Abstract:
Red Hat Software's flagship product is Red Hat Linux, a Linux based operating system. The vast majority of the components that make up Red Hat Linux are developed and maintained by a world-wide cooperative development effort, connected by the Internet. The people that make up that effort are known as the free software community.

The problem at hand is to modify existing quality assurance tools and methods to increase the reliability of the software that makes up Red Hat Linux. The free software community has testing procedures in place, but none are organized into what most people would consider rigorous practices.

Many of the techniques originally explored in the project are abandoned as they don't scale to the complexity of the model under which this test plan operated. Along they way, several new and unique methods were discovered and integrated into the Red Hat QA program. Much of the work contained within this thesis should not be considered complete, but rather under continuous development. The program itself has taken on a similarity to the cooperative development model. Based on the research exercised for this thesis, the following three primary solutions have been developed and implemented:
1. Derivations of traditional, statistically based tools to measure and monitor the defects present within the software packaged by Red Hat.

2. Development of methods to harness the body of people connected by the Internet and use those resources to raise the level of testing that Red Hat's software receives.

3. Increase the awareness of Linux to involve more people and companies in the free software movement.

A final addition to the thesis is the inclusion of the Red Hat test plan. Unfortunately, that plan was written while the early stages of work were underway and has become out of date as this thesis is completed. The resources and time have not been available to update it since the conclusion of the research time frame. It is included as a work in progress and as an example for other people doing software development tied to free software.

In summary, the results gained through this thesis effort are not attributable to any single change in Red Hat's operating procedures; however, there is no doubt in the author's mind that the aggressive quality assurance practices have led to Red Hat's ability to stay on top of the Linux market and to begin to compete directly with other big name operating systems.

QUALITY ASSURANCE IN A FREE SOFTWARE ENVIRONMENT

by

Michael Ray Wangsmo

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Industrial and Management Engineering

MONTANA STATE UNIVERSITY-BOZEMAN
Bozeman, Montana

November 1998

N378
W18655

APPROVAL

of a thesis submitted by

Michael Ray Wangsmo

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

Dr. Paul Schillings Paul L. Schillings Date 2Dec '98

Approved for the Department of Mechanical and Industrial Engineering

Dr. Victor Cundy [illegible] Date 12/2/98

Approved for the College of Graduate Studies

Dr. Joseph J. Fedock Joseph J. Fedock Date 12/3/98

STATEMENT OF PERMISSION TO USE

Signature [illegible]

Date 30NOV98

ACKNOWLEDGEMENTS

First and foremost, I have to thank Dr. Paul Schillings for keeping me working on this project and for sticking around even when he became sick. I would like to also thank the other two professors on my committee, Dr. John Borkowski and Dr. Joe Stanislao for continuing to work with me even though I ended up finishing 2500 miles from where I started.

The next big round of thanks go to Red Hat Software and Marc Ewing specifically. He hired me to do this work and gave me the support and freedom to do the job right. He also was the person that allowed Red Hat to financially support the conclusion of my graduate program. Even though it took a lot longer this way, I believe that both Red Hat and I have a great future together. I need to thank Donnie Barnes for the initial test plan work and for leaving me alone on Fridays and Ed Bailey for proofreading and some latex help.

Finally, I need to thank my family for encouraging me to follow this path, even though it took me a long way from Montana.

Contents

List of Tables

List of Figures

ABSTRACT

Red Hat Software's flagship product is Red Hat Linux, a Linux based operating system. The vast majority of the components that make up Red Hat Linux are developed and maintained by a world-wide cooperative development effort, connected by the Internet. The people that make up that effort are known as the *free software community*.

The problem at hand is to modify existing *quality assurance* tools and methods to increase the reliability of the software that makes up Red Hat Linux. The free software community has testing procedures in place, but none are organized into what most people would consider rigorous practices.

Many of the techniques originally explored in the project are abandoned as they don't scale to the complexity of the model under which this test plan operated. Along they way, several new and unique methods were discovered and integrated into the Red Hat QA program. Much of the work contained within this thesis should not be considered complete, but rather under continuous development. The program itself has taken on a similarity to the cooperative development model. Based on the research exercised for this thesis, the following three primary solutions have been developed and implemented:

1. Derivations of traditional, statistically based tools to measure and monitor the defects present within the software packaged by Red Hat.

2. Development of methods to harness the body of people connected by the Internet and use those resources to raise the level of testing that Red Hat's software receives.

3. Increase the awareness of Linux to involve more people and companies in the free software movement.

A final addition to the thesis is the inclusion of the Red Hat test plan. Unfortunately, that plan was written while the early stages of work were underway and has become out of date as this thesis is completed. The resources and time have not been available to update it since the conclusion of the research time frame. It is included as a work in progress and as an example for other people doing software development tied to free software.

In summary, the results gained through this thesis effort are not attributable to any single change in Red Hat's operating procedures; however, there is no doubt in the author's mind that the aggressive quality assurance practices have led to Red Hat's ability to stay on top of the Linux market and to begin to compete directly with other big name operating systems.

Chapter 1

INTRODUCTION

Red Hat Software develops a Linux based operating system (OS) for Intel x86, Sun SPARC, and Digital Alpha based computers. The OS is a derivative of Minux which is a UNIX-like OS for Intel processor based machines. It was first developed in 1991 by a Finnish computer science graduate student named Linus Torvalds. Red Hat formed in 1994 and began bundling the various components necessary for a complete OS from the vast pool of software being freely distributed world wide under various *copy-left*[1] licenses. For more information on the history of Linux in general, see `http://www.linux.org` and for more information on Red Hat Software, see `http://www.redhat.com`.

When a company grows from less than five employees to more than 30 in two years, many fundamental changes take place at the administrative and organi-

[1]The phrase *copy-left* was coined by Richard Stallman, founder of the GNU (Gnu is Not Unix) project. One of the core components of the GNU project is the development of the GNU Public License (GPL), a copy of which is included in Appendix C.

zational levels. As a company grows from 30 to 100 employees changes become more subtle, but no less significant. One of the most important changes a company can make as it grows in this age of highly competitive markets is to implement a Quality Assurance (QA) program. For a QA program to be successful, three fundamental components are required:

1. Support of upper management
2. A well defined test plan (with goals)
3. Qualified personnel to develop and implement the plan

For Red Hat Software, the upper management and qualified personnel were in place; the test plan was not. The focus of the research outlined within the thesis is the development the test plan to achieve Red Hat's business goals. In software development, a test plan is the corner stone to managing the testing procedures. The environment that Red Hat exists presented a unique opportunity to explore new ground in the field of both QA and in a growing new field called *free software*.

Background

To fully understand the ramifications and magnitude of the research project, one must first understand the concept of free software, and how a "for-profit" company can become successful in the field. Although the "free" in "free software" can mean "obtainable for no money", a more important meaning applies to the free availability of the software's source code[2]. Computer professionals started writing programs that accomplished specific tasks, desiring that others use the software and, whenever possible, contribute improvements back to the software pool. The only way for that to happen is for the software author to distribute the source code with all software. Through availability and unencumbered by restrictive license agreements, other programmers can modify and refine the original software. Thus, free software is the basis for the *cooperative development model.*

The tool that makes cooperative development possible is the Internet. Using the Internet, people around the world can effectively work together and develop software as if they were sitting in the same office. Cooperative development becomes much more than just a project that a few people participate in at universities. Most of the computers around the world that handle tasks such as email processing, the world wide web, and many other services run software that has been cooperatively developed and released as free software. The following is a description the Free Software Model [13]:

> "Probably the best feature of Linux and the GNU utilities in general and Red Hat Linux in particular is that it is distributable under the terms of the GNU Public License (GPL). This feature has allowed research institutions, universities, commercial enterprises, and hackers, to develop and use Red Hat Linux and related technologies cooperatively without fear that their work would someday be controlled and restricted by a commercial vendor.

[2]Source code consists of instructions written by a software engineer that is compiled into binary code which is executable by a computer's microprocessor.

> In short, the GPL changes the model of software development and distribution to one much like the model our Legal system and its industry uses. If a lawyer designs an argument that wins his case in front of the supreme court his reward is not only the fees his client pays him but also the additional clients that his achievement attracts to his practice. The "argument" he used becomes available for any other lawyer to use without restriction, and in fact becomes part of our collective legal heritage.
>
> This new model is already a new industry with companies like Cygnus Support (see `http://www.cygnus.com`) becoming multi-million dollar enterprises on the basis of providing support and services for large commercial users of GPL'd and other "freely distributable" software. Red Hat Software, Inc.'s rapid growth is based, similarly, on our development and support of the Red Hat Linux distribution, a product that we distribute worldwide on CD and over the Internet under the terms of the GPL."

In a world of constant security threats, crackers, and corporate espionage, traditional OS producers move too slowly to keep up. Linux's open nature provides the basis for constant security enhancements. Security fixes for newly discovered exploits are fixed and released for the various Linux components virtually instantly. Whereas, traditional OSes can take months to release fixes. For companies that are moving into the world of *e-commerce*[3], the rapid security releases and fixes make Linux an obvious choice.

Red Hat Software took advantage of these growing markets and focused a large portion of their resources into them. By acting as a corporate entity, able to manage and maintain a complete Linux distribution, Red Hat brought an image of reliability and trustworthiness to a sometimes chaotic software development en-

[3]This term was made popular by IBM, but generally denotes companies that do business via the Internet.

vironment. Historically, Linux has been thought of as a college-level learning tool and something that only midnight hackers could easily use. By battling such misconceptions, Red Hat has been able to compete with the likes of SUN, Microsoft, and Digital. Red Hat's leadership recognized that to remain and grow it's market, they needed to move faster and produce a more reliable and higher quality operating system than it's big name competitors. Because Red Hat is a member of the free software community, a method had to be developed to control the quality of software produced by thousands of disjoint developers around the world.

A project was started to develop methods, tools, and metrics to monitor and improve the quality of the software that Red Hat produced and sold. The project was to be deployed in several phases covering four distinct areas of the operating system. In addition, it was necessary to apply a more stringent standard to internally developed software versus packages developed outside of Red Hat.

Historical Problems

Before any research could be undertaken to address Red Hat's current issues, past performance had to be thoroughly understood. As Red Hat had grown, the original staff members had to deal with many issues and their memories of some of those issues was limited. That posed problems trying to document historical failures and difficulties.

The major problem found was a lack of communication between the various departments and individuals working on Red Hat Linux. The next priority problem was identifying bugs and fixing them with the limited resources available while still meeting distribution release schedules. It is the classic case of determining "when to ship" when there are more problems than can be reasonably tracked and fixed. Red Hat had been successful enough to continue growing despite these problems. That growth exacerbated the internal communication problems and raised public expectations for a more polished product to a higher level.

Although Red Hat had a system in place for bug tracking, it was a limited system that didn't receive as much usage as it could have. Standard within software companies, bug tracking systems allow individuals to enter a bug or problem into a common database. An "owner" is assigned to the bug and the status of that entry can be monitored. The Red Hat bug tracking system was used for long-term bugs, not considered extremely serious. Serious bugs were handled as encountered. Observations of the system revealed that its data management features were practically non-existent. For example, summary reports, time usage, and developer performance were statistics that could not be extracted from the system, thereby limiting functionality.

The most reliable method of communication within Red Hat is the use of email. Like the model under which Linux itself is developed, the Red Hat staff live and die by their email. While this provides quick and efficient exchanges of ideas and problems, it has the potential to become a source of information overload. It also provides little in the way of historical documentation of problems via a common, searchable database, and it limits access to information. Some people utilize email better than others, but in general, it is not an ideal method for primary bug tracking.

Chapter 2

RESOURCE REVIEW

Before beginning any major project, one searches for other projects related to the proposed undertaking, which eliminates duplication of effort. When considering Red Hat's project, circumstances were different. Not only did research results have to be placed into operation immediately, other organizations performing similar work were not publicizing their experiences. Also, few companies had been using a development model similar enough to Red Hat's to warrant closer examination. Those constraints led to the assumption that no publicly released work in this field had been done and therefore, Red Hat would require the development of a unique solution.

Traditional Quality Assurance Tools

The project was begun by determining what traditional QA techniques could potentially be applied. The use of more traditional QA techniques forms the basis of the QA plan developed for Red Hat. Trusting tools that have been tested and are commonly accepted ensures that improvements, if possible, will be gained. The following standard tools, in one form or another, were selected for further exploration:

- Factor Analysis
- Quality Function Deployment (QFD)
- Pareto Analysis
- Factorial Designed Experiments
- Trend Analysis
- Defect Analysis
- Customer Survey Analysis

A review of each of the tools and techniques now follows:

Factor analysis is a procedure for examining a situation and determining what specific items are related to and might have an influence on a quantity of interest. The technique is used in almost every form of statistical analysis from sampling to experimental design. Because QA has its roots in the statistical sciences, it too, makes use of factor analysis.

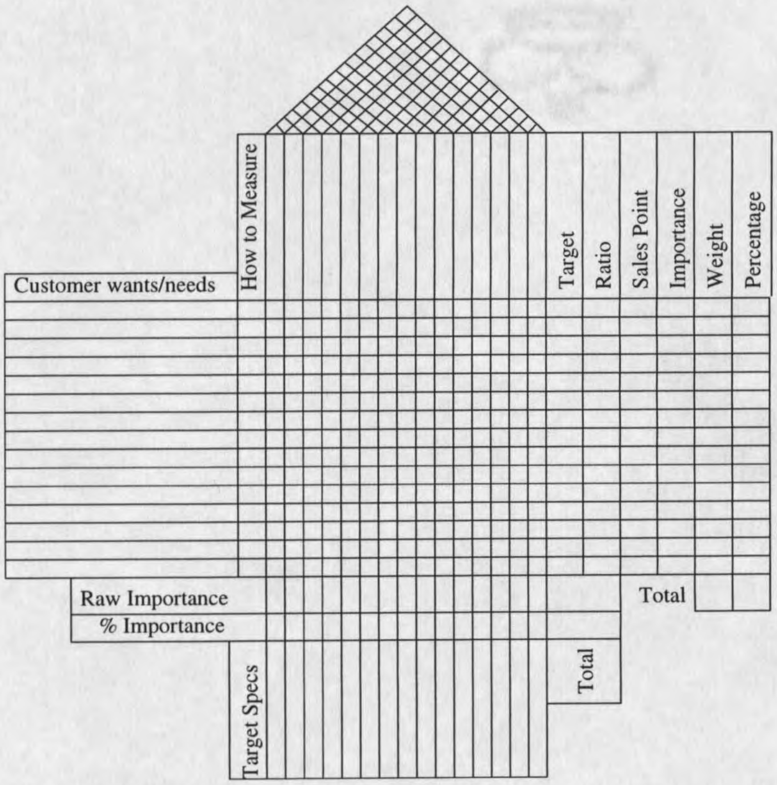

Figure 2.1: Sample QFD chart.

Quality Function Deployment (QFD) is a technique by which the needs of a customer can be matched with the capabilities of a producer. It allows a logical ranking of customer needs and wants that can be reasonably met by the available resources of the production departments. In the world of computer software, the problem is so profound, it can actually drive companies out of business by attempting to meet too many customer requests. The technique uses a grid arrangement that matches a list of customer desires with areas within the company that build customer features into products. Figure 2.1 shows a sample QFD chart. Further analysis of each need and capability pair results in a weighted measure. From the weighted measure, a business can determine how to best allocate resources to each area. As a branch of QFD, **Survey Analysis** is used to gain a *true* understanding of both customer needs and wants and the abilities of the production staff. Carefully written surveys can extract information from customers without inducing biases. Customer survey analysis is not so much a statistical tool as it is an exercise in human psychology. The production of a survey that will yield meaningful customer opinions that are unbiased is a true art form. Surveys should be constructed to lend themselves to valid statistical measurement upon data collection completion.

Pareto Analysis is an ordered bar graph for visualizing areas of concern within a process; Figure 2.2 shows a sample Pareto diagram. The area of analysis can be anything from product defects to customer complaints to product shipping problems. When attempting to solve a problem, one must first know exactly what the significant problem(s) are, or efforts and resources can be wasted. By using such a basic tool, one can quickly and easily identify areas that need to be addressed on a priority basis.

Factorial Designed Experiments use a binary expansion of the number of factors of interest to determine the combinations of factor levels to compare during the course of an experiment. Usually some fraction of the possibilities are used in the interest of simplicity and efficiency. Factorial experiments require that factor levels be discrete. One of the outcomes from an analysis of a factorial design is the ability to make inferences about potential interactions that may exist between fac-

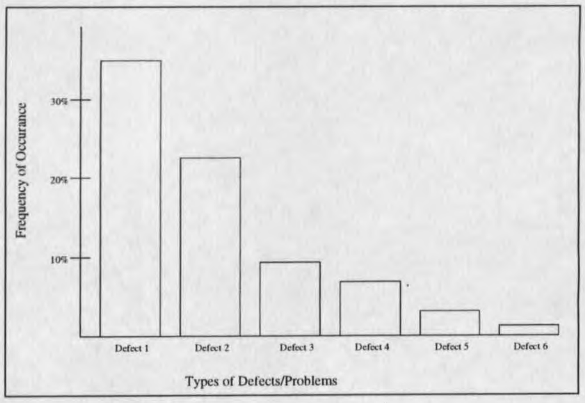

Figure 2.2: Sample Pareto diagram for product defects.

tors. A complete explanation of factorial experiments [11] is well beyond the scope of the research applied to the problem. In the methods derived here, successive fractions of the complete design were selected and examined. The typical factorial experiment design does not apply well to this case, but based on the methods of binary expansion and randomization, a modification of the 2^k design was developed.

Appendix A is a complete copy of the current Red Hat Test Plan [15]. The plan is an on-going model that evolves with Red Hat's growth. Although the plan is not directly tied to the actual content of this thesis, it is a product of the work done for this thesis. Based on the fundamental principles of QA, the test plan is constantly subjected to modifications from each release cycle. The plan is maintained consistent with Juran's *continuous improvement* concepts[9].

Trend Analysis is a statistically based tool generally associated with forecasting and time series analysis, also known as trend modeling [12]. Trend analysis is a method of extracting a trend component, if it exists, from a series of observations. There are generally assumptions placed on the observations, namely that they form a time series; however, it is possible to use the graphical component alone [5]. Through the examination of the patterns present, and using the techniques of *pattern analysis* for forecasting, the optimal timing of checkpoints in the software developmental cycle can be discovered. Trend analysis requires historical data before it can be used as an indicator. No data had ever been collected before, therefore trend analysis could not be performed until future data can be collected.

Defect Analysis is a methodology that measures the severity and prevalence of particular defects within a process. The primary information that is provided by this type of analysis is to pin-point the largest source of problems within the process. Although, extrapolation is required to link the primary defect to it's source, if the method in which the defects are measured is precise, drawing that link is straight forward. Pareto diagrams are used to present the summary information.

Customer Survey Analysis is not so much a statistical tool as it is an exercise in human psychology. The production of a survey that will yield meaningful customer opinions that are unbiased is a true art form. Surveys should be constructed to lend themselves to valid statistical measuring upon data collection completion.

Non-traditional Tools

Linux has been developed in a unique way. Therefore, the primary tools and methods used to develop QA procedures for Red Hat Linux are non-traditional in the field of QA. The most notable non-traditional approach is the use of anonymous, public testing. This radical approach is discussed in greater detail in this section, along with the following other other tools:

- Automated Software Test Suites
- Beta Test Groups
- Public and Private Mailing Lists
- Bug Tracking Systems
- Public Developers web site

It is generally accepted that modifying existing tools for a project is preferable to the creation of new tools. Given that the Red Hat Linux distribution has over 450 individual packages, runs on three hardware platforms, and has a long list of supported hardware, automated testing is considered a priority. However, the use of an **automated software test suite** to test an entire operating system is not a small task.

A large assortment of automated testing suites exists. Most were written for use on Microsoft operating systems. A small number have been written for UNIX platforms, are expensive and, at the time of writing, none have been ported to

Linux. Test automation is critical to alleviate the tedium of repetitive, fundamental "sanity" testing. For non-graphical based applications, traditional scripting languages (shell, tcl/tk, expect, etc.) seemed to be the most reasonable approaches. Automated testing of graphical based applications is complex and tools to perform such testing are not readily available. Several commercial options were examined, but due to the cost of licensing, none were adopted. The decision to construct a system utilizing the two previous methods was made and will eventually be implemented.

A private **beta test group** made up of selected individuals willing to test and submit detailed bug reports has been established. Careful screening of candidates should keep the pool small, manageable, and a good source of reliable bug reports. The group has direct contact with the development and QA staff; their bug reports are recorded and carefully examined. The availability of skilled testers at no cost to Red Hat represents a real world resource that cannot be replicated in a test lab environment. Each major beta release is sent to each group member on CD-ROM and the group receives instructions on the parts of the OS to test. Although the beta test group members work for free and donate resources required for testing, it is a standard practice to reward them with copies of final products. The primary method of communication with Red Hat is through a **private mailing list**.

Public beta mailing lists should be established when public beta releases are made to allow the Internet based public testers a forum to communicate through.

The use of these groups os commonly referred to as *leveraging the 'net community*. These lists can be monitored by Red Hat staff to retrieve the testers feedback.

By using these non-traditional tools, Red Hat can make use of the willing pool of public testers and increase the quality of Red Hat Linux. Finding an effective method of extracting valid bug reports becomes the most difficult problem associated with publicly supplied problem reports. There is no reasonable method of automation to assist with the sorting. Instead, it must be done by hand, although the use of mail filters helps. In the end, however, the QA staff still must read a massive amount of mail to extract the good information. The next hurdle to overcome is determining what to do with the information. Unless action can be taken immediately, it is likely that the information will be lost, never to be seen again. In the short term, writing reports on white boards works, but anything left on a white board for more than a week generally gets lost in the clutter.

Bug tracking systems have been developed for the purpose of categorizing and retaining problem and bug reports. Bug tracking systems are specialized databases that allow information to be stored and searched according to many criteria such as owner, severity of bug, etc. A good bug tracking system allows assignment of ownership to each entry and also maintains a method of reminding owners of their open bugs. The ability to produce summary reports is also a desirable feature enabling tracking of bug reduction progress. A further feature of the ideal bug tracking system is one in which the ability to track and measure defects exists. That functionality was not present in Red Hat's system, but is considered a priority for the future replacement system.

A final tool that is beneficial to promoting better QA and communication is a **web site** dedicated to expressing the current status of beta release software. That beta site needs to include such items as:

- Location of the beta software
- How and where to report bugs and problems
- Change log information explaining what changes have taken place since the last revisions
- Specific areas of testing currently being worked on
- Procedures for submitting patches or bug fixes

The developer web site should be publicly available and regularly updated. The need also exists for a section within the site that is restricted access for the private beta testers. That section should explain things like schedules, special server locations and how to access resources made available to them.

Chapter 3

DEVELOPMENT OF THE PLAN

To begin the project, the author was given a copy of the existing test plan written by Donnie Barnes [4]. A complete revision of that test plan became the foundation for the testing cycle for Red Hat Linux 5.0. Contained within that document is the knowledge gained from the previous two years of development and testing. It gave a great amount of insight to problem areas and provided a sound starting point. Based on that, the author released a complete revision of the document. The current version, which is based on Red Hat Linux 4.2 and modified for Red Hat Linux 5.0, is included in Appendix A [15]. The revised document was written entirely on instinct based on academic experience with testing and a strong familiarity with the Linux operating system.

Situation Assessment

During the time spent writing the new test plan, an overall assessment of the needs of Red Hat was completed. Within that needs assessment, the potential impact of various testing methods was evaluated. Because the actual deployment of the test plan was placed into service as it was developed on a running business, minimization of potential side effects was a concern. Based on the study of the original test plan and a situational assessment at Red Hat, the following decisions were made:

1. A test lab would be designed and maintained by the author throughout the testing/development cycle for Red Hat Linux 5.0.
2. All design changes to packages would be tested and approved by the author before release to the public.
3. The beta test group would report to the author, who would filter the bug reports, passing them to the development staff.
4. The author would acquire as much diverse hardware as the budget permits while still remaining feasible to test.
5. Package testing would be performed primarily by the author, with minimal help from the development staff.

The above list presented the opportunity to deploy a rigorous testing plan into Red Hat without drastically disturbing normal operations. Once the first development and release cycle had been completed, the whole system would be revised and improved in an effort to address the lessons learned from the first pass through the system. A further problem of this research project was that work needed to begin on QA testing as soon as the assessment of the situation was completed.

Overview of Test Schedule

The testing schedule was developed and implemented in real time as the author learned more about Red Hat Software's expectations and needs, which kept development and testing schedules in a fluid state. Many projects were thusly delayed and in the end, rushed to completion.

Initial research began in March, 1997 when the author was hired by Red Hat for a summer internship. The research performed at that point was limited to speculation regarding limited prior knowledge of procedures and practices at Red Hat. By May 1997, the author was at Red Hat's facility and began initial assessment of the situation to prepare for the upcoming development cycle. At about this time, the main development group had constructed what would become the first of several design goals for Red Hat Linux 5.0. The release date for Red Hat Linux 5.0 was set at December 1, 1997 with November 10th the final product due date for shipment to the production house.

Through the months of June, July, and August, initial development of features for Red Hat Linux 5.0 began. Part way through that time, a major shift in the design goals occurred, and many earlier features were scrapped. Also, during that time frame the author designed and built the test lab that would become a cornerstone of testing at Red Hat Software. Although a limited amount of testing took place during those months, the author began to learn the details of how things worked at Red Hat.

Starting in September, serious testing of the install code began in anticipation of the first beta release, version 4.8 named **Thunderbird**. The beta release was sent out for testing near the middle of September. While release testing proceeded, the author prepared for the next beta release and populated the test lab with the hardware required which would ensure that the next beta would get adequate hardware coverage. Several refreshes and updates of the 4.8 beta were built and released. Near the middle of October, the second beta release, version 4.9 named **Mustang**, was released. Mustang contained a much-improved installation program, as well as a few package updates. Consistent with the 4.8.x sequence, several refreshes of the 4.9 beta occurred. During this time all packages in the distribution were updated.

Based on the assumption that the device driver code would not change significantly between the pre-release kernel and the actual kernel[1] (shipped with Red Hat Linux 5.0), testing began on the hardware and devices. At this time, the Alpha[2] port of the beta was also released, which required that testing take place on two platforms concurrently[3]. Once the hardware testing had been completed, package testing became the primary focus. Package testing was primarily designed to locate problems with how the individual packages were built and that programs contained within those packages executed properly. One of the major changes

[1]The kernel is the core piece of software the makes up an operating system. The kernel is the piece of code that translates user level instructions to machine code that the hardware can understand.

[2]See page 37 for more information on the different the platforms.

[3]Due to problems with libraries and kernels, the SPARC port was not included in testing.

between Red Hat Linux 4.2 and the Red Hat Linux 5.0 was in the C library, see page 47 for a detailed description of this change. The main reason for the change was to move to a POSIX[4][1] compliant C library, called glibc (GNU libc). Because of this library change, the work in testing was essentially equivalent to testing a completely new port as every single package/program had to be rebuilt and individually tested.

During the last 15 days of the developmental cycle, a code freeze was implemented which restricted code changes to those fixing bugs and problems and no new features or packages were allowed. Beta tester feedback was critical at this stage. The feedback from both the private and public testing groups resulted in a constant stream of potential bug reports that needed to be tracked down and fixed. The distribution was packaged and released on time, although all known problems were not fixed due to time schedule constraints.

[4]Portable Operating System Interface

Chapter 4

TESTING METHODS

The testing portion of the development cycle was reduced to four phases:

I: Install/Upgrade Testing
II: Package Testing
III: Hardware Testing
IV: Functionality Testing

The main purpose for splitting the testing into distinct phases was to allow multiple testers to work in parallel on different components of the testing. Although parallel testing procedures were not implemented initially, the methodology was put in place for parallel work as Red Hat expands. There does exist some minimal overlap between the phases, but in general, the phases can be run independent of each other.

Phase I - Install/Upgrade Testing

The most distinctive part of Red Hat Linux is the installation software. Testing and development of the install software is one of the most important areas of testing. The software that installs Red Hat Linux sets Red Hat apart from all other operating systems. The install software is written in house and has unique testing requirements. Therefore, the amount of time and resources spent on testing the install software was equivalent to that spent on the remaining three phases combined.

At the beginning of Phase I, several people met to determine what new features needed to be added to the installation software. Also finalized at this meeting was a list of bugs from the previous version and how they would be addressed. The features that were discussed were ranked based on importance and feasibility of implementation. A valuable tool that can be used to assist with this process is *Quality Function Deployment* (QFD). QFD was not used due to a lack of prerequisite information; however, the framework for it has been put in place for use during the next release development cycle.

Features desired by customers are ranked on the horizontal rows by relative importance. The technical specifications of what the install code needs to do (and can feasibly be implemented) are listed in the vertical columns, along with a feasibility ranking. Within the grid area, a scale of 1-5 is used to match correlations between features and specifications. Then, based on the feasibility and customer weights, new install features can be prioritized.

Install testing

Figure 4.1 shows the install code has 5 distinct paths, one for each installation method. Although some of these paths are not present on the Alpha and SPARC architectures, for the purposes of this discussion, that point doesn't matter. NFS stands for *Network File System*, which is a protocol used to share file systems across a network. FTP stands for *File Transfer Protocol*, which is a method of moving a single file from one machine to another. SMB stands for *Server Message Block*, which is a protocol for sharing files and devices developed by Microsoft.

During the initial stages of development and testing, only the NFS path was used. Updating the install tree[1] on one machine and exporting the tree via NFS is much simpler than re-mastering a CD to test the new install tree. Also, because the FTP method requires a supplemental disk to be loaded at boot time, it's much quicker to use NFS in the test lab. Once most bugs have been worked out of the code, a CD-ROM is created to facilitate testing CD-ROM installs. The CD-ROM is then transfered to a Microsoft Windows based machine. The CD-ROM is configured as a standard Windows "shared" which can be accessed by the installer as an SMB filesystem. Next, FTP installs are tested to make sure that there are no problems down that code path. The install tree being tested is copied to a Jaz[2] cartridge, which is then inserted into a test machine. A local hard drive based

[1]The directory structure that contains the Red Hat installer, supporting files, and all packages is referred to as a *tree*.
[2]A removable 1G hard disk cartridge system manufactured by Iomega.

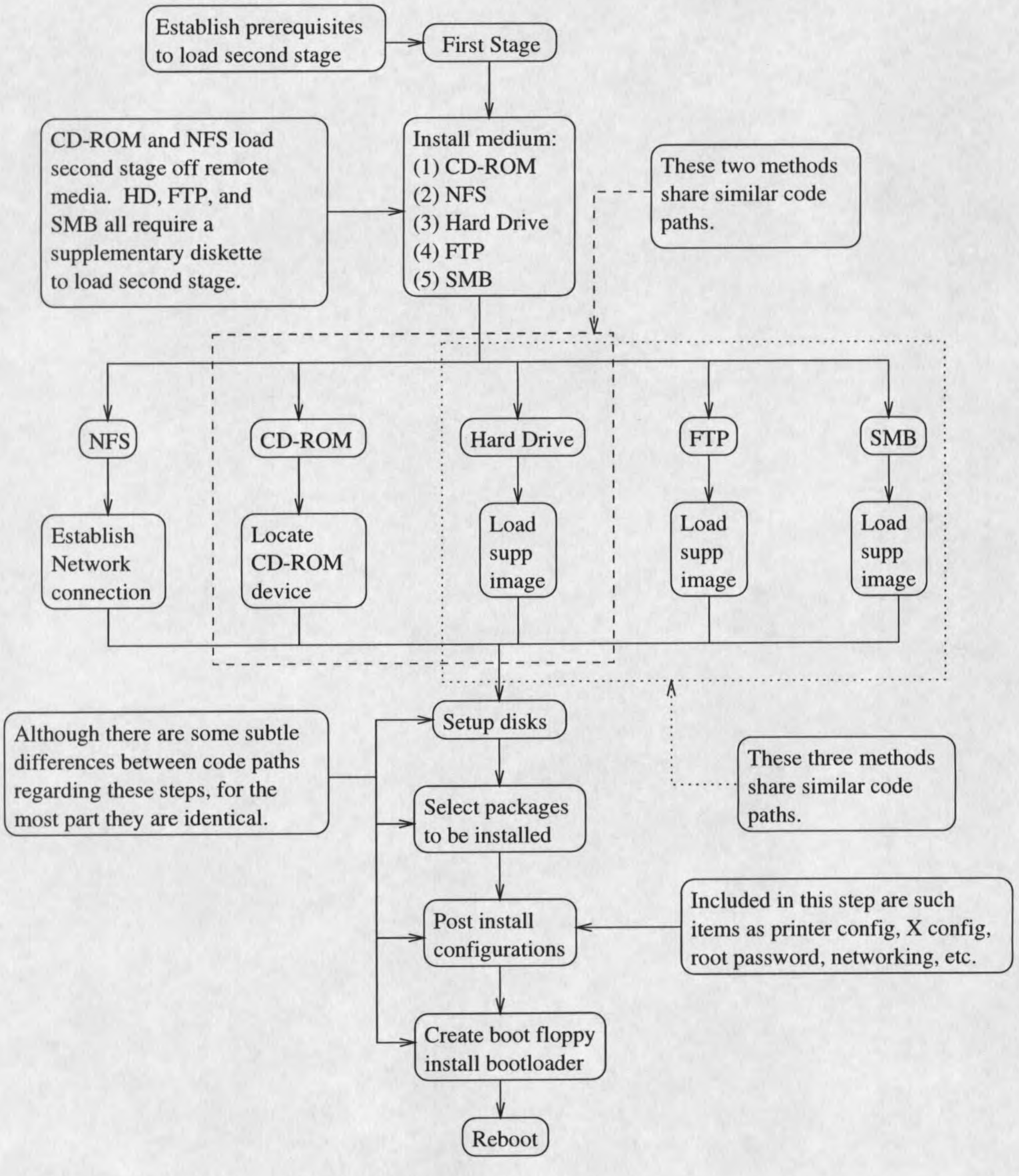
Establish prerequisites to load second stage
First Stage
CD-ROM and NFS load second stage off remote media. HD, FTP, and SMB all require a supplementary diskette to load second stage.
Install medium:
(1) CD-ROM
(2) NFS
(3) Hard Drive
(4) FTP
(5) SMB
These two methods share similar code paths.
NFS
CD-ROM
Hard Drive
FTP
SMB
Establish Network connection
Locate CD-ROM device
Load supp image
Load supp image
Load supp image
Although there are some subtle differences between code paths regarding these steps, for the most part they are identical.
Setup disks
These three methods share similar code paths.
Select packages to be installed
Post install configurations
Included in this step are such items as printer config, X config, root password, networking, etc.
Create boot floppy install bootloader
Reboot

Figure 4.1: Macro flowchart of install code on Intel.

install can then be tested. All problems, bugs, and behavioral inconsistencies are recorded by the tester(s) and passed on to the developmental team. The previous steps are repeated as often as necessary until all code paths in the install can be executed without problems. Check lists are used to ensure that all paths have been examined.

Upgrade testing

At this point, the upgrade paths remain untested. Meticulous care must be taken to ensure that a *reasonably* configured system can be upgraded with little loss of previous system integrity. The question is, what defines *reasonably*?

It is easy in the test lab to install a previous version of Red Hat Linux, upgrade it, then reboot the machine and note what problems occur. However, few people actually run unmodified Red Hat Linux systems. Chances are that the system has been heavily modified from its original state. Creating a "lived in" system in the test lab is expensive in terms of labor and time.

In an effort to address those issues while providing timely testing, a method of creating a "lived in" machine was developed. There were many servers and workstations throughout the Red Hat offices that are based on Red Hat releases of varying vintages. The solution to this problem was to take a snapshot of the root file system of those machines and store the snapshot on a CD. Those CDs can then be used to recreate the original, "lived-in" environment on a test lab machine in a short amount of time. These replicated systems can then be upgraded again without consuming a large amount of resources.

The same methods used for installation testing are used in upgrade testing. Although install and upgrade testing are performed in a repetitive, cyclical fashion, the install paths are usually tested first, allowing a period in which only upgrade paths are tested. When it comes to upgrades, no amount of testing on Red Hat's part can replace a well maintained machine with proper backups. The goal is to make the upgrade work well, and return a bootable system with all original config files preserved.

Most people do not want to extensively reconfigure their systems after an upgrade; however, the cost of making upgrade code that can understand all possible system configurations is infeasible. When a package is initially built, all the files that are considered *configuration* files are tagged with a special flag in the RPM[3] database. When that package is upgraded to a newer version, all configuration files from the original package are moved from their original name to the same name with a .rpmsave extension[4]. Once the upgrade is complete, the user then proceeds through the various .rpmsave files and restores their customized configurations. From a testing point of view, it is critical that the configuration files do, in fact, get saved.

[3]RPM stands for *Red Hat Package Manager* which is the foundation tool used to maintain the software installed on a Red Hat Linux machine.

[4]If the configuration file contained in the upgraded package is identical to the one already on the system, then the file is left alone and not renamed.

Closure of the install/upgrade loop

The final stage of Phase I testing is to verify that all bugs in the bug tracking system have been addressed in the current revision of the install code. The step is an application of Juran's QA methods of continuous improvement [9]. At this point, the development and testing groups are faced with the classical "When to ship?" question. Although the install and upgrade code is only the first part of development and testing for a release, the amount of time spent on it must be limited. Items from the original "wish list" of features that did not get implemented are recorded in the bug tracking system as change requests for the next revision of the install code.

At points through the development of the installation code, beta releases are put together and tested. Those betas are then released to the beta test group and their feedback carefully monitored. If there are enough changes (and enough time) to allow for a public beta release, the release is named and posted[5] for public testing. A public beta release is desirable because it exposes the software to a diverse and scrutinizing audience. The software is put through every imaginable scenario and many problems that do not surface in the test lab are reported.

[5]Posting a release implies moving the entire tree to a location on the public FTP servers and making appropriate announcements regarding its availability.

Phase II - Package Testing

Package testing can be an overwhelming task when one considers the vast number of items to be tested. There are more than 450 packages on three platforms, each containing an average of ten executables, which equates to 15,000 executables to be tested. Ninety-five percent of those packages are not developed or maintained by Red Hat staff, but rather by people in the free software community. Those people do not work on Red Hat's schedules, which also increases the complexity of the test plan. The maintainers of those packages release periodic, stable releases. At a fixed point in time, Red Hat packages the latest, stable releases and uses those packages for the basis of the Red Hat Linux OS.

Red Hat customers are accustomed to getting the latest versions of all packages contained within Red Hat Linux. That fact forces the package updating stage of building a release to be pushed as close to the actual release date as possible, which becomes a *min-max* type problem. The goal is to maximize the amount of time passing between the package freeze date and the product ship date while minimizing the number of packages that have a newer release before Red Hat Linux is shipped. Given the pace at which some packages are updated (at times daily), it is not possible to capture all the latest releases within a single version of Red Hat Linux. It becomes necessary to focus on the larger packages and to make all reasonable attempts to integrate the latest minor and major versions, ignoring the patch level revisions that occur frequently. Another key to making this process work is for Red Hat developers to be in contact with the people that maintain the various major packages and be aware of package release time frames.

Code freezes

Once a ship date has been established for a release, the scheduling begins by working backwards from that date. See Figure 4.2 for a representation of the timeline. A minimum of 1 week of hard code freeze is required. The definition of hard code freeze is "a date such that any changes made between that date and the ship date are considered bugs in the release itself." Although fixes can be made, they occur only for extreme cases. The hard freeze date also translates to the last date where external testing of individual packages has significant value. Up until that point, testing feedback to the developers translates into fixed and better packages.

Approximately 2 weeks before the hard freeze, a soft freeze is implemented. The vast majority of package testing is accomplished between the soft and hard freeze. A constant flow of email between testers and developers takes place. Packages are tested, bugs found and are fixed by developers.

The general freeze date is at least three weeks before the soft freeze. Items

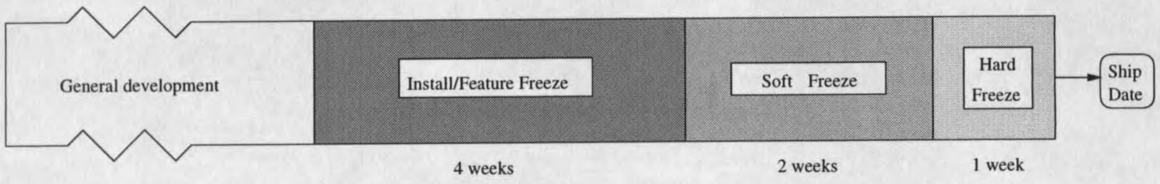

Figure 4.2: General timeline of development milestones.

covered by the feature freeze includes the install code and any major, new features that the release might contain. The period between the general freeze and the soft freeze is also used to test the overall stability of the beta level distribution. During this stage, most developers' individual workstations are upgraded to the beta release and outside testing by public and private beta testers occurs.

Test tracking

A well maintained database that consists of every package contained within the Red Hat distribution is necessary to track the status of the packages both during testing and during defect analysis. Among the many features that a database should have are the tested status of packages, package owners (both development and QA), and many other items related to package dependencies. In lieu of having that package tracking database, text files were maintained, listing all the packages that were in the release, their version numbers, and their tested status. The status was limited to *pass* or *fail*. Once a tool is in place to monitor all the relationships present between people and packages, the package testing phase will be much more manageable.

Testing criteria

Packages are tested based on the following criteria:

- Installability
- Removability
- All executables execute
- Documentation is present
- Minimal functionality

Installability is the ability of the package to be installed without errors. If the package has dependencies[6] which are unsatisfied by the machine onto which it is being installed, then RPM needs to report those dependency problems. File conflicts, previously installed versions, etc. can all lead to installation problems. Most of these problems are solved by installing the prerequisite packages and then proceeding with the tested package's installation.

Removal testing is quite similar to installation testing, only in reverse. A package should not allow itself to be removed if other packages depend upon its installation. Warnings need to be issued, and the user made aware of the consequences of removing that package. RPM also needs to perform any tasks that are listed in the package's post uninstall routines, such as removing its entry in the

[6]Most packages require that other packages be installed before they can function. Built into every package is a list of the packages it requires to install. RPM will report errors if those dependencies are not satisfied when attempting to install the new package.

/etc/rc.d/init.d directory (for packages that contain services started at boot time).

When an executable *core dumps*[7] (or fails catastrophically) it reflects negatively on the software. To test this aspect of a program, execution of the program is attempted. If it executes and performs simple tasks, the test is passed.

All packages shipped in Red Hat Linux need to be accompanied by some form of documentation. In the UNIX world, most executables are documented via *man pages* and *README* files. A man page is a piece of documentation that is displayed via the man command, i.e. man emacs. The man command will handle the formatting of the information to provide quick and simple text based documentation online. READMEs are a plain, ASCII text files supplying general information about a program's functionality. They are a little less practical, but still a valuable source of information. When Red Hat makes a RPM package of something for inclusion into the main distribution, occasionally documentation files are missed. This testing makes sure that available documentation is included with the package.

Finally, minimum functionality is a test that may seem somewhat obvious to the casual observer, but in reality, requires a large amount of work to be done correctly. Although similar to executability testing, functionality testing is reserved for more complex packages. For example, the tetex package contains the document

[7]A core dump is the output of the program's memory after it has died abnormally. A core file (the actual memory file produce by the core dump) can then be diagnosed for clues to the problems that caused the program to die.

formating language called *latex*. Simply testing that latex executes is not sufficient testing. Making latex format some `.tex` files and then viewing the output verifies that latex does what it is supposed to do.

Given the diversity of the packages that are included within Red Hat Linux, the tester has to be well versed in the entire spectrum of UNIX applications to perform this stage of testing. Many of the applications within the distribution are complex and difficult to use. Testing of this type requires prior knowledge of those applications that have been known to cause problems in the past[8], as well as those that are widely used by the customer base. Components of the distribution are ranked and testing proceeds based on prioritization of the importance of the various packages. The concept of "when to ship" is most evident during this stage of testing. Without performing a complete audit of all the software in the distribution, certain trade offs have to be made to actually ship the product. Beta testers play a key role in this area of testing because they extensively use a wide spectrum of the tools and packages in the distribution. They are more than willing to inform Red Hat when tools do not work as expected.

Phase III - Hardware Testing

Hardware testing can be the most interesting part of the testing procedures as well as the most frustrating. Very little control over problems with particular pieces

[8]By re-examining previous problems and bugs, this is a form of regression testing.

of hardware exist within Red Hat. The Linux kernel determines which pieces of hardware work and to what degree they work. Although Red Hat has some influence within the kernel development community, the final determination of what device support will be included in the official Linux kernel rests outside of Red Hat. However, Red Hat does have the option of disabling parts of the kernel that cause problems, or adding fixes and additional features to the kernel.

The main reasons for testing hardware are (a) to provide a list of hardware that works with the OS, as well as some relative scale of the level of functionality associated with each tested hardware item, and (b) reduce the cost of supporting the OS by solving problems with particular devices before customers experience them first hand.

For a well-populated test lab containing legacy, current, and yet-to-be-released hardware, Red Hat must contact hardware manufactures directly and encourage them to support the Linux community. Although this process was initially met with resistance, companies started to ship hardware to the Red Hat test lab, which enabled Red Hat to improve the level of support for many devices. That reduced the cost of supporting the OS as well as created a larger potential user base for both the hardware vendor and Red Hat.

Differences between platforms

Red Hat Linux has been ported to and is supported on machines based on the Intel x86, Digital Alpha, and Sun SPARC processors. Aside from the fact that the Intel architecture is by far the most popular choice, there are other significant differences that separate these architectures; most notable, the Alpha processors are 64 bit whereas the Intel and SPARC processors are 32 bit. The *bit measurement* of a processor is the length, in bits, of a single *word* that the processor can handle at one time. The word length is a measure of how much data the processor can move. Obviously moving 64 bits at a time is much more efficient than moving 32 bits. Difficulty arises because software written for a 32 bit processor may not be "64 bit clean"[9]. There can be problems arising from longer word lengths that need to be examined individually when they occur.

As a measure of comparison, the processors in Commodore 64s and Apple IIs were 8 bit processors. With the advent of Intel's 8086 came the 16 bit processors. Although the 32 bit Intel 80386 (and greater) processors have been the standard since the 80386's introduction in 1985, they too are being phased out. The Alpha processor has been around since 1992. SUN now only produces its Ultra SPARC processor which is 64 bit and Intel is gearing up to release it's Merced processor in 1999 which will also be 64 bit.

[9]The software for the different architectures is built from a single source code repository. No distinction is is made for architecture specific sources (in general). Therefore, some software may be written to take advantage of a known 32 bit word length. When that software is compiled on a 64 bit architecture problems with data management can occur.

The next major difference between these architectures is at the motherboard and BIOS/PROM[10] level. The Intel boards are based (mostly) on a bus[11] technology called PCI. The Alpha boards also use that bus standard as well. SPARCs use a more proprietary bus called S-BUS. Because S-BUS is proprietary, there exists limited hardware availability. Although that translates to fewer choices for the hardware purchasing customer, testing on the SPARCs is simplified.

Intel motherboards use a simple BIOS that allows the user to configure the behavior of the system and to configure the core level device information. Alphas motherboards use enhanced types of BIOSes called ARC/AlphaBIOS or SRM consoles. These add a boot loader and other higher level system functions not present in an Intel BIOS. The SPARC uses a PROM which is an interactive, command-line console. Hardware and boot options (as well as networking information) can be configured with the PROM.

A final difference between Intel/Alpha and SPARC is the support of serial consoles on SPARC. The serial console is a mode of operation in which all input and output can be directed to the serial port instead of a traditional monitor and keyboard. A serial cable is run from the SPARC's serial port and, via a NULL[12] mo-

[10]The BIOS or PROM (depending on architecture) is the low level instruction interface that configure the hardware physically attached to the motherboard of a computer. That instruction set also is responsible for pointing the processor to the bootable media source such as a floppy disk or a hard drive.

[11]The bus of a computer (most have several) is a data path between devices, memory to processor to I/O. The buses also follow standards such as ISA or PCI which define things like the data speed, data path width, hardware settings, etc.

[12]A NULL modem is a serial cable adapter that allows to serial devices to talk to each other directly.

dem, connected to another machine's serial port. Normal console activity of the SPARC is possible by running a terminal emulation program on the second machine. Because Red Hat Linux supports running the install program though a serial console, this mode of operation must be integrated into the test plan on SPARCs.

Kernel Issues

Because the level of hardware support and compatibility are completely determined by the kernel, a short discussion of the Linux kernel follows. The kernel is the code that becomes the interface between the hardware and the software running on the computer. All OS's are based on a kernel, although some OS's do not show off their kernel quite as much as Linux does. The Linux kernel is a cooperatively developed effort by people throughout the world. Linus Torvalds, the creator of the Linux kernel, maintains the official source trees and has the final say as to what code will be integrated. The Linux kernel has a unique method of development. Two trees are maintained simultaneously, a *stable* tree[13] and a *developmental* tree. The kernel versions consist of a three position number a.b.c where a is the major kernel version/family (currently 2). Position b is the minor version where an even number indicates a stable branch and an odd number indicates a developmental branch. These two numbers work in pairs (i.e. 0-1, 2-3, etc.). For example, when the .0 branch is released and stabilized, work begins on develop-

[13]The usage of the word *tree* in this context refers to the source code directory structure of the kernel.

ment in the .1 branch. Position c indicates the kernel's patch level. For the stable branch, the patch level usually does not change more than once a month. In the developmental branch, it can change daily. As a measure of how much source code actually makes up the Linux kernel, there are over one million lines of source code taking up over 50 megabytes of disk space.

The stable kernel branch is updated to add security patches or to fix serious bugs. As kernel updates are released, some device drivers are updated. To allow new or modified code to go into a stable branch kernel tree, the code is tested by a comprehensive process of public testing. The goal of that testing is to ensure that the new code doesn't break anything else within the kernel, because people depend on reliable stable kernels.

There are several reasons Red Hat hired some of the main kernel developers, but the most important is that these people are then funded to continue their work, ensuring that kernel development continues. A secondary reason is that Red Hat maintains strong ties to the kernel community.

Hardware Testing Procedure

Determining the best order to test hardware to obtain maximal coverage with minimal effort was a difficult hurdle to overcome conceptually. In practice, it was not that difficult because the Red Hat test lab had limited hardware in inventory. As Red Hat (and the test lab's inventory) grew, the problem became more pronounced. Labor required to test all hardware in the inventory became the bottleneck.

At the onset of the test plan's development, the goal was to create a method to perform testing based on fractional factorials of the complete population of the test lab's hardware inventory. As more of those fractions were tested, the results converged on a complete census rather than a sample of the test lab inventory. The only modification to a completely randomized orientation of the fractions was to weight each hardware item based on its perceived market share. The primary goal of hardware testing was to ensure that Red Hat Linux worked best with the most popular hardware, hardware not as popular received less testing. It makes little sense to expend the same effort testing a 4-year-old obsolete piece of hardware as on hardware that is currently being sold at the rate of 40,000 units per month.

The concepts of traditional factorial analysis were used as the basis for the hardware testing method. In factorial analysis, there are 2^k elements to be tested where k represents the number of variables in the model expressed at two different levels. For hardware testing, the variables of interest are the different categories of hardware, whereas the levels are the different pieces of hardware within that category contained in the test lab inventory. Table 4.1 shows the breakdown for Intel hardware combinations. Table 4.2 shows the actual combinatorics that were established based on the test lab hardware inventory. See Appendix B for a detailed list that corresponds to the combinatorics table.

Class	Levels
Motherboard	6
Processor	5
Memory	2
Disk type	3
SCSI adapter	18
Video card	14
Ethernet card	17
Mouse type	3

Table 4.1: Intel hardware levels for Red Hat Linux 5.0 testing

The following assumptions were made during hardware testing:

- Interactions are equally likely across all components
- All interactions are independent
- The tester is independent of outcome

It should also be noted that the kernel version used for initial Red Hat Linux 5.0 testing was the 2.0.30 kernel with a pre-release patch to the next version, 2.0.31. An assumption was made concerning the testing of device support under a patched kernel. The assumption was that the device drivers in the patched kernel would be the same as the ones in the final release kernel. That assumption turned out to be correct. However, the final release kernel was delayed, and Red Hat ended up shipping the 5.0 release with the patched kernel.

While Red Hat was working on the glibc port, the Red Hat development staff ran into some low level and difficult problems on the SPARC. Timing was unfortunate because most of the kernel developers working on the SPARC port were busy

PCI based motherboards					
ID	mother board	processor	ethernet card	video card	disk controller
1	1	3	2	1	1a
2	1	4	4	5	2b
3	1	5	5	7	2e
4	2	4	6	8	2f
5	3	4	7	9	2h
6	4	4	9	10	1b
7	4	4	12	11	2j
8	4	4	14	12	2k
9	4	4	17	13	2m
10	4	4	12	13	2n
11	4	4	12	13	2o
12	4	4	12	13	2p
13	4	4	12	13	2q
ISA based motherboards					
ID	mother board	processor	ethernet card	video card	disk controller
14	5	1	1	2	1a
15	5	1	2	3	2a
16	5	1	3	4	2c
17	5	1	8	6	2d
18	5	1	10	14	2g
19	5	1	11	3	2i
20	5	1	13	3	2l
21	5	1	15	3	2r
22	6	1	16	3	1a

Table 4.2: Usable combinations of Intel hardware

with both their regular jobs and with work on the 64 bit version of the kernel for the UltraSPARC processors. It was decided to not ship Red Hat Linux 5.0 for the SPARC, which made testing simpler, but did not make happy customers. From a QA standpoint, it was better not to ship the SPARC port in a beta state for 5.0, but instead, allow for more development and release a more stable port when Red Hat Linux 5.1 was finished. Therefore, there will be no further SPARC discussion.

At the time Red Hat Linux 5.0 was being developed, the test lab had a total of three different Alpha based machines. Table 4.3 shows the Alpha machine types that were available for testing.

Alpha Architecture	Bus
PC164	ISA/PCI
UDB Noname	PCI
Cabriolet	ISA/PCI

Table 4.3: Alpha architectures

The Alpha port was not tested as extensively as the Intel port due to a large discrepancy between the volume of unit sales between the two architectures. Subsequent policy changes have erased this practice and treat all fully supported Red Hat Linux ports equally.

Implementation

A candidate kernel package was built based on version 2.0.30 with the pre-patch to version 2.0.31. The kernel package was then integrated into the development build tree, which in turn fed the QA installation trees. The kernel and associated driver modules[14] were extracted from the package and integrated into the Red Hat installation boot floppy image. An important point to be noted here is that the kernel used to boot and install the system is the same kernel that is actually installed on the system during the package installation phase. Future work may cause this to change, but during work for this project, it was a crucial point.

The primary purpose of a single kernel is for the benefit of both Red Hat support and for QA. Support can assume that if the install software was able to recognize all present hardware, the resultant installed machine will as well. For QA, the single kernel reduces the number of variables present within a release.

At times, both the device driver and the install code were broken. Although repetitions showed that there was definitely a problem, there was no simple way to isolate the problem's location. Some of the hardware in the test lab is not well documented, so the author was forced to learn how to make the drivers work with the hardware while at the same time determining whether or not the installation software was at fault. At this point, the author relied heavily on people from the developmental team. With their help, the author was able to determine that some SCSI[15] controllers simply did not work, and that the install code was doing the correct things. The user interface of the install software received simplifications

[14]Red Hat uses a modular kernel that allows parts of the kernel, typically device drivers, to dynamically loaded and unloaded from memory as needed.

[15]Small Computer System Interface

with respect to how it deals with some types of hardware to alleviate the driver configuration problem.

Those difficulties aside, this approach worked well. There was little wasted effort spent testing hardware only, or testing the install software only. It should be noted that at the time the work was being done, the author was the only person involved with QA, so this method was successful. As more people became involved with the testing process, the coverage of both hardware and install software testing deteriorated. That showed that the independence assumption made previously was good. In fact, independence is required when more than one tester is involved. The actual implementation of independent phases slows the progress of QA, but increases the coverage by having more people involved with the process.

Many times, *brute force* was resorted to; the author simply plugged card after card into machines until complete coverage had been obtained. There was a certain degree of overkill involved with these sessions. It ensured that while the theory of the QA procedures was being developed, the overall quality of the product would not suffer. Confidence was high in the new ideas; however, reality dictated that brute force be used as well. The research completed laid the ground work for the future direction of Red Hat's QA programs.

Phase IV - Functionality Testing

To test the functionality of an OS, several things need to happen.

1. Wide coverage of testers using various applications on the OS
2. Interoperatiblity of the applications and packages that are distributed with the OS
3. Continuous, day to day desktop usage
4. Continuous, day to day server usage

Because of the staff limitations within Red Hat, extensive internal coverage was not possible. During functionality testing, the public beta releases and the private beta test group were most valuable. Before the actual functionality testing issues can be addressed, a short background on the subject of C libraries is required.

C Libraries

Figure 4.3 shows a simplified computer system hierarchy. Each layer of the diagram depends upon lower layers working correctly. Each layer is functionally independent of the layers above it. The position of the C library shows how significant it is to the overall functionality of the system from the users' point of view.

The C library is the library of function calls providing the basic interface to system functions and the runtime support defined in ANSI.[6] These functions are segments of code that can be called by higher-level programs to make building

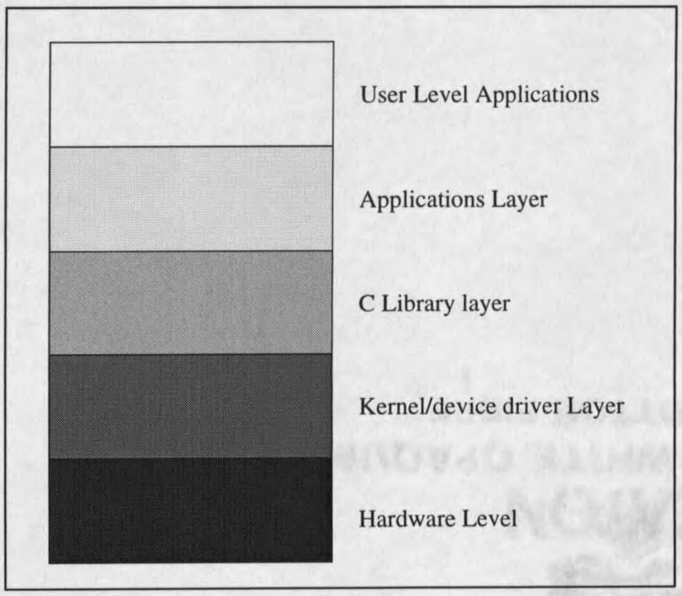

Figure 4.3: Typical Operating System Hierarchy

applications simpler and more portable[16].

The C library is fundamental to the majority of applications and software that make up Red Hat Linux. Therefore, when a core level change in the C library is made, all the software within the distribution has to be rebuilt and re-tested. There are problems associated with the switch from libc5 (version of C library used by Red Hat Linux 4.2) to glibc (GNU libc). The switch was necessary because libc5 was no longer being maintained. In addition, it had many fundamental problems, such as lack of POSIX compliance [1]. Many applications that were written specifically for Linux did not conform to POSIX standards by using hooks contained in libc5. The adoption of glibc was undertaken by Red Hat Software to improve Linux in general. The developers at Red Hat spent almost a year porting all the

[16]Portability in the context of software programming describes the ability of the software's source code to be compiled on other computers and operating systems.

software applications that make up Red Hat Linux to glibc. It was because of this tremendous shift in the core of the OS that functionality testing became so crucial.

Another major benefit driving Red Hat to adopt glibc is glibc's uniformity between architectures. Before Red Hat Linux 5.0, the C libraries on the Intel, Alpha, and SPARC ports were all different and caused discrepancies among builds of the same packages on different architectures. Since the change, virtually all of these types of problems have ceased to exist.

Public beta releases

Because of these major changes to the C library, functionality testing was critical. There were two minor releases of Red Hat Linux under the public beta program. Red Hat Linux 4.8 (aka Thunderbird) and Red Hat Linux 4.9 (aka Mustang). These betas had a life of approximately six weeks each. There were mailing lists set up for each beta and most of the Red Hat development staff monitored mail traffic daily. Unfortunately, a method of defect analysis or tracking was not in place at that time, therefore, the actual significance of these public releases couldn't be measured. However, it was the general consensus of all involved that the feedback received from the public betas was valuable enough to justify the effort administering them.

The largest problem with public beta testing is that people are not on Red Hat's schedule and are not privy to internal information. These factors contribute to the downside of the public betas; people using the betas were not providing feedback on the same time frame that Red Hat needed. During the first beta, Red Hat's primary focus was the install code, not the packages that made up the distribution; however, most of the people using the first beta saw it as a chance to gain a sneak peak at the next Red Hat release. They didn't realize that it was a beta and that only certain portions of the distribution had been modified. A large amount of feedback was given to the Red Hat staff, but because a significant amount of the feedback referred to areas of the distribution that weren't being worked on yet, good information was lost.

The second beta, with it's huge assortment of updated packages, reduced the complaints on the beta list. People then began to submit real, valuable functionality feedback. It was during this phase that the open, public beta really began to show its worth. The only drawback to this beta was the Red Hat time schedule. By the time the second beta was released, it was getting close to final code freeze, which meant that it was increasingly difficult to make changes or update any packages. People on the outside did not know this, nor could they be told. Feedback kept coming in, but internal development had stopped. It was unfortunate, but because of this, Red Hat shipped release 5.0 with known bugs and problems. There were updates available on the Red Hat FTP site at the same time as the release of 5.0, but because of marketing schedules, it was not possible to delay production of the actual product.

A final drawback to the public beta testing is that it virtually stopped the sales of Red Hat Linux 4.2. People become aware that Red Hat had a new release in the works and they held off purchasing the current release with the anticipation of a newer version just around the corner.

Private beta release

At the same time the public betas were being released, CD images with the same releases were being shipped to a private group of beta testers. Those people were placed onto a private beta test group mailing list along with the Red Hat development staff. These people were much more diligent and had their focus much more in tune with Red Hat development. They were also informed of critical marketing and development schedules, which made it easier to communicate with them.

Because of the higher level of technical proficiency and general Red Hat Linux background, the signal to noise ratio[17] of the beta testers list was very good. One of the primary objectives of that list was to maintain group focus at all times and test only areas of the distribution that were being developed by Red Hat staff.

The greatest contribution of the private beta testers fell in to the areas of install code testing and hardware testing. They were focused in their attempts to make the software fail. When software did fail, they had all the necessary technical skills to report the failures intelligently and in a manner almost always reproducible, which improved trouble shooting.

[17]The signal to noise ratio of a mailing list refers to the amount of unfocused debate and meaningless topic threads versus the amount of concise, relative information posted.

Functionality testing was better in the private than the public group, but again, because of the production deadlines it was difficult to integrate all problems found by the group.

Internal Red Hat Usage

All beta releases were installed on internal Red Hat workstations, giving the OS good day to day coverage. Once things began to stabilize, some of the OS components were installed on machines running the Red Hat web sites and FTP servers. Since Red Hat used those machines for day to day operations, it was crucial that changes made to them were as non-disruptive as possible. The Red Hat web site averaged around a million hits[18] per day, and the FTP machines served out about five gigabytes of data per 24 hour period, throughly stress testing some components.

Between employee workstations and the servers, a large cross section of packages in the OS were tested reasonably well. When internal Red Hat machines experience problems with a package, it is easier to gather information about the problem. Normally, external bug reports have to be replicated internally, which lower reliability and create time delays.

The most beneficial segment of users within Red Hat is the support staff. The people who work in support tend to focus on historical problems because of their

[18]A hit is defined as a web browser requesting a web page or part of a web page from a server.

day to day involvement with reported problems. The support staff perform a good job of regression testing, that is, making sure that previously corrected problems have not been reintroduced into the distribution.

Chapter 5

ANALYSIS OF THE QA PROCESS

On May 29th, Red Hat Linux 4.2 was released, thus signaling the start of work on version 5.0. On June 1, Red Hat Linux 5.1 was released which was the end of 5.0's life. The period of time covered by the research on this thesis is between those two dates.

The process of measuring the impact that the tools and procedures developed in this research is every bit as complex as the QA process itself. Part of the problem was to ensure that accurate records were taken during the process of development and testing so that analysis could be performed upon completion of the project. Unfortunately, resource limitations limited the ability to accurately track all defects and problems with the release.

The following metrics were used to measure the success of the project:

1. Number of non-security errata items for Red Hat Linux 5.0
2. Raw counts of bug reports (where ever possible)
3. Feedback from Red Hat staff

The first two items are self explanatory metrics requiring little interpretation. Number three is a measure that provides indications of customer difficulty with the release, which is not numerically measurable. The results of querying people within Red Hat can be used to judge overall impressions of the QA process.

Errata Analysis

The Red Hat errata are updates to a release that are crucial to a machines' functionality. The errata items fall into the following types:

- Security
- Non-security, critical system component
- Non-security, non-critical system component
- Other

Any situation that allows for exploits, denial of service (DoS) attacks, or other breeches of system integrity are classified as *security errata items*. Since the complete source code to the Red Hat Linux OS is freely and publicly available, there are people who search though it looking for exploits. At the same time, other people hear about those exploits and modify the source code to prevent the attack. Many times security errata items are available from Red Hat's FTP server *before* the attack or exploit is openly published.

Since exploits and attack methods are not something that can be tested for during development in a reasonable manner, security errata items are *not* considered defects or blemishes against the overall quality of a Red Hat release.

All other errata items are considered *non-security* and are counted against the quality of the release. Items that can fall under this category include, but are not limited to, packages that do not work, packages that work only partially, and other problems that impede system performance. All of these items are considered defects and reflect negatively on the overall quality and reliability of the release. A primary goal of QA at Red Hat is to reduce the number and severity of the items that fall into this category.

Table 5.1 shows the number of packages that were released as security updates to recent Red Hat Linux releases. It also shows the total number of errata items that

Red Hat release	security	total	percentage
4.0	20	61	33%
4.1	14	53	27%
4.2	43	57	75%
5.0	40	82	49%

Table 5.1: Errata item totals for recent Red Hat Linux releases

were posted as updates to each release. The percentages show what proportion of the updates were security related. The higher the percentage, the better the overall quality of the release. Although 4.1's percentage went down, the jump from 4.0 to 4.2 is significant[1]. The important observation that can be drawn from the table is how much higher the percentage is for 5.0 as compared to 4.0. Both 4.0 and 5.0 were the first release of a major version change for Red Hat Linux. Both had a great many feature and package changes. Given those changes, noting that non-security updates were lower in terms of percentages shows a significant improvement in the testing that took place for Red Hat Linux 5.0.

Table 5.2 shows the approximate length of time that each recent Red Hat Linux release was current. The point to focus on is the correlation between the lengths of time that 4.0 and 5.0 were current and their errata totals. 5.0 lived 50% longer than 4.0 and the yet, the number of total errata items only increased by 33%. When the fact that the ratio of security related errata items to total errata items increased by 42%, it becomes clear that 5.0 was a better tested release.

[1]4.2 was the last release of the 4.x series

Red Hat release	Release Date	Life Length
4.0	Oct 1996	4 months
4.1	Feb 1997	3 months
4.2	May 1997	7 months
5.0	Dec 1997	6 months
5.1	Jun 1998	current

Table 5.2: Lengths of Red Hat Linux releases' lives

Raw Bug Report Counts

In future work, bug report counts will be tracked more accurately. It was one of the areas that resources were not available to fully track during the research time frame. However, the raw information in Table 5.3 is presented.

The contents of the table show the volume of mail received on the beta mailing lists during the stages of beta testing for Red Hat Linux 5.0. The first column is the approximate number of items reported as bugs in the two public betas (Thunderbird 4.8 and Mustang 4.9) and the private beta testers group. The second column is the total volume of mail received on each list during the betas. As can be seen, the total volume of mail increased significantly in the second public beta, which can be attributed to two causes; first, the second beta was closer to final release quality and content and second, news of the Red Hat betas slowly ramped up over time. Interestingly, the ratio of bug reports between the two public betas remained fairly constant. Not surprisingly though, the private beta group had a higher signal to noise ratio.

Although the numbers do not show this information, personal exposure to the

Mailing list	Bug instances	Total messages	Ratio
Thunderbird	48	393	12%
Mustang	152	1036	15%
Beta Testers	145	682	21%

Table 5.3: Raw counts of bug reports

mail messages posted indicated that the level of technical competence and focus was much higher in the private beta group. That comes as no surprise as those people are carefully screened and are compensated (free software products) for their time.

Feedback and Empirical Observations

The material in this section was derived from interviews and discussions with various Red Hat staff members, including Bob Young, CEO of Red Hat. An aggressive QA program has drawbacks, among them are the following:

- Increased development to production time frame
- More bureaucracy is induced into the system
- Increases in tension between various groups within the organization
- Costs for projects that do not directly produce revenue

All of the above costs are negligible when the results of those costs are significantly increased overall revenue. Red Hat's gross revenue more than doubled during the fiscal year during which Red Hat Linux 5.0 was released. Although, there are many other contributing factors to that increase, the overall quality of the OS is one of the primary factors.

By working with individuals from each group involved and with the entire developmental process, procedures were developed to avoid conflicts between groups. The most notable potential conflict is between development and QA. This conflict results when a development team works on a project and then hands it off to QA for testing. For QA, this becomes the first glimpse of the project and usually finds problems that development needs to address. This is an example of *over the wall* development. In order to reduce the level of conflict and tension, the granularity for which projects are developed is increased. By integrating development and QA on projects, the projects are finished more quickly and with less adversarial tensions.

Another area of potential conflict is between marketing and QA. QA performs the final check on the product between the time that development has finished and marketing's production date. To make room for the final round of testing for Red Hat Linux 5.0, marketing's ship date had to slip. Development's schedules cannot be pushed backwards once a start date has occurred. To make room for final QA testing, marketing's schedule was forced to slip forward. After the initial introduction of QA into the process, development's start dates are adjusted to allow for more time at the end of the time line before the production date.

The final costs of the QA program to address are the actual dollar costs associated with the processes. QA will likely never become a profit center for Red Hat software, but the department can perform services that help offset Red Hat's operating costs. One of the programs that is being examined is hardware certification. By expanding the function of the QA department to become a testing center for hardware, Red Hat can charge hardware companies a reasonable fee for certification of their hardware as well as the right to use a Red Hat logo. By offsetting the cost of overall QA, the testing department is able expand further to begin lower level testing of the core components that make up Red Hat Linux.

The benefits of the aggressive QA program developed for Red Hat 5.0 from a marketing point of view are immense. Among the more notable points are the following:

- Independent Software Vendors (ISVs) feel better porting their products to a well tested OS
- Top Red Hat management can point to an organized QA process when other companies are looking at Red Hat for stability and future growth potential
- Customers require a higher level of stability
- An organized QA department provides accountability
- Metrics to measure the stability of Red Hat Linux are possible

Demand for software companies to port their products to Linux is beginning to run high. However, many ISVs are cautious about porting to Linux because of unfamiliarity with the free software model. With an established QA program, ISVs are reassured of the stability and integrity surrounding Red Hat's version of the Linux OS. Red Hat gains from this by having more applications running on Red Hat Linux as well as causing more overall growth in the Linux market. The ISVs benefit from this partnership by increasing the pool of potential customers for their products.

Having well defined QA process in place gives Red Hat credibility to companies that do not understand how the cooperative development model works. Most companies realize that Linux is based on freely available software components from the Internet. Most traditional companies are hesitant to accept the credibility of Linux without a corporate organization backing the OS. People in companies tend to follow the saying of "Nobody was ever fired for going with IBM" which has be updated to read "Nobody was ever fired for going with Microsoft." The established QA procedures in place at Red Hat pushes Red Hat Linux into a traditional, mainstream paradigm.

The final major benefit of Red Hat's QA processes is the ability to measure the stability of new releases. Methods have been developed, as a result of the work done in this project, that allow metrics to measure the quality and stability of the release. By analyzing the number, type and severity of defects of a release, the development staff is able to produce a better product. The QA process is then modified to ensure that more extensive coverage exists so that less defects slip though the QA process.

Defect Analysis

Defect analysis is the examination of a process and its output. By setting a level of acceptability versus unacceptable, a measure of defective output can be developed. Based upon types and quantity of defects present in a process, changes to the process can be implemented to reduce the quantity and severity of the defects. The fundamental principle of total quality management states that the cycle of continuous improvement can only become a closed loop when metrics for the amount of defects and waste in a process are established. By analyzing those measurements, locations in the process causing the most problems can be addressed and corrected.

Based on the work done for Red Hat Linux 5.0, a method of defect analysis was developed. It is a first abstraction that will be implemented in future releases and modified based on its results. The most difficult part of defect analysis in Red Hat is the tracking of the defects. As with most organizations, the last part of a process to be improved is process accounting. In this proposed defect analysis method, the information necessary for the analysis will be tracked in the bug tracking system previously discussed.

Defect analysis at Red Hat has two goals:

1. To provide a means of relative comparison of quality between Red Hat releases
2. Identify areas of testing and development that need improvement

Definitions of terms

- defect: anything that impedes the proper (or correct) operation of a piece of software.

- bug: see defect.
- observation: see defect.
- population: all possible sources of defect reports
- sample: group of observations from a well defined population.
- sample frame: length of time covering sample.
- sample size: number of observations in sample.
- Type I error: concluding that sample I has a higher level of defects than sample II when it doesn't.
- Type II error: concluding that sample I does not have a higher level of defects than sample II when it does.
- class: mutually exclusive categories of observations.
- score: adjusted observation measurement.

Method

Classes need to be defined in order to preserve the scope of the analysis as well as to allow the data to be collected in meaningful components. To that end, the following classes have been established (where R is a scaling factor):

1. install/upgrade ($R = 1.5$)
2. packaging ($R = 1.2$)
3. functionality ($R = .8$)
4. security ($R = .2$)
5. documentation ($R = 1$)

Although there might appear to be some overlap between these classes, they are by definition, mutually exclusive. Care needs to be taken to ensure that observations are placed into the correct class.

Observations can come from the following sources (where w is a weighting factor):

1. bug reporting address and mailing lists ($w = 4$)
2. support mail ($w = 5$)
3. support staff ($w = 4$)
4. development staff ($w = 2$)
5. QA staff ($w = 3$)
6. rumors/other ($w = 1$)

Proper tracking of observation source location is critical to assigning accurate weights to each observation. The Red Hat bug tracking system will be used to maintain this information.

When an observation is recorded, a severity weighting is associated with the observation. The allowed values are $s = 1, 2, 3$ where 1 is low severity and 3 is high severity.

Each bug is assigned a weight based on how long it has been in the bug tracking system as an open bug (the life measurement is in days).

- $t = 1$; life≤ 1
- $t = 2$; $1 <$life≤ 7

- $t = 3; 7 <$life≤ 14
- $t = 4; 14 <$life≤ 30
- $t = 5;$ life> 30

Once an observation has been recorded in the bug tracking system, repeated observations of the same bug are also recorded. Therefore, there exists a counting factor, $c1, ...c6$ that represents the number of occurrences of the same observation within each source group.

Each observation obtains its score by the following method:

$$score_c = \frac{\sum_{n=1}^{6} C_n W_n}{\sum_{n=1}^{6} C_n}.$$

Periodic assessments of the overall score for a class can be obtained by computing the above value for all observations within the sample frame. The weighted arithmetic average of those scores represents the sample's score.

$$sample_{score} = \frac{\sum_{c=1}^{5} score_c R_c}{5}$$

where R_c is the scaling factor for class c.

Analysis

The tracking information provided by this method allows indications of problem areas within the distribution development process and QA procedures. Once identified, problem areas can be corrected and the resultant scores should indicate a downward trend. By using Pareto diagrams, the information calculated can be clearly displayed and significant sources of defects can be easily identified.

Until further research can be performed, error analysis is not possible. The values obtained from these procedures have errors associated with them. Particularly, Type I and II errors. Although both types are bad, it is the Type I errors that can be the most misleading. One method to reduce Type I errors is to have the same people recording observations across all sampling frames. Since that isn't always possible, clearly defined boundaries for all categories and classes must be made and followed when recording observations.

Defect Analysis Wrap-up

The processes examined in this project are not complete. They have begun to look a lot like the free software community which they are attempting to improve. The procedures that were examined and put in place during the Red Hat Linux 5.0 development and release cycle provided a solid foundation for future growth and expansion. Although not every topic investigated was implemented at Red Hat, most were and have proven to be successful. The expectations of the Red Hat staff as well as customers is much higher presently than when the project started.

Chapter 6

SUMMARY

Although measuring quality for Red Hat Linux is a continuous process that can not be quantified in terms of a single starting and stopping point, a relative measurement of the improvements between two points can be examined. The entire goal of the work in this thesis was to make the processes at Red Hat work better. Better processes lead to more efficiency within the organization, which leads to higher productivity, which leads to the company becoming more profitable. Based on how well this implementation works, the company will then be able to justify more resources for QA.

When the work in this thesis started, Red Hat had just finished releasing Red Hat Linux 4.2, and developmental work on Red Hat Linux 5.0 was beginning. The thesis was meant to cover the work that was done to develop procedures for 5.0, but since both the author and Red Hat have not remained static, the actual content of the thesis encompassed more material and time. At the time this summary was written, Red Hat had released Red Hat Linux 5.2 and was beginning to develop Red Hat Linux 6.0. That should give the reader some insight as to the pace that development occurs at Red Hat.

What is gained

Red Hat's sales have more than doubled during the time frame that this thesis covers. Although there are many attributable causes for that growth, a significant amount is due to the concepts and procedures that were developed and implemented by the author. As was shown in the previous chapter, there was a tremendous change in the type of defects, and the level at which Red Hat's customers scrutinize the operating system. Those same people are not only taking notice of Red Hat Software, but are also demanding a higher level of stability, quality and features in Red Hat Linux.

Because of the increased awareness of Red Hat Linux by corporations world wide, the demand for Red Hat products and services has increased dramatically. The major future growth area for Red Hat is supporting the organizations that use the OS. Increased quality and reliability is one of the major contributing factors to this growth.

Who benefits

It is difficult to assess the full spectrum of organizations that benefit by the increased presence of Red Hat Linux. The following categories are some of the major groups that can reap the benefits of Linux's growth:

1. Educational institutions
2. Government installations and organizations
3. Small and medium sized businesses
4. Independent Software Vendors (ISV)
5. The free software community at large

Universities are always the first place to reap the rewards of a better quality operating system at a reduced cost. Learning institutions are able to take advantage of older hardware as well as reduce the amount of money spent on licensing commercial operating systems. Historically, that worked well for certain segments of the university population (such as computer science departments) that were able to fix things that broke because of Linux's immaturity. However, with the recent improvements in quality, other departments within the university are also able to enjoy these benefits.

Government organizations are always under pressure to reduce their operating budgets but at the same time increase their responsiveness to the tax paying public. To increase the responsiveness, faster and better computer systems are integrated into their operations. Until Linux matured to the point it is today, it was not acceptable to use it in mission critical environments. It is now acceptable and common place for government groups to run part or all of their systems on the Linux OS. Not only does it increase the stability of their systems, it greatly reduces the costs associated with true enterprise class operating systems.

Small businesses are always looking for advantages in the market place to grow into larger companies. With the growth of Linux's popularity, more ISVs have ported their applications to Linux. Products such as Wordperfect, Netscape, Matlab, Oracle and many others have been ported to Linux, providing growing companies with a reasonable alternative to expensive commercial operating systems to build their businesses around. ISVs also do well because of Linux's growth by opening up larger markets for their existing customer base as well as being able to expand their product lines into new markets.

The final beneficiary to be mentioned is Red Hat software. By increasing the growth and acceptance of the Linux OS, Red Hat is able to provide more solutions to customers and is able to access more resources to improve on the foundations that make the company successful.

Conclusion of Research

Although many topics that were explored in this research project have not been resolved to a definitive answer, much progress has been made. Red Hat truly is better off for having started down this path. Many of the topics that were explored have led to new areas of exploration for advancements in Red Hat's QA programs. Some of the concepts and topics that were initially explore have been abandoned for one reason or another. That does not mean that the time was wasted, it shows that not every procedure can be applied effectively to every situation.

Much work remains to be done, both in the field of QA applications to free software and at Red Hat specifically. One of the great things about doing this work at Red Hat is the company's willingness to share all discoveries with the public. Just as all software developed by Red Hat is freely released, the work performed to develop QA methods and practices is also. Any tool or software developed in the course of this work is released so that others may benefit from the effort.

As an example of that free release of information, the Red Hat test plan has been included as a part of this thesis. Appendix A is a complete copy of the test plan that was developed during the 4.2 to 5.0 transition period[15]. The plan is an on-going model that evolves with Red Hat's growth. Although the plan is not directly tied to the actual content of this thesis, it is a product of the work done for this thesis. Based on the fundamental principles of QA, the test plan is constantly subjected to modifications from each release cycle. The plan is maintained consistent with Juran's *continuous improvement* concepts[9].

It really makes doing the work feel like an accomplishment. Although Red Hat may not benefit from some topic exploration, another company may use that work as a starting point for them. As more companies begin to understand the free software model, QA methods that work in that environment are going to become more beneficial. It is hoped that many others will take a look at the work performed here and raise it to new heights.

Bibliography

[1] IEEE POSIX Standards Organization, http://standards.ieee.org/regauth/posix/index.ht 1998.

[2] Linux documentation project, 1998. http://www.redhat.com/linux-info/ldp/.

[3] BAILEY, E. *Maximum RPM.* Red Hat Software, Inc., 1997.

[4] BARNES, D. Red hat test plan. version 1.0, 1996.

[5] CLEVELAND, W. S. *The Elements of Graphing Data.* Hobart Press, 1994.

[6] DEITEL, H. *An Introduction to Operating Systems.* Addison-Wesley Publishing Company, 1990.

[7] DEVOR, R. E., CHANG, T., AND SUTHERLAN, J. W. *Statistical Quality Design and Control.* MacMillan Publishing Company, 1992.

[8] GRANT, E. L., AND LEAVENWORTH, R. S. *Statistical Quality Control, Fifth Edition.* McGraw-Hill, Inc., 1980.

[9] JURAN, J. *Juran on Quality by Design.* MacMillan, Inc., 1992.

[10] KOLARIK, W. J. *Creating Quality.* McGraw-Hill, Inc., 1995.

[11] MONTGOMERY, D. C. *Design and Analysis of Experiments, 3rd Ed.* John Wiley and Sons, Inc, 1991.

[12] POLE, A., WEST, M., AND HARRISON, J. *Applied Bayesian Forecasting and Time Series Analysis.* Chapman and Hall, 1994.

[13] RED HAT SOFTWARE, INC. *The Official Red Hat Linux Installation Guide.* Red Hat Software, Inc., 1997.

[14] WANGSMO, M. Ime 577 final project. Final project for IME577, May 1997.

[15] WANGSMO, M., BARNES, D., AND TATERSALL, D. Red hat test plan. version 1.1.1, 1998.

Appendix A

RED HAT TEST PLAN

Author's note: This test plan is considered a work in progress and is incomplete. It is provided as a reference and as an example of some of the work completed as part of this thesis.

Introduction

This test plan is based on Donnie Barnes' original testing plan, but has been extensively modified and should be considered the ideal testing plan. Because of time, money, and hardware constraints, the actual testing plan will likely be a trimmed down version of the test combinations presented in this document. Once a complete installation flowchart has been completed, there will likely be additional modifications and reasons for a reduction in the number of actual installs necessary to achieve reliable confidence.

One of the main goals of this methodical testing plan is to attempt to capture any and all bugs and in the Red Hat installation software. Although that goal may appear to be a very lofty destination, it is not one that is out of reach. The concept of continuous improvement through aggressive testing and methodical analysis is going to be the driving force behind the success of this mission.

Installation Testing

The concept of install testing is very simple, but becomes complex given the wide array of hardware choices available, especially on the i386 platform. Varying degrees of hardware combination resolution will be employed to create a suitable mixture of feasibility and confidence in our product. At the highest level, the testing strategy breaks into the three broad categories of i386, Alpha and SPARC. Testing of installation should be broken across those three platforms independently.

Within each of those three categories, testing will be performed as a complete factorial experiment with each factor representing the peripheral hardware items and the levels representative of the individual devices within each of the factors. One needs to be very careful to *not* make inferences regarding extrapolations from one hardware device to another similar device. This is also to include devices made by the same manufacturer and even with the same "brand/product" name. Hardware manufactures are notorious for changing firmwares and chip sets on the same cards. To account for this, exact bios revision numbers should be recorded as part of the testing documentation.

It can be assumed that each piece of hardware is independent of the other components that are installed in the computer. Because of that assumption, randomization of the testing patterns is not required to return valid results. Also, all combinatorics of hardware need not be considered. All combinations of hard-drive, CD ROM, and controllers should be tested with all the different install methods, this is to ensure that all parts of the install code get tested.

NOTE: The testing scheme presented in this paper is a brute force approach of ensuring that every possible combination of hardware is tested. This level of complexity will become unnecessary once a detailed flow chart of the installation tool is complete. Based on that flow chart, a *possible paths* approach to testing can be undertaken to greatly reduce the complexity of the testing system.

Intel-Hardware

Since the Intel platform is the most widely used and also contains the widest array of potential components, it will take the longest to test and will be the most complex. At the fundamental level, we support only 100% Intel compatible processors in non-SMP configurations. The complete list of supported processors is: 80386, 80486, Pentium, Pentium w/ MMX, PPro, PPro w/ MMX, and the Pentium II. Since the 386's are all but obsolete, they are excluded from the list of test processors. Also, since all the various Pentium chips perform in a similar fashion under Linux, only base Pentiums are necessary to be tested. Bus configurations are tested by way of testing performed on the 486 processors since most are not on PCI based boards. The final fundamental test criteria is the amount of RAM in the system. Tests need to be performed that demonstrate the ability of the RHD to install and function in 8M of RAM. This testing should be applied to each of the different installation mediums. The laptop installation testing is described in a separate section.

Declaration of Test Factors

The following items are the test factors of interest for testing purposes:

- Installation Medium

- Type of Installation *
- RAM *
- HD Controller Type
- CD ROM Type
- Network Card
- Video Card *
- Sound Card *
- Mouse Type *

(*) Indicates a category in which combinatorics of factor levels are not necessary.

Factor Levels

1. Installation Medium
   - CD ROM
   - Hard Drive
   - FTP
   - NFS
   - Floppy
   - PLIP
   - Zip Drive
   - SMB
2. Type of Installation
   - Minimal
   - Non-minimal

   Minimal installation is one in which nothing other than the base packages are installed.
3. RAM
   - 8M
   - >8M

The 8M cases only need to be tested once with each installation medium to confirm the ability of the installation to be performed in low memory conditions.

4. HD Controller Type
    - (E)IDE
    - SCSI **
        - Adaptec AHA-154x
        - Adaptec AHA-294x
        - Advansys
        - BusLogic
        - NCR 53C810
5. CD ROM Type
    - (E)IDE
    - Mitsumi
    - SCSI **
        - Adaptec AHA-154x
        - Adaptec AHA-294x
        - Advansys
        - BusLogic
        - NCR 53C810

    (**) Testing of interactions between SCSI cards is not necessary, therefore, the same SCSI controller will be used for the CD ROM and the hard drive when testing occurs at these factor levels.
6. Networking Card
    - 3c509
    - 3c59x
    - 3x905
    - DE435
    - DE500
    - NE2000
    - SMC EtherPower
    - SMC Ultra

7. Video Card
   - CirusLogic 542x
   - CirusLogic 543x
   - #9 GXE64 Pro
   - Diamond Stealth 2000
   - ArkLogic PV1000
   - S3 Trio64V
   - S3-Virge
   - ATI Mach-32
8. Sound Card
   - AdLib
   - Ensoniq SoundScape
   - Gravis Ultrasound
   - Gravis Ultrasound Max
   - Microsoft Sound System
   - Pro Audio Spectrum 16
   - Sound Blaster Pro
   - Sound Blaster 16
   - Sound Blaster 16 PnP
   - Sound Blaster AWE32
   - Turtle Beach Tropez
9. Mouse
   - No Mouse
   - Microsoft
   - Logitech Mouseman
   - Mouse Systems
   - PS2 Mouse
   - Microsoft Bus Mouse
   - Logitech Bus Mouse
   - ATI Bus Mouse
   - Emulate 3 Buttons

- With 3 Button Emulation
- Without 3 Button Emulation

This list of hardware to be tested is based on the current inventory of hardware we have available. As time progresses, it would be wise to increase this supply of testable hardware. Attempts should be made to assure that we are able to test hardware that will cover at least 80% of the most popular hardware in use. Ways to increase out testable hardware base include correspondence with hardware manufacturers requesting test hardware, possibly soliciting the Red Hat/Linux communities for older hardware to be bought at reasonable prices, and of course an aggressive purchasing plan by Red Hat.

A side note here is the desire to entice hardware manufacturers to submit hardware to be tested and certified as Red Hat compliant. This serves several purposes, namely more current hardware is brought into the test lab. A secondary benefit of a program like this is it encourages hardware vendors to consider Red Hat Linux to be the linux platform to be compliant with. It positions Red Hat as the recognized leader in the Linux community.

Testing Design

NOTE: As stated previously, this is a "brute force" testing approach and will change as soon as the install tool flow chart is complete.

XServer Types - Columns

1. XFree86
2. Metro-X

Video Card Types - Rows

ID	HD Type/Controller	CD ROM type/Controller	Ethernet Card
1	IDE	IDE	3c509
2	(s) AHA-154x	Mitsumi	3c59x
3	(s) AHA-294x	(s) AHA-154x	3c905
4	(s) Advansys	(s) AHA-294x	DE435
5	(s) BusLogic	(s) Advansys	DE500
6	(s) NCR 53C810	(s) BusLogic	NE2000
7		(s) NCR 53C810	SMC EtherPower
8			SMC Ultra

Table A.1: Coded Hardware Levels

	HD Type						
Install Medium	1	2	3	4	5	6	7
Bootable CD	*	*	*	*	*	*	*
HD	*	*	*	*	*	*	*
Zip Drive	*	*	*	*	*	*	*

Table A.2: Simple Install Cases

		HD Type					
		1	2	3	4	5	6
CD ROM Type	1	*	*	*	*	*	*
	2	*	*	*	*	*	*
	3	*	*				
	4	*		*			
	5	*			*		
	6	*				*	
	7	*					*

Table A.3: Floppy Based Install

		Ethernet Card Type							
		1	2	3	4	5	6	7	8
HD Type	1	*	*	*	*	*	*	*	*
	2	*	*	*	*	*	*	*	*
	3	*	*	*	*	*	*	*	*
	4	*	*	*	*	*	*	*	*
	5	*	*	*	*	*	*	*	*
	6	*	*	*	*	*	*	*	*

Table A.4: FTP Based Install.

		Ethernet Card Type							
		1	2	3	4	5	6	7	8
HD Type	1	*	*	*	*	*	*	*	*
	2	*	*	*	*	*	*	*	*
	3	*	*	*	*	*	*	*	*
	4	*	*	*	*	*	*	*	*
	5	*	*	*	*	*	*	*	*
	6	*	*	*	*	*	*	*	*

Table A.5: NFS Based Install

		Ethernet Card Type							
		1	2	3	4	5	6	7	8
HD Type	1	*	*	*	*	*	*	*	*
	2	*	*	*	*	*	*	*	*
	3	*	*	*	*	*	*	*	*
	4	*	*	*	*	*	*	*	*
	5	*	*	*	*	*	*	*	*
	6	*	*	*	*	*	*	*	*

Table A.6: SMB Based Install.

		HD Type					
		1	2	3	4	5	6
Protocol	FTP	*	*	*	*	*	*
	NFS	*	*	*	*	*	*
	SMB	*	*	*	*	*	*

Table A.7: PLIP Device Install

1. CirusLogic 542x
2. CirusLogic 543x
3. #9 GXE64 Pro
4. Diamond Stealth 2000
5. ArkLogic PV1000
6. S3 Trio64V
7. S3-Virge
8. ATI Mach-32

Sound Tests - Columns

1. Driver Installed
2. Application(s) Work

Sound Card Types - Rows

1. AdLib
2. Ensoniq SoundScape
3. Gravis Ultrasound
4. Gravis Ultrasound Max
5. Microsoft Sound System
6. Pro Audio Spectrum 16
7. Sound Blaster Pro
8. Sound Blaster 16
9. Sound Blaster 16 PnP
10. Sound Blaster AWE32
11. Turtle Beach Tropez

Mouse Tests - Columns

1. GPM

	XServer Type	
Video Card	1	2
1	*	*
2	*	*
3	*	*
4	*	*
5	*	*
6	*	*
7	*	*
8	*	*

Table A.8: Video Card/X Install

	Test Type	
Sound Card	1	2
1	*	*
2	*	*
3	*	*
4	*	*
5	*	*
6	*	*
7	*	*
8	*	*

Table A.9: Sound Card Install

2. XFree86
3. MetroX

Mouse Types - Rows

1. No Mouse
2. Microsoft or Compatible
3. Logitech Mouseman
4. Mouse Systems
5. PS/2 Mouse
6. Microsoft Bus Mouse
7. Logitech Bus Mouse
8. ATI Bus Mouse

Miscellaneous Install Testing

This section describes the various configuration options of the install process that need to be tested, but do not need to be tested in conjunction with the various hardware combinations described in the previous section. The primary aim of this section is to verify that when a component group is selected, then the RPMs in that component are correctly installed, and that when a package group, like "emacs .el" is installed it is installed correctly. This testing is based on the bill of materials for the CD set.

1. Component Choices
   (a) Default Install
   (b) Everything Install
   (c) Individual Components
2. Package Groups
   (a) Each Package Group
   (b) All Package Groups

	3 Button			2 Button		
Mouse Type	1	2	3	1	2	3
1		*	*		*	*
2	*	*	*	*	*	*
3	*	*	*	*	*	*
4	*	*	*	*	*	*
5	*	*	*	*	*	*
6	*	*	*	*	*	*
7	*	*	*	*	*	*
8	*	*	*	*	*	*

Table A.10: Mouse Type Install

General Install Testing

This is somewhat a free form testing section, in that we want to cover the unexpected here. There is really no specific pattern or requisite approach to this kind of testing, although, as research into problematic install problems as perceived by the support department progresses, this section may become more structured. Essentially, this section involves trying to install without a hard drive or without root partitions defined; pressing of random keys during the install, keys such as Control-c, Control-z, and other assorted key combinations, noting the behavior of the installation software.

Partitioning

The installation process needs to be tested with a multitude of partitioning schemes, in order to verify that the install software can handle a variety of partition tables. The following partition setups should be tested:

1. Linux only partitions
   - 2 Partitions, one swap and one "/".
   - More than 4 partitions
   - More than 10 partitions
2. Linux only w/ multiple drives
   - 2 (E)IDE drives.
   - 2 SCSI drives.
   - 1 (E)IDE and 1 SCSI drive.

3. Mixture of partition types
    - multiple partitions with at least 1 defined as FATxx

Time-zone/Keyboard/Mouse

Testing in this area involves setting up of several different time zones to ensure that the time gets set correctly. Keyboard testing is to ensure that foreign and Dvorak keyboard layouts get mapped correctly. Mouse testing is a bit more complicated in that we need to test that the install software will correctly set the mouse settings so that both X and gpm work properly. Testing should also be done to see that the 3 button emulation works properly, and that both serial and bus mice work properly. Mouse testing is detailed in Table 10 above.

LILO

Various combinations of testing the installation and operability of LILO need to be performed. The following *ALL* need to be checked:

1. Linux only on file system
    (a) LILO on MBR
    (b) LILO not on MBR, but on bootable partition
    (c) LILO on floppy
    (d) LILO and a kernel image on a floppy
2. DOS\Linux with LILO on MBR
3. win95\Linux with LILO on MBR
4. Red Hat rescue mode of boot floppy

Partitions	SCSI	(E)IDE
1 swap, 1 linux	*	*
1 swap, 5 linux	*	*
1 swap, 11 linux	*	*

Table A.11: Partition test, 1 Hard Drive

Networking

Network testing can be broken down into the the following broad categories:

1. Different levels of networking
    (a) Loop back only: No testing is required here except that the /etc/hosts file has only the loop back address and the local host information.
    (b) Class C IP network: At this level, the /etc/hosts file should contain the proper entry resolving the host name to the host IP address. The networking configuration of the route tables and device assignment should be properly handled at boot time by the rc.d/init.d scripts.
    (c) Class B IP network: Everything as with the class C network setup should work, and the parameter guessing of the appropriate broadcast and net mask information should be functional.
    (d) PPP only configuration: See application testing in the previous section.
2. bootp setups
    (a) Obtain networking information from bootp for install only
    (b) Obtain networking information from bootp for install and keep information for running system
    (c) Obtain networking information from bootp for install and use bootp for running system
    (d) Use static start-up information, but use bootp for running system
3. DNS configurations
    (a) Without DNS: machines on LANs without DNS should function properly. This should be tested with two isolated machines running RH without any DNS configuration.
        i. telnet
        ii. FTP
        iii. rsh, rcp, etc.
        iv. samba
        v. email
            A. standard
            B. POP
        vi. WWW
            A. Arena w/o networking
            B. Arena w/ networking

C. Netscape w/ networking

vii. NFS

A. client

B. server

(b) With DNS: A remote DNS server should be used to test that the local machine can resolve names properly. For further testing of DNS, see the section regarding bind.

i. telnet

ii. FTP

iii. rsh, rcp, etc.

iv. samba

v. email

A. standard

B. POP

vi. WWW

A. Arena w/o networking

B. Arena w/ networking

C. Netscape w/ networking

vii. NFS

A. client

B. server

X Configuration

X configuration should be tested for both XFree86 and for Metro-X. The configuration needs to occur inline with the install software and should only occur if X server packages are selected to be installed. The X configuration software should allow at the very least a minimal X window environment to be established for any supported video card/monitor combination.

During the various install testing combinations, all testable video cards should be swapped into the test machines and the success or failure of producing a working X environment documented.

Sound Testing

This section is incomplete due to the lack of a firm plan for implementation of modular sound support in the RHD. However, the list of sound cards that are supported by our modular sound patch should be tested in much the same fashion as the video cards. Sound cards are independent of the install medium and other components of the computer so they can be tested in any order.

Laptops

Currently, the selection of laptop computers available to the test lab is limited. Based on that, the following test strategy will be applied. We need to test installs via NFS, FTP, SMB and CD ROM using the PCMCIA interfaces. The same testing for packages described within the i386 section applies to laptops so testing can be limited strictly to install testing.

Alpha-Hardware

Testing the installation of the RHD for the Alpha platform should be considered a scaled down version of the same plan presented for the i386 platform. We are limited on the variety of hardware that can be tested, but the UDB will be the base machine for the supported Red Hat Alpha platform. The following is the list of supported and testable hardware:

- NCR810 SCSI
- BusLogic SCSI
- AHA2940 SCSI
- NE2000 ethernet
- DE4x5 ethernet
- S3 based video card
- TGA based video card

Because of the limited range of supported hardware, testing on the Alpha should be very complete and not consume much time.

Alpha(Misc. Install Testing)

All install driven configurations described in the i386 section need to be tested as described in that section with the exception of LILO. Obviously this needs to be replaced with MILO configurations. The testing that should be performed with respect to MILO at Red Hat should be limited to ensuring that the version of MILO that we ship works with the current RHD.

SPARC-Hardware

Due to hardware availability limitations, the testing on the SPARC platform will be limited to testing on the Axil 311 workstations. Installation needs to be tested via bootable CD ROM, floppy CD ROM, NFS, FTP and serial terminal.

SPARC(Misc. Install Testing)

Because of the limited spectrum of hardware in use on the SPARC platform, there is no X window or mouse configuration. Networking configurations should be tested as described within the i386 section.

Package Testing

Two approaches to package testing need to be implemented. Functionality testing for some packages can be a part of an automated package testing tool. The remainder of the more complicated packages and the ones that can not feasibly be tested by an automated tool need to be tested by hand. Only an outline of the operation of the testing tool is available now. The tool should have an interface similar to the install software that will allow the tester to select from a list, the packages that should be tested. The tool should then grab a module, possibly written in perl, for each package to be tested. These modules should perform tasks that determine if the software installed properly at install time and is functioning as would be expected. These modules need to be written to a common standard that logs all output to stdout, also reports any errors to stderr and returns a error code for failed testing of the package. Based on that description, the main tool should be fairly simple in function so that when all packages have been tested, it simply reports the overall success, or failures, of the tested packages. All information output from the modules is logged to a file that can be examined by the tester upon completion.

The package testing needs to be performed on all three platforms given that the packages may behave differently across platforms and there is a potential for mistakes to be made regarding the building of the rpms on different platforms. The following are some specific packages that need to be thoroughly tested by hand. These packages have been notorious sources of problems in the past.

Compilers

The proper operability of the various GNU compilers is essential to to functionality of the RHD. Several different packages can be compiled to put the system compilers through their paces. The first test is to make it successfully through a complete kernel build. Given the amount of C and C++ code that is now in the kernel, it is definitely a good test for the compilers and the system libraries. This test should be done as root. Another test is to compile a package such as ghostview which compiles against the X libraries. This should be done as a user to make sure that all routines and commands are accessible as a user. A final test should be to compile a motif based source tree against the Red Hat motif libraries as both a user and root.

LaTeX

Since LaTeXis a very complex package, there is the potential for a great number of bugs and errors to work their way into the RHD. To test the functionality of the tetex package included with the RHD, the following steps should be taken:

- Latex the Red Hat Users Guide as a user and then as root. This should ensure the proper permission bits have been set correctly.
- The resulting *.dvi file should be converted to postscript via dvips to check to functionality of the kpathesa font decompression tools.
- The resulting *.ps file should be viewed with ghostview to ensure that it has been correctly processed.
- A document from the linuxdoc-sgml should be processed and viewed.
- A postscript file should be printed to both a postscript printer and a non-postscript printer.

PPP

Using the netcfg tool in the Red Hat control-panel, a complete testing of setting up and connection using PPP should be performed. The configuration should be tested as root and users with both dynamic and static IPs. A test dial in user to the Red Hat modem pool that will allow for both the dynamic and static IP setups should be used. Although this is a small section of testing, it is very critical as a large majority of customer complaints are related to configuring PPP.

Email package testing

The five mail programs (elm, pine, mh, exmh, and xmh) need to be installed and tested on systems with DNS configured, without DNS, and with loop back only. Mail also needs to be tested on machines on a LAN not connected to the internet. Mail should be send between two machines running clean RHD installs.

News readers and servers

Tin, trn, slrn, and INN need to be tested to ensure that base configuration works correctly.

UUCP

UUCP needs to be tested based on the included configuration files to ensure that the default configuration does work correctly.

DNS testing

Bind needs to be installed with a simple named.boot file and be used as a test name server for another machine on a LAN.

Apache

The Apache web server should be installed and tests performed that check that the server starts up at boot time on port 80. CGI scripts should be tested as well as virtual domains.

FTP

Installation of wu-ftp should allow immediate user logins and anon-ftp should allow immediate anonymous ftp logins.

Red Hat Applications

All of the Red Hat applications should drop in seamlessly on top of a fresh Red Hat system install. This needs to be tested, but should be a moot point. The applications are tested completely against RHD's before we ship them, so unless something very major is broken, this should not be a problem.

Integration Testing

This section of testing ensures the ability of the RHD to communicate within a non-homogeneous OS environment. From the base configurations, the following testing needs to be performed.

SMB

The smbd and nmbd daemons should be running upon boot after the Samba package is installed. The smbd machine should be "browseable" from other smbclient machines. We need to test it with both Windows 95 and NT. The test should include attaching volumes and copying files both to and from the samba server. Printing should also be tested. Printing should be possible from the SMB clients to a printer attached to a SMB server. Testing needs to be conducted with smbmount and smbclient. The Linux machine should be able to attach to exports from a Windows based machine using these two packages. The main test we need to be concerned with aside from printing is file transfer from SMB server (Linux) to SMB client (Windows based) and from SMB export (Windows based) to smbclient/smbmount (Linux).

Netware

We should be able to mount Netware volumes from a Linux machine and copy files both ways. We should be able to print to a Netware printer. We should also be able to print from a Netware machine to a Linux machine and printer. The Netware machine should also be able to mount Linux volumes.

Upgradability

The ability to upgrade to the current version of the RHD from a prior version or from another distribution is of utmost importance. The upgrade not only needs to be intelligent enough to determine which packages need to be installed and any associated dependencies, but also needs to preserve any prior configuration files. On a Red Hat system version 2.1 or greater, this should be handled mostly by the functionality of RPM. However, each version should be tested in an upgrade situation. The install tool needs to be designed so that the default options will not format or delete anything unnecessarily. The upgrade.log file needs to be clear and direct the user to the locations of every configuration file relocated or renamed by the install tool.

The releases that we will test for upgradeability from are: 2.1, 3.0, 3.0.3, 4.0, 4.1, and 4.2. This represents 6 test cases.

The other area to be tested in the upgrade stage is that of upgrading other linux distributions. The lack of a RPM database makes it is impossible to know what is installed on the system. Based on that fact, the installation tool must cleanly install any and all software need by Red Hat linux. No assumptions can be made about what is or is not already on the system. The exception to this would be any Linux distribution that uses RPM such as Caldera. On those systems, the system's rpm database should be consulted and any installed packages that follow the Red Hat packaging conventions should be detected.

Intel

The i386 version of the RHD needs to be drop in compatible from any previous RHD v2.1 or greater. All configurations need to be preserved and/or used with the upgraded system.

Alpha

The Alpha version of the RHD needs to be upgradeable from v4.0 or greater.

SPARC

The SPARC version of the RHD needs to be upgradeable from v4.0 or greater.

Documentation of Testing

Documentation of the testing procedure needs to be carefully carried out in order to be able to produce repeatable problems and to help the developers correct problems. A further extension of the documentation effort helps lead to ISO900x compliant procedures. By slowly adopting these structured methods, the transition to becoming a globally recognized leader in the desktop operating system community should be smooth.

By properly documenting all testing procedures, it should be easier to replicate bugs and/or problems with the tested configurations. A standard documentation scheme is being developed and will be available before 5.0 beta testing begins. Upon completion of each testing phase, the documentation will be filed in the test lab and made available to anyone needing that information.

A starting point for documentation is to label each test case to be run. Then a database of test cases can be kept with their progress, i.e attempted, finished, wait on fixes, etc. As long as this database is kept up to date, we can know where we are in the test process, how much is left to do, and measure that against the time left to test.

Beta Test Program

A group of beta testers should be assembled and maintained throughout the beta test period. The size of this group should be kept small for organizational and cost reasons. However, it should be large enough to produce some value to Red Hat. The suggested size for this group is approximately 20 people of a technical background. This group of people should be sent beta copies of the RHD and asked to install and report back any problems or concerns they encounter.

At the same time, the true strength of the linux development community should be tapped by making the beta releases available for anonymous ftp download. Although some of the reports that filter in from these channels can be less reliable, the amount of exposure that the beta release gains through these channels far out weighs the down sides. A corresponding mailing list should be established for the discussion of the beta release(s) and should be closely monitored by Red Hat staff. All bug reports need to be explored and treated as potential problems with the distribution.

Intel Test Cases

The number of test cases that must be run to test the installation process and the RPMs that make up the distribution is based on the hardware tables previously defined. These cases are specific to the Intel platforms. SPARC and ALPHA platforms will be considered separately.

Simple Install

Based on Table 2, there are 21 cases, identified by 2-(row)(column). These cases are labelled 3-11 through 2-37. These cases all use more than 8 meg of RAM. In addition, there are three more cases, one for each row of Table 2, which use 8 meg of RAM. These three cases are labelled 2-40, 2-50, and 2-60.

Floppy Based Install

Table 3 contains 7 rows by 6 columns, but is sparsely populated by 22 test cases. The test case identification scheme is 3-(row)(column) e.g. 3-11 or 3-76. These cases all use more than 8 meg of RAM. In addition, one additional case, labelled 3-80, uses 8 meg of RAM.

FTP Based Install

Table 4 contains 6 rows and 8 columns, densely populated, for a total of 48 test cases, labelled 4-(row)(column) e.g. 4-11 or 4-68. These test cases are all run with anonymous FTP, using bootp to assign the internet address. All cases use more than 8 meg of RAM. Two additional cases are 8 meg, 4-70, and more than 8 meg, non-anonymous FTP, 4-80. This makes a total of 50 cases.

NFS Based Install

Table 5 contains 6 rows and 8 columns, densely populated, for a total of 48 test cases, labelled 5-(row)(column) for 5-11 through 5-68. In addition, there is test case 5-70 which uses 8 meg of RAM. This totals 49 test cases.

SMB Based Install

Table 6 contains 6 rows and 8 columns, densely populated, for a total of 48 test cases, labelled 6-(row)(column) for 6-11 through 6-68. These cases all use Windows NT as the SMB server. In addition, there is test case 6-70 which uses 8 meg of RAM. There is one more test case, 6-80 which uses Windows 95 as the SMB server.

PLIP Device Install

Table 7 contains 3 rows by 6 columns, labelling test cases 7-11 through 7-36. The SMB cases use Windows NT as the server, and all use more than 8 meg of RAM. In addition, there are two cases, one using 8 meg of RAM, 7-40, and 7-50 using Windows 95 as the SMB server.

Video Card Install

Table 8 contains 8 rows by 2 columns, labelling testcases 8-11 through 8-26. These represent 16 test cases.

Sound Card Install

Table 9 contains 8 rows by 2 columns, labelling testcases 9-11 through 9-26. These represent 16 test cases.

Mouse Install

Table 01 contains 8 rows by 6 columns, two cases not executes, labelling testcases 10-11 through 10-48. These represent 46 test cases.

Hardware Dependent Cases

The total of these hardware dependent test cases is 294. Since there are not enough machines to set up all these configurations at once, they will need to be scheduled to minimize the set up time between cases. This task remains to be done.

Miscellaneous Install Testing

These tests deal with the components and package groups that get installed. The purpose is to verify that the lists which drive the installation program actually do get installed and produce a working system. Test case 22-00 is a default install. Case 22-01 is an everything install. Cases 22-10 through 22-xx are to verify the installation of each component. Cases 22-100 through 22-5xx are to verify that each individual package is installed. Cases 22-600 and 22-601 verify that the two choices of XServer are properly installed.

Partitioning

These tests deal with various partitioning schemes. There are 6 cases in Table 11, labelled P-11 through P-32. There are 3 cases of multiple drives, cases P-40 through

P-42 for two SCSI, two (E)IDE, and one SCSI one (E)IDE respectively. There is one case for multiple partition types, 1 swap, 4 linux, 1 FAT, case P-50.

Time zone/Keyboard/Mouse

The time zone tests consist of 8 tests, T-00 through T-07, which test the correctness of time zone setting for EST, EDT, CST, PST, GMT, CET, East Coast Australia, and Japanese time. (Note: look up abbreviations for last two). The keyboard tests consist of 4 cases, K-00 through K-03, corresponding to US English, Dvorak, French, and German keyboards.

LILO

The following test cases test the LILO installation. These test are labelled L-00 through L-06.

1. Linux only on file system
   (a) LILO on MBR L-00
   (b) LILO not on MBR, but on bootable partition L-01
   (c) LILO on floppy L-02
   (d) LILO and a kernel image on a floppy L-03
2. DOS\Linux with LILO on MBR L-04
3. win95\Linux with LILO on MBR L-05
4. Red Hat rescue mode of boot floppy L-06

Networking

This testing consists of 30 test cases, N-00 through N-03, N-10 through N-13, N-20 through N-2A, and N-30 through N-3A. N-00 through N-03 correspond to the different levels of networking. N-10 through N-13 correspond to the bootp setups. N-20 through N-2A correspond to the different applications used without DNS, and N-30 through N-3A correspond to the tests with a remote DNS server. The total is 30 test cases.

X Configuration

X configuration testing with different video cards is detailed in Table 8 and the count of test cases is in the hardware dependent section above.

Sound Testing

Sound testing with different sound cards is detailed in Table 9 and the count of test cases is in the hardware dependent section above.

Laptops

The laptop testing needs four test cases; these are via CD ROM, NFS, FTP, and SMB. The cases are L-00 through L-03. The only difference between laptops and workstations is in the PCMCIA attachment of adapters, hence all the replications above are not needed.

Total Test Cases

The total number of test cases for the Intel platform testing is 357 plus the number of component and package test cases which are yet to be determined.

The total number of cases then is on the order of 500, which, at 45 minutes per test case, represents 375 man hours. (There is no justification for the 45 minutes per test case; it is a guess based on hardware set up and tear down times).

ALPHA Test Cases

There are many fewer configurations of ALPHA hardware than of the Intel, therefore there are many fewer test cases to be run. There are three SCSI adapters, and 2 (Is this right?) ethernet adapters and only 2 video adapters.

Thus there will be 6 testcases for each of NFS, FTP and CD ROM installs, and two test cases for the two video cards., which leads to 20 cases. The networking tests are the same as on the Intel platform, so there are an additional 30 test cases. This results in 50 test cases plus the install selection test cases which have not been sized yet.

SPARC Test Cases

There are 5 test cases for testing the SPARC platform, as there is no hardware choice. The test cases are bootable CD ROM, floppy CD ROM, NFS, FTP, and serial terminal. In addition the install selection test cases must be executed.

Appendix B

TEST LAB INVENTORY

The following sections detail the exact inventory of Intel based hardware contained in the test lab during Red Hat Linux 5.0 testing.

Adapter Cards

Most of the following adapter cards are either ISA bus or PCI bus cards. Those types of cards both work on Intel and Alpha machines. SPARC machines have a very small and limited amount of hardware that work on the S-BUS.

Disc controllers

1. non-SCSI
   (a) IDE
   (b) IDE UDMA
2. SCSI
   (a) Adaptec AHA-1542CF
   (b) Adaptec AHA-2930A
   (c) Adaptec AHA-1522 PnP
   (d) Adaptec AHA-1540
   (e) Adaptec AHA-2920
   (f) Adaptec AHA-2940
   (g) Advansys ABP-930/40
   (h) Advansys ABP-940UW
   (i) Buslogic BT-545C

(j) BusLogic BT-548
(k) BusLogic BT-958
(l) Future Domain TMC-885
(m) Future Domain TMC-860
(n) Data Technologies Corp. DTC-3530-A
(o) NCR 53c810
(p) Symbios Logic SYM20401
(q) Tekram NCR 53c875
(r) Trantor T160

Video card

1. ArkLogic PV1000
2. ATI Mach 32A VLB
3. CirrusLogic CL-GD5428
4. CirrusLogic CL-GD54M30
5. CirrusLogic CL-GD5446
6. Diamond Stealth 2000
7. Matrox Mystique
8. Matrox Millenium
9. Matrox Millenium II
10. Number Nine GXE64 Pro
11. Number Nine Imagine 128 II
12. S3 Trio64V
13. S3-Virge
14. Trident 8900D

Ethernet card

1. 3Com Etherlink II/16
2. 3Com 3c509B
3. 3Com 3c589B Etherlink III
4. 3Com 3c595-TX 10/100
5. 3Com 3c950-TX 10/100
6. DEC DE435
7. DEC DE500
8. Intel EtherExpress 16
9. Intel EEPro100
10. Linksys EthernetCard Type II
11. Microdyne NE2000plus3
12. NetGear FA310TX
13. Novell, Inc. NE2000
14. SMC DEC21041
15. SMC EtherEZ 16
16. SMC EtherPower II
17. Winbond PCI 10

Mouse type

1. Busmouse
2. PS/2
3. Serial Microsoft

Intel specific hardware

The following sections detail hardware specific to the Intel architecture.

Intel motherboards

1. Asus P55/TXP4
2. Asus P55-VX97
3. Asus P55-TX97XE
4. Intel P90/NX-Chipset
5. DIGIS VLB 486
6. Generic ISA 486

Intel based processors

1. Intel 486DX2/66
2. Intel 486DX/33
3. Intel P100
4. Intel P133
5. Cyrix 6x86 P166+

Memory categories

1. 8 Megabytes
2. >8 Megabytes

SPARC specific hardware

The following sections detail hardware specific to the SPARC architecture.

Base systems

1. sun4c
2. sun4m
3. IPC

Framebuffers

1. CG3 (8bit)
2. TGX Plus (8bit)

Appendix C

GNU PUBLIC LICENSE

Gnu Public License

GNU GENERAL PUBLIC LICENSE Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0

This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1

You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2

You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3

You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a. Accompany it with the complete corresponding machine readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4

You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5

You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6

Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7

If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8

If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9

The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10

If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

```
and/or modify it under the terms of the GNU General Public
License as published by the Free Software Foundation; either
version 2 of the License, or (at your option) any later
version.

This program is distributed in the hope that it will be
useful, but WITHOUT ANY WARRANTY; without even the implied
warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.  See the GNU General Public License for more
details.

You should have received a copy of the GNU General Public
License along with this program; if not, write to the Free
Software Foundation, Inc., 675 Mass Ave, Cambridge, MA
02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright © 19yy name of
author Gnomovision comes with ABSOLUTELY NO WARRANTY; for
details type 'show w'.  This is free software, and you are
welcome to redistribute it under certain conditions; type
'show c' for details.
```

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest
in the program 'Gnomovision' (which makes passes at compilers)
written by James Hacker.

$<$signature of Ty Coon$>$, 1 April 1989
Ty Coon, President of Vice
```