
dtype=include&bug_id=&votes=&chfieldfrom=" + self.startDate +
"&chfieldto=" + self.endDate +
"&chfield=%5BBug+creation%5D&chfieldvalue=&cmdtype=doit&order=Reuse+sam
e+sort+as+last+time&field0-0-0=noop&type0-0-0=noop&value0-0-0="
        return BugzillaSearch( self._BugzillaQuery__doQuery() )

class BugzillaBugQuery( BugzillaQuery ):
    """
    Encapsulates a Bugzilla query
    """
    def __init__( self, bugID ):
        self.bugID = bugID
    def __buildURL( self ):
        """
        Creates the URL that will be used for the Bugzilla query
        """
    def doQuery( self ):
        """
        Executes the Bugzilla query encapsulated by this object
        """
        self.url = "http://bugzilla.mozilla.org/show_bug.cgi?id=" +
self.bugID
        return BugzillaBug( self.bugID, self._BugzillaQuery__doQuery() )

class BugzillaSearch( object ):
    def __init__( self, data ):
        """
        Parse the data of this search result
        """
        p = re.compile( r'show_bug\.cgi\?id=(\d+)"' )
```

```
        for m in p.finditer( data ):
            bugQuery = BugzillaBugQuery( m.group( 1 ) )
            try:
                self.bugQueries.append( bugQuery )
            except AttributeError:
                self.bugQueries = [ bugQuery ]

    def __getBugNum( self, line ):
        """
        Extract the bug number from this line of data from Bugzilla
        """
        return ( line.split( '"' ) )[ 1 ]

    def queries( self ):
        """
        returns list of bugQueries
        """
        return self.bugQueries

    def bugIDs( self ):
        return [ bugQuery.bugID for bugQuery in bugQueries ]

class BugzillaBug( object ):
    def __init__( self, bugID, data ):
        """
        Parse the data of this bug
        """
        self.bugID = bugID
        p = re.compile( r'Reporter:</b>.*?<a
href="mailto:(.*?)".*?Status</a>:.*?<td>(.*?)</td>.*?Resolution</a>:.*?
<td>(.*?)[</td>|\s]', re.DOTALL )
        m = p.search( data )
        try:
            self.reporter   = m.group( 1 )
            self.status     = m.group( 2 )
            self.resolution = m.group( 3 )
      except AttributeError:
            self.reporter   = None
            self.status     = None
            self.resolution = None

        p = re.compile( r'Additional Comment.*?mailto:(.*?)"',
re.DOTALL )
        for m in p.finditer( data ):
            try:
                self.commentCounter.add( m.group( 1 ) )
            except AttributeError:
                """
                commentCounter not yet created
                """
                self.commentCounter = Counter( m.group( 1 ) )

        """
        Bugs that have not been looked at yet can sometimes have
        blank resolutions
        """
        if self.resolution == "":
```

```
            self.resolution = "NOT_YET_RESOLVED"

    def info( self ):
        """
        Returns the id, status, and res of this bug
        """
        return ( self.bugID, self.reporter, self.status,
self.resolution, self.commentCounter )


if __name__ == "__main__":
    """
    Unit testing
    """
    searchQuery = BugzillaSearchQuery( "1999-02-15", "1999-02-20" )
    searchResults = searchQuery.doQuery()
    for bugQuery in searchResults.queries():
        bug = bugQuery.doQuery()
        print bug.info()[ 0 ], bug.info()[ 4 ]
```

doAnalysis.py

```
from urllib2 import HTTPError, URLError
from httplib import BadStatusLine
from BugzillaTools import BugzillaSearchQuery
from Counter import Counter

def dumpCounter( counter, fileName, mode ):
    """
    Dump the contents of a Counter to the specified file
    """
    counterFile = open( fileName, mode )
    for key, value in counter.counts( True ):
        counterFile.write( str( key ) + "," + str( value ) + "," + "\n"
)
    counterFile.close()

def getNextDate( releaseDates ):
    """
    Returns the first noncommented date in the list of release dates
    """
    date = releaseDates.pop( 0 ).strip()
    numSkipped = 0
    while date[ 0 ] == "#":
        date = releaseDates.pop( 0 ).strip()
        numSkipped += 1
    return date, numSkipped

if __name__ == "__main__":
    """
    Do the analysis operation
    """
    releaseDatesInfile = open( "./releasedates", "r" )
```

```
    for line in releaseDatesInfile:
        try:
            releaseDates.append( line )
        except NameError:
            releaseDates = [ line ]

    reporterCounter   = Counter()    # counts how many bugs reported
    commentCounter    = Counter()    # counts how many comments made
    fixedCounter      = Counter()
    invalidCounter    = Counter()
    wontfixCounter    = Counter()
    duplicateCounter  = Counter()
    worksformeCounter = Counter()
    movedCounter      = Counter()
    otherResCounter   = Counter()

    #
    # The version number is kept track of in the version variable.  The
script
    # is designed so that if it terminates, it can be restarted easily
by
    # commenting out dates in the releasedates file.  The getNextDate
returns
    # as its second return val the number of dates that were commented
out
    # before it found a legit one.  On a virgin file, this will be 0;
it is
    # incremented at the head of the below loop.
    #
    startDate, version = getNextDate( releaseDates )
    while releaseDates:
        #
        # Pull and tabulate the information for each Mozilla version
        #
        endDate, numSkipped = getNextDate( releaseDates )
        version = version + 1 + numSkipped
        search = BugzillaSearchQuery( startDate, endDate ).doQuery()
        queries = search.queries()
        for bugQuery in queries:
            print str( version ) + ": " + bugQuery.bugID,
            #
            # Pull and tabulate data for each bug in Bugzilla query
            #
            try:
                bug = bugQuery.doQuery()
            except ( HTTPError, URLError, BadStatusLine ):
                #
                # The lookup for the url for this query failed; add to
                # retry list and reloop (will end up retrying)
                #
                queries.append( bugQuery )
                print ", failed"
                continue
            else:
                print

            reporterCounter.add( bug.reporter )
```

```
            try:
                commentCounter.merge( bug.commentCounter )
            except AttributeError:
                #
                # A commentCounter was never created for this bug,
                # which means that the bug must have no comments
                #
                pass

            if bug.resolution == "FIXED":
                fixedCounter.add( bug.reporter )
            elif bug.resolution == "INVALID":
                invalidCounter.add( bug.reporter )
            elif bug.resolution == "WONTFIX":
                wontfixCounter.add( bug.reporter )
            elif bug.resolution == "DUPLICATE":
                duplicateCounter.add( bug.reporter )
            elif bug.resolution == "WORKSFORME":
                worksformeCounter.add( bug.reporter )
            elif bug.resolution == "MOVED":
                movedCounter.add( bug.reporter )
            else:
                otherResCounter.add( bug.reporter )

        #
        # Dump information in the counters to file
        #
        reporterFileName   = "./" + str( version ) + "-reporter"
        commentFileName    = "./" + str( version ) + "-commenter"
        fixedFileName      = "./" + str( version ) + "-fixed"
        invalidFileName    = "./" + str( version ) + "-invalid"
        wontfixFileName    = "./" + str( version ) + "-wontfix"
        duplicateFileName  = "./" + str( version ) + "-duplicate"
        worksformeFileName = "./" + str( version ) + "-worksforme"
        movedFileName      = "./" + str( version ) + "-moved"
        otherResFileName   = "./" + str( version ) + "-otherres"

        dumpCounter( reporterCounter, reporterFileName, "w" )
        dumpCounter( commentCounter, commentFileName, "w" )
        dumpCounter( fixedCounter, fixedFileName, "w" )
        dumpCounter( invalidCounter, invalidFileName, "w" )
        dumpCounter( wontfixCounter, wontfixFileName, "w" )
        dumpCounter( duplicateCounter, duplicateFileName, "w" )
        dumpCounter( worksformeCounter, worksformeFileName, "w" )
        dumpCounter( movedCounter, movedFileName, "w" )
        dumpCounter( otherResCounter, otherResFileName, "w" )

        #
        # Make a list of all people that contributed to Mozilla this
verison
        #
        contributors = commentCounter.keys()
        for key in reporterCounter.keys():
            if key not in contributors:
                contributors.append( key )

        #
```

```
        # Build a list of 4-tuples in the format:
        # ( contributor, #comments, #totalreports, #dupreports )
        # using the information in the contributors list and the
        # generated counters and write them to a file
        #
        contributorInfo = []
        for contributor in contributors:
            contributorInfo.append( ( contributor,
\
                                      commentCounter.get( contributor, "0" )
,     \
                                      reporterCounter.get( contributor,
"0" ),    \
                                      duplicateCounter.get( contributor, "0"
) ) )

        contributorInfoFile = open( str( version ) +
"-contributorInfo", "w" )
        for cbtor, numcmts, numrpts, numdups in contributorInfo:
            try:
              contributorInfoFile.write( cbtor + "," + str( numcmts )
\
                                                 + "," + str( numrpts )
\
                                                 + "," + str( numdups ) +
"\n" )
            except TypeError:
              pass
        contributorInfoFile.close()

        """
        Clean up for the next pass
        """
        reporterCounter.clear()
        commentCounter.clear()
        fixedCounter.clear()
        invalidCounter.clear()
        wontfixCounter.clear()
        duplicateCounter.clear()
        worksformeCounter.clear()
        movedCounter.clear()
        otherResCounter.clear()
        startDate = endDate
```

analyzeFreq.py

AnalyzeFreq.py is a script used for interpreting the output of doAnalysis.py.

```
def stripBogusContributors( contributors ):
    """
    Iterates through the contributors list and removes any contributors
that
    have no value for the second, third, or fourth items.  This can
```

```
happen
    for certain bogus character strings that the regex in the data
extraction
    script incorrectly determines to be contributors.
    """
    bogusContribs = []
    for i in range( len( contributors ) ):
        contributor = contributors[ i ]
        try:
            int( contributor[ 1 ] )
            int( contributor[ 2 ] )
            int( contributor[ 3 ] )
        except ( ValueError, IndexError ):
            bogusContribs.append( contributor )
    validContribs = [ x for x in contributors if x not in bogusContribs
]
    return bogusContribs, validContribs


def getContribInfoFiles( dirName ):
    """
    Generator for returning each *-contribInfo file in the dirName
    sequentially.
    """
    i = 1
    try:
        while 1:
            fileName = dirName + str( i ) + "-contributorInfo"
            infile = open( fileName, "r" )
            yield infile
            infile.close()
            i += 1
    except IOError:
        """
        In this case, we've hit a filename that doesn't exist. Falling
through
        this method will cause the iteration to terminate.
        """
        pass

def commentsCmp( x, y ):
    """
    The comparator for sorting the list of contributor info lines.
Sorts
    in order of the second element of the list that represents the
contributor
    info.
    """
    return int( x[ 1 ] ) - int( y[ 1 ] )

if __name__ == "__main__":
    dirName = "C:\\My_Data\\Thesis\\scripts\data\\"
    for contribInfoFile in getContribInfoFiles( dirName ):
        print contribInfoFile.name
        """
        For each contribInfoFile, read it line-by-line into a list and
then
```

```
        sort that list by its second element (the number of comments).
        """
        contributors = []
        for line in contribInfoFile:
            contributors.append( line.strip().split( ',' ) )
        bogusContribs, contributors = stripBogusContributors(
contributors )
        contributors.sort( commentsCmp )
        contributors.reverse()

        """
        Generate statistics.  To do this, first get the total number of
        contributors, then set the thresholds for data collection.This
is done
        by finding the particular range of contributors that fall into
each
        of the chosen bins.
        """
        numContributors = len( contributors )
        bins = [ 0.01, 0.02, 0.05, 0.10, 0.20, 0.50 ]
        top = []
        bottom = []
        for bin in bins:
            binSize = int( numContributors * bin )
            top.append( binSize )
            bottom.append( numContributors - binSize )

        numBins = len( bins )
        numCmtsTot = 0
        numCmtsTop = [ 0 ] * numBins
        numCmtsBot = [ 0 ] * numBins
        numBugsTot = 0
        numBugsTop = [ 0 ] * numBins
        numBugsBot = [ 0 ] * numBins
        numDupsTot = 0
        numDupsTop = [ 0 ] * numBins
        numDupsBot = [ 0 ] * numBins

        for i in range( len( contributors ) ):
            """
            For each contributor, first increment the comment,
            bug report, and duplicate totals for this version.
            """
            contributor = contributors[ i ]
            cmts = int( contributor[ 1 ] )
            bugs = int( contributor[ 2 ] )
            dups = int( contributor[ 3 ] )
            numCmtsTot += cmts
            numBugsTot += bugs
            numDupsTot += dups
            for j in range( len( top ) ):
                """
                For each top bin, get the threshold from this bin and
see
                if this this contributor number (in the /i/ counter) is
less
                than the threshold.  If it is, then it belongs in this
```

```
bin and
                we accumulate the values.
                """
                threshhold = top[ j ]
                if i <= threshhold:
                    numCmtsTop[ j ] += cmts
                    numBugsTop[ j ] += bugs
                    numDupsTop[ j ] += dups
            for j in range( len( bottom ) ):
                """
                Same thing for the bottom, except if it's greater is
when
                we need to accumulate the values.
                """
                threshhold = bottom[ j ]
                if i > threshhold:
                    numCmtsBot[ j ] += cmts
                    numBugsBot[ j ] += bugs
                    numDupsBot[ j ] += dups

        """
        Now we need to output the statistics for this version.
        """
        dumpFileName = contribInfoFile.name + "-stats.csv"
        dumpFile = open( dumpFileName, "w" )
        dumpFile.write( "total contributors," + str( numContributors )
+ "\n")
        dumpFile.write( "total comments," + str( numCmtsTot ) + "\n" )
        dumpFile.write( "total bugs," + str( numBugsTot ) + "\n" )
        dumpFile.write( "total duplicates," + str( numDupsTot ) + "\n" )
        dumpFile.write(
"top,n,#comments,#bugs,#dups,%comments,%bugs,%dups\n")
        for i in range( len( bins ) ):
            dumpFile.write( str( bins[ i ] ) + "," +
                            str( top[ i ] ) + "," +
                            str( numCmtsTop[ i ] ) + "," +
                            str( numBugsTop[ i ] ) + "," +
                            str( numDupsTop[ i ] ) + "," +
                            str( numCmtsTop[ i ]/float( numCmtsTot ) )
+ "," +
                            str( numBugsTop[ i ]/float( numBugsTot ) )
+ "," +
                            str( numDupsTop[ i ]/float( numDupsTot ) )
+ "\n")
        dumpFile.write( "\n" )
        dumpFile.write(
"bot,n,#comments,#bugs,#dups,%comments,%bugs,%dups\n")
        for i in range( len( bins ) ):
            dumpFile.write( str( bins[ i ] ) + "," +
                            str( top[ i ] ) + "," +
                            str( numCmtsBot[ i ] ) + "," +
                            str( numBugsBot[ i ] ) + "," +
                            str( numDupsBot[ i ] ) + "," +
                            str( numCmtsBot[ i ]/float( numCmtsTot ) )
+ "," +
                            str( numBugsBot[ i ]/float( numBugsTot ) )
+ "," +
```

```
                            str( numDupsBot[ i ]/float( numDupsTot ) )
+"\n")
        dumpFile.write( "\n" )
        for bogusContrib in bogusContribs:
            dumpFile.write( str( bogusContrib ) + "\n" )
        dumpFile.close()

        """
        Put the figures into a list that we will tap at the end of the
process
        to create a 'master' spreadsheet.
        """
        l = [ numContributors ]
        l.extend( [ numCmtsTot, numBugsTot, numDupsTot ] )
        l.extend( numCmtsTop )
        l.extend( [ i / float( numCmtsTot ) for i in numCmtsTop ] )
        l.extend( numBugsTop )
        l.extend( [ i / float( numBugsTot ) for i in numBugsTop ] )
        l.extend( numDupsTop )
        l.extend( [ i / float( numDupsTot ) for i in numDupsTop ] )
        l.extend( numCmtsBot )
        l.extend( [ i / float( numCmtsTot ) for i in numCmtsBot ] )
        l.extend( numBugsBot )
        l.extend( [ i / float( numBugsTot ) for i in numBugsBot ] )
        l.extend( numDupsBot )
        l.extend( [ i / float( numDupsTot ) for i in numDupsBot ] )

        try:
            masterList.append( l )
        except NameError:
            masterList = []
            masterList.append( l )

    """
    Here, it's time to dump the master sheet.
    """
    print "Writing Master"
    masterFileName = dirName + "master.csv"
    masterFile = open( masterFileName, "w" )
    masterFile.write( "Version,#Contribs,#CmtsTot,#BugsTot,#DupsTot" )
    for bin in bins:
        masterFile.write( ",#CmtsTop" + str( bin ) )
    for bin in bins:
        masterFile.write( ",%CmtsTop" + str( bin ) )
    for bin in bins:
        masterFile.write( ",#BugsTop" + str( bin ) )
    for bin in bins:
        masterFile.write( ",%BugsTop" + str( bin ) )
    for bin in bins:
        masterFile.write( ",#DupsTop" + str( bin ) )
    for bin in bins:
        masterFile.write( ",%DupsTop" + str( bin ) )
    for bin in bins:
        masterFile.write( ",#CmtsBot" + str( bin ) )
    for bin in bins:
        masterFile.write( ",%CmtsBot" + str( bin ) )
    for bin in bins:
```

```
        masterFile.write( ",#BugsBot" + str( bin ) )
    for bin in bins:
        masterFile.write( ",%BugsBot" + str( bin ) )
    for bin in bins:
        masterFile.write( ",#DupsBot" + str( bin ) )
    for bin in bins:
        masterFile.write( ",%DupsBot" + str( bin ) )
    masterFile.write( "\n" )
    for i in range( len( masterList ) ):
        masterListLine = masterList[ i ]
        masterFile.write( str( i + 1 ) )
        for item in masterListLine:
            masterFile.write( "," + str( item ) )
        masterFile.write( "\n" )
```

Counter.py

Counter.py is a general-purpose class used to keep track of a histogram-style mapping.

```
class Counter( dict ):
      def __init__( self, words = None, delim = None ):
            try:
                  for word in words.split( delim ):
                        self += word
            except( TypeError, AttributeError ):
                  pass  # no arg passed, or arg is not seq
      def __iadd__( self, key ):
            """
            Add an item to the freq count and increment the total
counted
            """
            try:
                  self.__dict__[ key ] += 1
            except KeyError:
                  self.__dict__[ key ] = 1
            return self
      def __str__( self ):
            return str( self.__dict__ )
      def keys( self ):
            return self.__dict__.keys()
      def get( self, key, default ):
            return self.__dict__.get( key, default )
      def add( self, key ):
            """
            Adds an item to the frequency count.
            """
            self += key
      def merge( self, c ):
            """
            Merges two Counters.  The count of this counter will
            be changed to equal the sum of the counts of the two
```

```
      Counters together.
      """
      for key in c.__dict__:
            try:
                  self.__dict__[ key ] += c.__dict__[ key ]
            except KeyError:
                  self.__dict__[ key ] = c.__dict__[ key ]
def counts( self, reverse = None ):
      """
      Returns list of keys, sorted by values.
      Feed a 1 if you want a descending sort.
      """
      l = [ ( val, key ) for key, val in self.__dict__.items() ]
      l.sort()
      if reverse:
            l.reverse()
      l = [ ( key, val ) for val, key in l ]
      return l
def clear( self ):
      self.__dict__.clear()
```

BIBLIOGRAPHY

BECK, K.. 2000. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Boston.

BROOKS, F. P.. 1995. *The Mythical Man-Month*. Addison-Wesley Professional, Boston.

FREE SOFTWARE FOUNDATION. 2004. The Free Software Definition. http://www.gnu.org/philosophy/free-sw.html.

FREE SOFTWARE FOUNDATION. 2005. Overview of the GNU System. http://www.gnu.org/gnu/gnu-history.html.

GERMAN, D. AND MOCKUS, A.. 2003. Automating the Measurement of Open Source projects. In *Proceedings of the 3rd Workshop on Open Source Software Engineering*, Portland, Oregon, May 2003, pp.63-67.

JACOBSON, I., BOOCH, G. AND RUMBAUGH, J.. 1998. *The Unified Software Development Process*. Addison-Wesley, Boston.

MOCKUS, A., FIELDING, R. T. AND HERBSLEB, J. D.. 2002. Two Case Studies of Open Source Software Development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, *11*, 3, pp.309-346.

MOZILLA FOUNDATION. 2005a. Hacking Mozilla. http://www.mozilla.org/hacking/life-cycle.html.

MOZILLA FOUNDATION. 2005b. Bonsai - CVS Query Form. http://bonsai.mozilla.org/cvsqueryform.cgi.

MOZILLA FOUNDATION. 2005c. mozilla.org Bugzilla. https://bugzilla.mozilla.org/.

NAKAKOJI, K., YAMAMOTO, Y., NISHINAKA, Y., KISHIDA, K. AND YE, Y.. 2002. Evolution Patterns of Open-Source Software Systems and Communities. In *Proceedings of the International Workshop on Principles of Software Evolution*, Orlando, FL, 2002, ACM Press, Orlando, FL, pp.76-85.

OPEN SOURCE INITIATIVE. 2005. The Open Source Definition. http://www.opensource.org/docs/definition.php.

PYTHON SOFTWARE FOUNDATION. 2005. What is Python?. http://python.org/doc/Summary.html.

RAYMOND, E. S.. 2001. *The Cathedral and the Bazaar*. O'Reilly Media, Inc.,

Cambridge, MA.

REIS, C., PONTIN, R. AND FORTES, M.. 2002. An Overview of the Software Engineering Process and Tools in the Mozilla Project. In *Proceedings of the Open Source Software Development Workshop, Newcastle upon Tyne*, February 2002, pp.155-175.

STALLMAN, R.. 2001. About the GNU Project. http://www.gnu.org/gnu/thegnuproject.html.

YE, Y. AND KISHIDA, K.. 2003. Toward an Understanding of the Motivation of Open Source Software Developers. In *Proceedings of the 25th International Conference on Software Engineering*, Portland, OR, 2003, pp.419-429.