The Dissertation Committee for Hyrum Kurt Wright
certifies that this is the approved version of the following dissertation:

Release Engineering Processes, Their Faults and Failures

Committee:

Dewayne E. Perry, Supervisor

Randolph Bias

Christine Julien

Miryung Kim

Sarfraz Khurshid

Release Engineering Processes,
Their Faults and Failures

by

Hyrum Kurt Wright, B.S.; M.S.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2012

Dedicated to my loving wife Heather and our children.

Acknowledgments

The work described herein is largely qualitative, and as such required the assistance of a number of people. I am grateful firstly to the anonymous research subjects who lent of their time and expertise to make this work possible, as well as their employers who consented to providing me with access to them.

The collaboration and discussion with other researchers has also helped shape this work. I am grateful to my advisor, Dr. Dewayne Perry, for the many discussions regarding results and methodology, and to the remainder of my committee, Dr. Christine Julien, Dr. Sarfraz Khurshid, Dr. Miryung Kim and Dr. Randolph Bias for their insights and patience as I pursued this rather unique line of research.

I am thankful to the members of the Apache Subversion development community and Project Management Committee from whom I have learned much, both socially and technically. Their patience with my early experiences as the project release manager allowed the planted seed to germinate and sprout.

Finally, I am grateful for my loving wife, Heather, who has supported me in every step of the way, and for our four children: Hannah, Jonathan, Charlotte, and Spencer. I could not have done this without each of you.

Release Engineering Processes, Their Faults and Failures

Hyrum Kurt Wright, Ph.D.
The University of Texas at Austin, 2012

Supervisor: Dewayne E. Perry

Release processes form an important, if overlooked, part of the complete software development life cycle. Many organizations implement the roles of release engineering and release management in different ways, with a wide amount of variance within the software industry. Ill-designed processes can lead to a higher number of software faults and costly delays. Failures in release engineering can have negative implications, yet the causes of release process failures are not well understood within in the software engineering research community.

This dissertation addresses the questions of what the common release process structure is, what the common failure modes are, and how organizations recover from and prevent these failures. We address these questions through a series of case studies with practicing release engineers at commercial software companies. The live interviews with these individual companies provide insight into the state of the practice in release engineering today across a broad spectrum of organization and software domains.

The results of these studies indicate four areas of theory in release engineering which future researchers can probe in more depth. These areas center around process organization, social causes of release process failure, the relationship between software architecture and the release process, and how organizations attempt to improve release processes.

For practicing release engineers, these results show that most organizations would benefit from three primary improvements: increased process automation, more modular software design, and improved organizational communication and support of release engineering groups. By implementing these improvements, software development companies and the release engineering processes they support will avoid the most common process failures in this critical phase of the software life cycle.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

"Software only has value when it is released," proclaimed one of the subjects interviewed for this dissertation, indicating the important role that release processes have in the software development process. Whether a web service, an open source project, a commercial system, or an internally-developed application, to a user, unreleased software is practically nonexistent software. Delayed or cancelled releases can have many negative consequences for a software project. A high-quality release process forms an important part of a software development strategy to create low-fault and high-frequency releases.

Almost all software organizations of non-trivial size establish formal processes to create releases. These processes may differ significantly between organizations, but they usually share common features. They may be codified, either through technical documentation, the actual tools used to create the release, or, most dangerously, solely through organizational tradition. Developing and maintaining release processes requires a level of effort that many organizations are not willing to invest.

As with any part of the software development cycle, release processes may not always be strictly followed. When deviations to these processes arise, release engineers are often under high pressure to complete the release and may be completing their tasks with improvised tools and recovery processes. This work will also show, that an improperly devised release process can also have software quality implications by allowing too little time or resources for release artifact testing.

Because of the unique position of the release process in the overall software development cycle, anomalies in the release process may have significant impact. These impacts may be quantifiable, such as lost revenue and project delays or more subjective, such as a decrease in organization morale, and project market share. Regardless of the type of impact, the results of this work show that creating effective and timely release processes can improve the overall quality of software an organization ships.

Unfortunately, the processes whereby organizations predict, encounter and recover from release process anomalies are not well understood. Much literature is devoted to software engineering and development processes, but the topic of release engineering is largely absent (see Chapter 2 for a discussion of existing release engineering topics). We propose that by better understanding release processes, release engineers can better predict when process failures will occur, and be better positioned to recover from the eventual anomalies they encounter.

To this end, we address the following three questions in this dissertation and develop four theories in response to these questions. They are:

I What is the common form of release processes?

II What process faults and failures commonly occur?

III What strategies or techniques can help prevent these faults and failures in the future?

Through the empirical case studies used to better understand release processes, we develop theories in the following four areas:

I The structure of release engineering processes

II Common release engineering failure modes

III The relationships between software architecture and release processes

IV Release process improvement

By addressing these questions, future researchers and practitioners will have a better understanding of the nature of release processes and how to improve them.

1.1 Background

Broadly speaking, *release engineering* consists of the part of the software engineering process during which the release artifact(s) are produced. Many software organizations of sufficient size have release engineers or release engineering teams. Although the nomenclature may be common, the roles fulfilled by these groups, as well as the artifacts produced, are as varied as the groups themselves.

1.1.1 Description

The artifacts created by release engineering may vary. Traditional examples include binary executables, installers, libraries, and source code packages. Newer service-oriented-software delivery paradigms provide an alternative to the traditional artifact distribution model. Instead of installing an artifact for local use, users often interact remotely with the software running in a hosted environment. This shift alters the method by which the software is released. While these two paradigms are discrete, in practice, artifacts and their corresponding release processes may exist anywhere along this continuum [45].

Whatever the artifact, the software must eventually be released, and this release process should be viewed as part of the software development process. Traditional software development methodologies, such as the spiral or waterfall models, usually considered release engineering part of the deployment and maintenance phases [6, 41]. However, several of the subjects interviewed in this research treated release engineering as a concern relevant to all stages of software development.

To effect releases, many proprietary and open source software projects employ dedicated release teams that are tasked with building the final shipping product, very literally "engineering the release." The hand off between development and release teams may be a discrete step, or the separation between the two contexts may be more nebulous. As is the case with the types of artifacts produced, team composition exists along a continuum, rather than conveniently constructed taxonomic divisions.

1.1.2 Context

For many development teams, release engineering can usually be broken into several phases: stabilization, validation, and publication (as shown in [16] and [29]). During the entire process, the release is overseen by one or more *release engineers*. This individual or team may coordinate release activities, determine schedule, and make binding decisions regarding releases. Depending on the size of the development team, release management may be a dedicated assignment, or may rotate among team members.

Release processes evolve as organizations change. A small startup company with a single product will release software much differently than a multinational vendor supporting multiple product lines. While developers and architects focus their energies on the design of the software itself, in many cases the release process tends to gradually evolve, with very little comprehensive design. Instead, companies and projects add release engineering techniques as required, often with little forethought or structured design of the process. As release faults can increase project delay (as shown in [50]), there is often little time for process examination and improvement. The research presented in this dissertation shows that organizations often attempt to improve their release processes after such failures, but resource demands to do so are large.

1.1.3 Consumers

Release artifacts produced by the release process are targeted toward consumers, or users of the software. These consumers may fall into two general categories: *internal* and *external* [42]. The type of consumer a software product is targeted toward often impacts the release process.

1.1.3.1 External Consumers

External consumers usually expect the software to function *as is* with little interaction with the producer of the software. They may buy the software in a box, or download it through the Internet, but the developer often has little control over the environment the software will be deployed in and how it will be used. Software of this type is often mass-produced with the intention of being sold to a large number of individual users.

1.1.3.2 Internal Consumers

Internal consumers of a release are often entities within the software producer's own organization. A common scenario for this is a hosted software service, where the company building the software and the company deploying the software are the same. A development team may build a system and then hand off deployment to an operations group, but the target environment is generally much more constrained than in an external consumer scenario.

Developers of software intended for internal consumption often have the luxury of knowing their intended deployment environments, workloads, and datasets during the development process. They can also coordinate with release engineers and developers during and after the release process.

In the realm of release engineering, communication is one of the distinguishing factors between internal and external consumer environments. When providing software to external consumers, the feedback channel may be limited, usually to paid phone support, email or issue trackers. Internal consumers often have direct access to developers and can provide rapid feedback or assist with release-related issues. Interestingly, as the cost and customizability of the software product grows, the distinction between internal and external consumers is often blurred.

1.1.4 Release Timing Models

Release managers employ a number of strategies to time their future release plans, but two major models stand out: time-based and feature-based [28]. In addition, a hybrid between these two models often emerges as a compromise option during the release process.

1.1.4.1 Time-based Releases

The commonly accepted definition for a time-based release process is one that follows a strict calendar-based schedule, aiming to release major versions of the software at regular intervals. While some projects allow for schedule slippage, releases usually aim to ship as close to the target date as possible. Time-based releases can often be a successful strategy, particularly for a highly modular project.

1.1.4.2 Feature-based Releases

Alternatively, feature-based releases focus on completing a set of features prior to release. While this feature set may change as the release process progresses, feature-based release processes usually include feature collections that are difficult to separate due to their interdependencies. Feature-based releases require release manager and developer discipline to coordinate and plan proposed features, as well as ensure the plans are properly executed [10].

1.1.4.3 Hybrid Model

Some organizations employ a combination of feature- and time-based release models. This hybrid model recommends releasing at regular intervals, but also attempts to have a firm collection of features in the release. Alternatively, release managers may choose a set of features for the release and then set a time-line for the implementation and release of those features. As with many other aspects of release engineering, release models exist along a spectrum.

1.2 Definitions

In addition to the above descriptions of release consumers and timing models, throughout this dissertation we use a variety of terms, which are defined below.

Monolithic Architecture A software project comprising a number of highly-coupled components or modules. While they may be packaged separately, there is still a high level of interdependence between them. The Linux kernel described in [7] is an example of a monolithic system.

Modular Architecture A software project comprising loosely-coupled components, whether packaged together or independently. An example of a modular software system is described in [19].

Release Artifact The software bundle created or deployed for the release; the product of the release. It is usually a binary package created by a build system but may be a script or other object provided to consumers.

Release Failure A release event which does not meet organizational standards. Examples include a release artifact with a high level of faults, releases that require extra time to produce, and releases that require extra personnel to create.

1.3 Personal Experience

In late 2006, the author became involved in the development of the Subversion open source project [1]. During the course of participating in the Subversion development community, he volunteered to coordinate the project's release under the title of Release Manager. The project had an extensive release process in place, yet the creation of the Subversion 1.5 release resulted in number of process and product anomalies[1]. Thus, finding the causes of these deviations from process, and the accompanying methods to prevent them, is not only a topic of general interest, but one of personal interest for the author.

1.4 Motivation

In searching the literature to better understand the causes of the process failures the Subversion team experienced, I was struck by the lack of concrete research about release processes in the software engineering community. Combined with a large number of anecdotal conversations with industrial release managers, this inspired me to apply the degree of academic rigor to release engineering to which other subjects are treated by the software engineering research community.

Thus far, release engineering has been area of scattered research. The research outlined in this dissertation does not purport to fully address the general topic of release engineering or circumscribe the field as a whole. In-

[1]A discussion of these anomalies was published as [50], and is included herein as Chapter 3.

stead, the author address an issue of practical importance to practicing release engineers, namely, how release processes fail and how those failures can be predicted and recovered from. The final result of this research is an improved understanding of release processes, their faults and failures, and ways to improve the state of the practice.

1.5 Summary of Contributions

In summary, this dissertation presents:

- A personal case study experience with a difficult release
- Several case studies of release processes from various proprietary development organizations
- Theories derived from these case studies in the areas of:
  - the structure of release engineering
  - causes of release engineering failures
  - relationships between software architecture and release process complexity
  - release process improvement

Chapter 2

Related Work

This chapter examines the related work in the field of release engineering. Even with the relevance of the topic, very little research has been done into the release process itself. This chapter will outline the existing research in the field of release engineering, and how this dissertation complements the existing research.

As Michlmayr outlined [29], development and maintenance of software projects have historically been considered discrete steps in the development processes. Existing research in the area of release engineering and management can likewise be broken down into the multiple phases of deployment and maintenance. With the advent of the Internet, however, these distinctions are beginning to blur [45] as it enables more efficient distribution models.

2.1 Release Engineering

The open source world has provided a particularly fertile—if biased—source of data for examining release processes. Erenkrantz outlined release processes in several open source projects [16], but the work is quite dated and each of the projects surveyed has changed their process in the interim.

Michlmayr examined release engineering processes in open source projects, and problems that the open environment presents to the process [28, 30], such as the difficulties in coordinating release tasks across geographically distributed volunteers. Additional researchers have also examined how release managers fit into the onion model of open source projects [13].

Release engineering is also an integral part of the larger software engineering processes. Software processes have been modeled and analyzed [48], but the analysis has not yet been applied to the entire release process itself. Nor have specific release process models been generalized to include the entire process, preferring instead to investigate release engineering on a micro level. Generalized release process models would give researchers and practitioners a common framework in which to address process concerns and suggest improvement.

Improved software development processes have been shown to decrease the development time and effort between releases [21], suggesting that study of release processes themselves would be candidates for study and improvement. This result both validates and motivates our contribution because it does not specifically address how release processes can be improved, only that by so doing, development time will decrease.

Process improvement has also been shown to decrease the amount of faults in software projects [14], although most literature in this area focus on the development process and not the release process. Through our interviews with release engineers, we explore how improvements to the release engineering process can also improve the quality of the released software.

2.1.1 Release Timing

As part of this micro-level investigation, several researchers have tried to establish models defining the proper software fault/feature ratio at which to publish a software release [31]. Levin and Yadid extended this work to look at incremental update or “bug-fix” releases [26].

Releasing software with known issues is never comfortable for developers, but eventually the marginal cost for finding the next bug outweighs the marginal benefit obtained by its elimination. Several researchers have introduced various models to help determine the minimal cost of a project given the probability of faults and the cost of finding them [23, 25, 24, 51]. In many cases, these models rely upon parameters that are not well-defined in real systems and are thus overly simplistic. Some papers attempt to counter this issue by using artificial intelligence to improve release timing [15].

These studies primarily come from the software reliability literature and are focused on creating low-fault, yet cost-effective software. Unlike the contributions in this dissertation, they focus on artificially-constrained models, instead of the process used to create the software artifacts.

As mentioned above, additional work looks at how release cycles can be shortened by improving development processes [21]. Some researchers have also developed prototype tools to assist with the release planning process [9], but neither of these efforts look at how the process can be improved and what particular failures cause timing delays.

While this work involving release timing is useful, it does not address the fundamental question of the release process itself, nor do these studies address the highly variable nature of the release process. It may be possible that the variance introduced into the release timing by process-related concerns outweighs the marginal time required to find the next bug, but this is yet unfounded.

2.1.2 Deployment

Some authors have looked at where the release process fits in the software development cycle. The problem of distributing the release artifact is and managing dependencies addressed by van der Hoek, et al. [45]. While useful, this neglects the issue of the process used to create the release artifact, which this dissertation addresses. Other related work describes the Software Dock [20], a system for configuration, deployment, and maintenance of software installations.

These systems describe how to deploy software, but do not address the key problem of creating the actual deployment artifact, nor the faults encountered in doing so. While deployment could be considered an important part of the general release process, release engineering encompasses much more, such as artifact generation and testing.

2.1.3 Maintenance

Once software has been initially released, it typically undergoes a period of maintenance, which may last the entire operational lifetime of the software system. This maintenance phase has itself been the subject of much research, including that of Perry, et al. [37]

2.2 Relationship to Other Disciplines

Software development is often compared with other product development disciplines, from building bridges to making automobiles [39]. While many parallels with other engineering disciplines do exist, at some point the analogies break down, and software engineering must be approached as a problem unique unto itself. This research addresses software release engineering on its own right.

In the generic engineering space, the term *Product Lifecycle Management* (PLM) refers to the organization of a product throughout its entire life, from conception to destruction [43]. Most frequently, this is applied to physical products, which have production and distribution costs, and may even have defined decommissioning and destruction costs. While the amount may vary, physical products always have some form of marginal (per-unit) cost associated with their use, which PLM seeks to help track and manage.

While many of the lessons of PLM can be applied to software products, software engineering is unique enough to warrant specific study [5]. The marginal costs of a software product are often near-zero, given the fact that duplicating and distributing software via the Internet is now commonplace. Thus, the majority of software product costs are fixed in the development stage, where process improvements can have a dramatic impact.

In short, while existing research in engineering Product Lifecycle Management may appear useful at first blush to apply directly to software engineering, the difference between disciplines warrant research directly on and applicable to software release engineering and management.

2.3 Limitations of Existing Research

In summary, the existing research consists of surveys of existing practices, analyses of how these practices influenced overall project productivity, or tools and models to attempt to find optimal release timing. The research does not look at the overall process, process failures, or develop general areas of theory surround release engineering. This work overcomes these limitations by taking a holistic view of the release process, and developing theories from observations of real release engineering processes.

Chapter 3

Initial Study

This chapter describes an initial analysis of release anomalies encountered during the release cycle of a prominent open source project. Much of this information was collected via first-hand experience with the project in question. This study led to the questions raised in Section 3.4, and ultimately to the studies described in Chapter 5.

Apache Subversion [1] is a popular version control system whose initial goal was to replace the aging Concurrent Versions System (CVS) with a more modern design and feature set. With the release of Subversion 1.4.0 in September 2006, these goals were largely accomplished, and the development community focused on making additional improvements to Subversion. These included features requested by both open source users and corporate deployments, with the primary one being merge tracking.

Although Subversion had a well-established process for crafting releases, the process broke down during the subsequent feature release, Subversion 1.5.0. This process failure led to frustration in both the developer and user communities, and this initial study focused on how these problems could be prevented in the future.

3.1 Subversion Release Process

In the early days of the project, Subversion developers established a guiding document known as the "Hacker's Guide to Subversion" [3]. Colloquially referred to as HACKING, this document outlines many aspects of community processes and procedures, including release processes. Although the community allows for circumstantial variation in these processes, HACKING is fairly specific as to how the release process should proceed.

Crafting a release of Subversion involves many individuals in a coordinated effort following established procedures. In the following sections, we describe the roles these individuals fill, the different types of releases, and the version numbering scheme for Subversion releases. We also describe the process used to create a new feature release of Subversion. In Section 3.2, we compare the ideal described here with what actually happened when releasing Subversion 1.5.0.

3.1.1 Community Roles

In a large and complex open source community, such as Subversion, different members take on different roles within the project. Individuals may fill more than one role, (i.e., a person may be both the release manager and a committer), but the roles themselves are distinct [13]. Below we describe the pertinent roles in creating a release of Subversion.

3.1.1.1 Release manager

The release manager for the Subversion project is a volunteer individual who oversees the entire release process. Typically one of the developers, the release manager coordinates branching dates, signature collection, tarball distribution, and release publication and announcement. Rarely does the release manager make unilateral decisions, but his voice is influential in directing the release process and coordinating discussion within the community.

3.1.1.2 Committers

Committers are individuals with full commit rights to all locations in the Subversion source code repository. How an individual becomes a committer is beyond the scope of this paper, but the primary qualifications for this designation are good judgment and trust within the community. As part of the release process, committers run independent tests of the candidate tarball[1] on a platform of their choosing. Upon successful completion of the tests, they provide cryptographic signatures verifying the integrity of the release.

3.1.1.3 Third-party distributors

Rarely do users make direct use of Subversion source code as provided, and the project itself does not provide binary packages. Instead, a vibrant community of third-party distributors provides binary packages of Subversion

[1]Tarballs are a standard source code release artifact on many POSIX platforms.

for various platforms.[2] Because of Subversion's well-documented APIs, many third parties build tools on top of the Subversion libraries that integrate with other platforms and environments. While not directly involved in the release process, the feedback from these consumers helps validate API consistency between releases and provides important testing during the validation of a potential release.

3.1.2 Versioning Guidelines

Subversion has adopted the "MAJOR.MINOR.PATCH" release numbering strategy, similar to that used for the Apache webserver [16]. The version numbers allow users to know what compatibility guarantees they can expect between different releases.

All releases with the same MAJOR.MINOR numbers are considered part of the same release *series*, with MAJOR.MINOR.0 being the first release in the series. Subsequent releases within the series are considered patch or bug fix releases, and the project guarantees that several important parameters, such as APIs and on-disk working copy database formats will not change. Thus, users and API consumers can know that interfaces will stay consistent between patch releases. New features are never delivered as part of a patch release.

[2]One informal poll at a meeting of Subversion users indicated that not one in a group of over sixty professionals used the source packages as provided by the project. When deploying Subversion, these users all relied on third-party packages.

Changes to the MINOR number result in a new feature release. These releases contain new features and may change database formats on both the client and server. Features releases are promised to be backwards compatible, both in features and APIs, and work with old database formats. Newer releases can read and write older formats, but old releases are not guaranteed to be able to read newer formats—though they often are able to.

In addition to code and database compatibility, all releases with the same MAJOR version number are compatible client-to-server. Older clients may not be able to take advantage of more advanced features in newer servers, but they will still be able to communicate. This compatibility is both forward and backward.

3.1.3 Release Procedure

For several years, Subversion has used a hybrid between feature- and time-based release strategies. Feature-based releases define particular releases by specific features, while time-based releases use strict timetables to determine release dates [30]. In Subversion's hybrid model, the developers would wait some amount of time, usually around six months, determine which features were completed or nearing completion, and use those to define the next release.

Several weeks prior to a new feature release, a release *branch* is created for that release. This branch is a snapshot of the main development branch, *trunk*, and is used for bug fixing and stabilization prior to release. This branch is ideally created at a time when *trunk* is considered stable enough for release, but frequently the need arises to perform additional stabilization on the branch prior to releasing.

To port fixes from *trunk* to the release branch during stabilization, committers nominate and vote on specific changes or groups of changes. A change must receive three positive votes from different committers to be approved for inclusion in the release. Any change may be nominated, but successful nominations are for changes which fix a known bug, increase performance in a non-invasive manner, or fix known API problems. Any committer may veto any change.

When the release branch is considered sufficiently stable, a release candidate (RC) is created from the branch. This release candidate is just that: a candidate for what will eventually become the official release. The RC enters a period known as the *soak*, a four-week waiting period during which early adopters are encouraged to test the potential release. If no critical errors are found during the soak, a final RC is created, which eventually becomes the new feature release. If a critical error is found, the release manager publishes a new RC with the problems fixed and restarts the soak period. Table 3.1 shows the historic times for creating Subversion feature releases.

Each RC, feature release, and patch release goes through a validation process before being published. As mentioned before, committers thoroughly test the candidate using the included unit and regression test suites and, upon successful completion, cryptographically sign the release artifacts. In addi-

Release	Branch date	Release date	Days to first RC	Number of RCs	Days to release	Days from previous release
1.0	19 Dec 2003	23 Feb 2004	63	1	66	N/A
1.1	10 Jul 2004	29 Sep 2004	4	4	81	219
1.2	04 Apr 2005	21 May 2005	1	4	47	234
1.3	28 Sep 2005	30 Dec 2005	7	7	93	223
1.4	05 May 2006	10 Sep 2006	27	5	128	254
1.5	30 Jan 2008	19 Jun 2008	69	11	141	648
1.6	16 Feb 2009	20 Mar 2009	0	4	32	274
1.7	11 Oct 2011	13 Jul 2011	37	4	90	935

Table 3.1: Dates between Subversion releases

tion, enthusiastic users are invited to test the candidate tarballs and provide feedback, but their testing is not counted toward the required number of signatures.

Committers test on the platform of their choice, but the project requires three signatures from testers on both POSIX and Windows platforms, in addition to that of the release manager, for a total of seven independent signatures. When these signatures have been collected, the release manager uploads the release tarballs to the distribution server and publicly announces the release. For each release, the project distributes source code in `.tar.gz`, `.tar.bz2`, and `.zip` formats, a set of dependencies in the same formats, and the signatures generated as part of the validation process.

After the feature release is published, development on the next feature release continues on the main *trunk*. As developers find and fix bugs, they continue to nominate and port candidate changes to the release branch. When

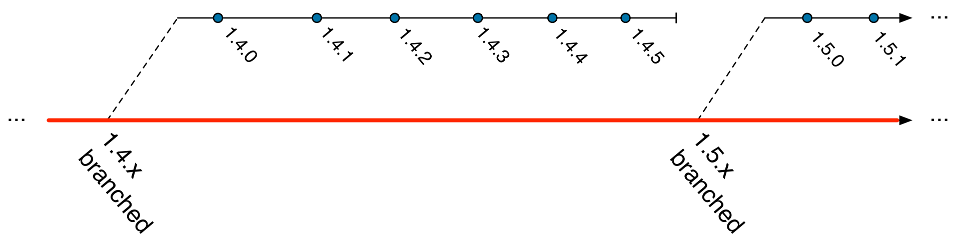

Figure 3.1: Subversion Release Process

a sufficient group of such fixes accrues, a new patch release is issued from this branch, following the same pattern as creating a RC, including committer testing and signature collection. This process may be expedited for serious bugs or regressions. Figure 3.1 illustrates the branch-and-release structure of the Subversion release process.

3.2 Releasing Subversion 1.5.0

Following the Subversion 1.4.0 release in September 2006, the developers turned their attention to Subversion 1.5.0, the next major feature release. Subversion had largely fulfilled its goal as a replacement for CVS, and the developers started looking for ways to further enhance the feature set. The project needed direction and found it in merge tracking.

3.2.1 Merge tracking

Merge tracking was defined within the project as keeping track of which changes occurred on which branches and how these changes have been applied, or *merged*, to additional branches. In Subversion 1.4 and earlier, Subversion required users to manually track this information, which proved tedious and error-prone. Individual users, as well as corporate customers, wanted Subversion to track this information and automatically use it when performing merges between branches. The developers decided that merge tracking would be the defining feature for Subversion 1.5.0 [4].

Work on the merge tracking feature began on a feature branch, a copy of *trunk* used to implement potentially destabilizing features. Feature branches are useful in isolating incomplete or broken code from unwitting developers but have the drawback that code on the branch is not as well reviewed or tested. Six months after creation, the merge tracking branch had grown quite complex but had not yet been merged back to *trunk*.

Several months after merge tracking was started, in March 2007, several developers proposed releasing currently available features in an intermediate feature release, prior to releasing merge tracking. However, the community felt that merge tracking was close to completion, and that any effort spent creating and stabilizing an interim feature release would further delay this feature. Shortly after this decision, the merge tracking branch was merged to *trunk*, and the developers felt that Subversion 1.5.0 would be released by September 2007.

The complexity of merge tracking also hindered development efforts. Only a small percentage of the development community was actively working on the merge tracking feature, and it had grown so complex that additional developers were hesitant to invest the time required to make meaningful contributions. As the release cycle progressed, many individuals knew enough about merge tracking to raise important concerns but lacked the knowledge to solve them.

As the testing of merge tracking progressed, defect rates failed to stabilize, and the developers continued to work to increase performance. Additionally, the initial design was flawed, which required additional workarounds. Internal and external pressure mounted to create a release, in spite of the chaotic state of the code base.

3.2.2 From branch to release

Finally, after a couple of abortive attempts, the 1.5 series release branch was created at the end of January 2008. Fixes began to flow into the branch, leading to an initial alpha release on 22 Feb 2008. This release did not pass committer verification and was quickly followed by a second alpha release on 29 Feb 2008. This was the first time the Subversion project had used the term “alpha” on a release, and both alpha releases contained a number of known issues.

While stability continued to increase, a discussion opened within the project about what to call the next pre-release. One faction wanted to proceed with an RC so the four-week soak period could start, while others, recognizing the bugs that existed were severe enough to prevent an actual feature release, wanted to be more conservative when naming pre-releases. Eventually, the groups reached a compromise, and a beta release was followed by the first true

RC on 7 Apr 2008. This was more than two months after the branch was created (see Table 3.1), an abnormally long time for branch stabilization for a feature release.

Unfortunately, the first RC had critical bugs, and it was not officially published, nor were the second or third RCs. It was not until RC-4 was announced on 24 Apr 2008, nearly three months after the feature branch was created, that the official soak period began. Additional minor bugs were found and more RCs created, some of which were never published due to the near-immediate discovery of still more problems. As the soak period ended, third-party consumers found additional API bugs that required yet more RCs, often with less than a week of separation between them. Over the course of the process, the release manager created eleven separate RCs, five of which would never be released because they did not pass internal validation.

Subversion 1.5.0 was finally released on 19 Jun 2008. This release came after much debate and struggle within the community, but the developers decided to release even with known issues. The prevailing rationale was that postponing the release would do more harm than good, and existing bugs could be fixed in subsequent patch releases. After experiencing the marathon 1.5.0 release process, developers also felt it was time for a change in release processes.

3.3 Discussion

In the several months following the release of Subversion 1.5.0, and as the developers worked toward the next feature release, they identified several places where the 1.5.0 release process failed. The developers planned on using these observations to implement improvements when releasing additional feature releases.

3.3.1 Learn from the past

Despite the transparency of process and free exchange of ideas, open source projects can sometimes be slow to learn from the experiences of others. Sometimes this happens intentionally, but most of the time community members are either unaware of the problems other projects face, being too focused on their own work to notice, or convinced that the same problems are not at play in their own project. A combination of these factors played into the delay in releasing Subversion 1.5.0.

Software projects are notorious for being delivered late [46]. For example, the Emacs text editor released version 22.1 in June 2007, nearly 6 years after the previous feature release. The long development cycle and time between releases frustrated users, who were forced to download and build their own copy of the latest development sources just to have access to features that had already been included in the development branch for years, but were unavailable in the last official release. Developers also felt alienated by the unresponsiveness of the community leadership and frustrated by the long time between when code was written and when it actually shipped [12]. The Subversion community could have seen and worked to avoid frustration by releasing sooner but did not.

Another project that faced problems similar to Subversion 1.5.0 was the FreeBSD operating system [2]. FreeBSD 5.0 languished in stabilization for several years as new features were added and stabilized in an ambitious development effort. As with Emacs, some of the Subversion developers were aware of the experiences of FreeBSD, but no one thought these same problems could apply to Subversion. Naïveté, ego, or both prevented developers from learning from these mistakes in preparing Subversion 1.5.0.

3.3.2 Follow the process

Projects typically create guidelines to assist with the release process, and Subversion is no different. HACKING exists to bring order to the occasionally chaotic nature of open source development and to help newcomers become involved in the project. Consistently following established guidelines can help a project create releases that are both timely and of acceptable quality.

In an effort to publish a release, *any* release, the release manager began putting out alpha and beta releases, without any formal definition of what they meant and how those releases differed from typical RCs. When true RCs did start appearing, there was some question as to their quality; over the course of the release cycle, five of the eleven RCs were never published. The Subversion community did have a process and attempted to follow it, but the process was not designed for such large features as merge tracking.

It was not until the developers neared the end of the soak period and actually started threatening to create the final release that API consumers and third-party distributors started seriously testing the RCs. This led to the discovery of another set of bugs, more RCs, and more schedule slippage in attempting to deliver Subversion 1.5.0.

The community was also unwilling to release code with known issues. No developer wants to release buggy code, but for most users, perfect code—even if achievable—is nonexistent code if it has not yet been released. As the number of changes in subsequent patch releases attests, Subversion 1.5.0 did have many bugs, but none were showstoppers for the release, and most have been addressed in subsequent patch releases.

3.3.3 Time-based releases

Many projects, from complete GNU/Linux distributions to individual software packages, have adopted a time-based release strategy. The theory behind such a strategy is to keep the time between when a feature is implemented and when it is released beneath some known upper bound. We call this the "bus station philosophy": if a feature misses a release, the next will be along shortly, so the release should not be held up for any one feature. This type of process helps keep developers engaged in the project and gives users the ability to plan upgrade cycles around known dates. It also helps developers plan their efforts to allow *trunk* to be in a branchable state when release dates approach.

This type of strategy should work well for Subversion in particular because of the large number of third-party distributors who rely on releases created by the project. These consumers' products often require extensive development and testing to incorporate features new in a Subversion release. During the Subversion 1.5.0 release process, it was not until the developers threatened imminent release that some third-party users started testing thoroughly. Having a well-publicized schedule helps these communities as they build their own products.

Creating "time-based" releases is not a panacea for ensuring a consistent release process. Planning releases becomes difficult in open source projects where resource levels are unknown and constantly shifting. When release deadlines approach and features are not complete, the community has to make difficult decisions about removing features, letting release dates slip, or shipping partial features. In a consensus-based community, such as Subversion, making these decisions can be difficult and require resources that detract from further development.

3.3.4 Defining releases independent of features

Early in the 1.5 release process, developers and other interested parties came to expect that Subversion 1.5.0 would include the much-hyped merge tracking feature. This feature was crucial for a number of potential adopters and heavily desired by one of Subversion's corporate sponsors, CollabNet. As the development cycle continued, it became evident that merge tracking was a complex problem and would take much longer than anticipated, but it continued to define the 1.5.0 release.

Since most of the merge tracking development was happening on a branch, *trunk* was still in a releasable state, and an intermediate feature release could have been created in early 2007. A number of developers floated this idea, but the community ultimately rejected it in favor of focusing on delivering a 1.5.0 release that contained merge tracking, estimated to be delivered by September 2007.

Instead of defining Subversion 1.5.0 as the release that would add merge tracking, the community should have examined which features already existed and been satisfied with creating a release with those features. As a result of a delay in merge tracking, many other desirable features were also delayed, forcing users to run potentially unstable development sources to obtain those new features, much like Emacs users did during the period described in Section 3.3.1.

By defining features independent of releases, developers not only create the opportunity to release more frequently, they also constrain themselves to more modularly designed software. In the case of Subversion 1.5.0, the merge tracking feature ended up being much larger than anticipated and the community ill-equipped to handle it. As a result, instead of dividing and parallelizing the development effort on merge tracking, the developers forced themselves to deliver it as an atomic feature. This also increased testing complexity, further prolonging the release cycle.

3.4 Questions Raised

This look at the release processes of the Apache Subversion project, and the process failures accompanying one of its releases motivated several questions about release processes generally. The include:

- Were these types of failures specific to Subversion, or were they manifestations of general problems other projects also faced?
- If other projects and organizations faced these problems, how did they detect and attempt to recover from these process failures?
- Would these results be specific to the organizations which encountered them or shared across many software development groups?

The desire to answer these questions led to the case studies described in Chapter 5.

Chapter 4

Study Design and Methodology

After conducting the initial study described in Chapter 3, we wondered if the problems encountered by the Apache Subversion 1.5.0 release were specific to it, or if such problems and their solutions were more common. To answer the question described in Section 3.4, we decided to perform a set of multiple case studies with practicing release engineers to gather more information about release process design and common failures in release engineering.

In this chapter, we describe our study design, including interview format and data source selection. Of particular importance are how we chose interview subjects and what effect this has on the results of our study. We also discuss the threats to validity in Section 6.

4.1 Data Source Selection

In selecting the subjects for our set of case studies, we decided to focus primarily on proprietary organizations and their processes. Part of this decision was due to the fact that open source release processes have already received some coverage (see [29]), but more importantly, there are validity concerns with using open source data as the primary and only data source [49], as addressed below.

In addition, proprietary software systems often include additional external impacts which are not always present in open source systems. Business goals, marketing departments, and the presence of paying customers impact proprietary release processes in ways that are not always present in open source systems. For example, business concerns affected the timing goals of releases in Case H, an effect which would not be as marked in an open source environment. These types of issues have a definite impact in release failures, and want to ensure we capture them adequately in our case studies.

4.1.1 Open Source vs. Proprietary Data

Table 4.1 demonstrates how open source data can potentially dominate (and ultimately influence) the results of software engineering research. It shows the results of a brief survey of data sources used in software engineering research papers presented at several ICSE and FSE conferences. This survey, while not completely representative of all software engineering research, does show what the prevailing trends are at the major software engineering conferences. In this survey, we have investigated the extent of empirical studies that use only open source software artifacts (OSS) vs. proprietary source software artifacts (PSS).

It should be noted that open source projects exist along a continuum of *open* development practices and licenses, so this classification is, of necessity, subjective. In classifying the papers, we looked for papers which *used* open

Conference	Total papers	Papers using data	Data source			
			open	closed	custom	combination
ICSE '07	49	39	18	9	10	2
ESEC/FSE '07	42	23	12	5	2	4
ICSE '08	56	36	17	9	5	7
FSE '08	31	19	7	5	2	5
ICSE '09	50	38	22	7	3	6
ESEC/FSE '09	38	20	10	5	2	3
Total	266	175	86	40	24	27
As percent of total with data	—	100%	49%	23%	14%	15%

Table 4.1: Use of open source as data sources in research papers

source data, not just those that built an open source tool or provided their tool under an open source license. Neither did we classify such papers as *open* when their authors implemented their tool as a part of an open source framework. Table 4.1 illustrates the results of our survey regarding the use of OSS vs. PSS.

Of the recent papers at ICSE or FSE that use software projects as study subjects, nearly half use OSS data exclusively, while another quarter use just PSS data. Only 15% of the papers used any combination of OSS, PSS, or custom data (which includes manufactured examples and benchmarks). We hope that the difference between OSS and PSS is not as drastic as believed, lest the validity of a large amount of software engineering research comes into question.

Even though the release processes studied in this dissertation do not concern themselves directly with source code, focusing on proprietary development organizations, rather than open source communities, will better contribute to the body of knowledge in software engineering. Although this technique poses logistical challenges (described below), it will produce more complete results, because it captures aspects of release engineering not existent in pure open source communities, such as those related to marketing and business concerns. This technique better complements the existing body of research.

4.1.2 Subject Selection

Finding release engineers embedded deep within proprietary software development organizations is not a trivial task. We could not simply contact a trade group or visit a convention and solicit opinions. Rather, to reach as many potential release engineers as possible, we sent requests to various software development mailing lists asking for references to practicing release engineers. An example of such a request is included in Appendix B. We also made inquiries among professional networks, and asked interviewees for references to other potential subjects. These methods obviously suffer from various kinds of biases, such as self-selection, but it gives sufficient variety to lend validity to the results.

Our group of interview subjects spans a range of software domains and development methods, from small "agile" teams that release frequently to large organizations that only occasionally create release artifacts. Similarly, the artifact distribution models ranged from deploying to internal corporate customers in a controlled hosted environment, to sending a hard disk with 80GB of software updates with a technician to a customer site, to sending an image to a manufacturing plant for use on new hardware.

Where possible, we tried to interview multiple individuals from a single organization to get a more rounded view of the release process under study. However, such a constraint was often difficult to fulfill, due a number of reasons. First, scheduling conflicts often dictated that only one member of an organization was able to participate in our interviews. Second, even though we granted participating organizations anonymity in publication, many groups wanted to limit the number of interviewees for legal purposes, since our request to interview people often required legal review. Lastly, and perhaps most significantly, many organizations only had one individual responsible for releases, a point further discussed in Section 7.1.1.1.

Each of the interview subjects fell into one of two self-identifying categories: a dedicated release engineer whose primary responsibilities were on release and deployment; or a member of a development team who was also responsible for that team's release activities. Insights from both groups were useful, with many common themes present. Table 5.1 summarizes the organizations in our studies, and they are discussed more thoroughly in Chapter 5.

Many of the subjects had been in release engineering roles prior to their current projects and offered to share insights based on those experiences as well. Where possible, we incorporate that feedback into our analysis, even when it was not directly related to the case at hand.

Prior to our interviews, this study was reviewed and approved by the local Institutional Review Board, with IRB Approval number 2011-01-0041.

4.2 Interview Format

The interviews used as the basis for these studies were conducted over the phone or in person and recorded for future review. The subjects were sent an initial questionnaire explaining the study purpose and their role in it. Appendix A contains a copy of the questionnaire we distributed.

Interviews typically began with a description of the proposed research and an opportunity for the subject to share his or her role within the organization. In later interviews, common themes from earlier interviews were also presented by the interviewer as topics for discussion in an effort to get more thorough coverage of these areas.

After the initial discussion surrounding questions from the questionnaire, the interview subject was invited to discuss topics of interest to his or her organization in the area of release engineering. Sometimes these included in-depth discussion of tools to create a release, or the social problems accompanying the efforts to improve the release process. While not all topics were covered equally by all subjects, the unique collection of comments from each subject provided interesting insights into their release processes. Some of these topics included:

- A description of the software product produced
- The composition of the release team
- How a release is timed
- Description of the product release cycle
- How release artifacts are tested
- How release artifacts are distributed
- Experiences when the release processes failed
- Observations by the interviewee on release engineering generally

This range of topics helped establish a more complete picture of release processes from the people directly involved with them.

4.3 Analysis

After the interviews were collected and recorded, we listened to the recordings, noting common themes throughout the collection of interviews. We also looked for comments by interview subjects about the release process failures and how their organization recovers from them as well as overall challenges to creating a workable release process. We did not perform full transcription or coding but did generate detailed notes about each interview over the course of repeated reviews of the audio recordings. The descriptions of the individual interviews are found in Chapter 5, and the analysis is found in Chapter 7.

Chapter 5

Case Descriptions

In this chapter, we detail each of the case studies used in this research, the interview subjects, the organization release processes, and failure episodes from those releases. Chapter 7 describes the commonalities and differences between these processes and general observations on the release processes and failures based upon these cases. Table 5.1 summarizes the cases described in this chapter along with some of the their defining characteristics.

Each case is unique. Some interviews go deeply into what steps are required to create release artifacts, while others focus on team structure and the social efforts needed to create releases of complex software systems. While each interview is different, they complement each other to create a view of dynamic release processes.

Likewise, the role of the interview subject for each of these cases is also unique, which reflects on the differences among release techniques. Some interview subjects are full-time developers who only do release engineering responsibilities when called upon, while others spend all their time focused on release and manage entire teams in doing so. These varied roles help add additional perspective to our review of release processes.

Case	Software Type	Consumer	Release Type	Software Architecture
A	Python-based application environment	Internal / External	Hybrid	Modular
B	Social-networking platform	Internal	Time-based	Monolithic
C	Online property-rental provider	Internal	Time-based	Monolithic
D	Network router control software	Internal / External	Varies	Monolithic
E	Online publishing software	Internal	Continuous	Modular
F	Payment system for online rental-property provider	Internal	Time-based	Modular
G	Network appliance manufacturer	External	Unknown	Unknown
H	Binary packages of open source system	External	Time-based	Monolithic
I	Software-as-a-service provider	Internal	Time-based	Monolithic

Table 5.1: Case Descriptions

Also, it should be noted that while release processes and failure descriptions are accurate, product and company names have been changed in the interests of preserving anonymity among with interview subjects.

5.A RJD

The company studied in Case A is primarily a consultancy, which builds custom software tools for their clients, often in the industries of finance or scientific applications. Many of their tools are based upon the open source language Python. Company A produces and maintains a number of packages to enable them to meet the needs of their clients. These packages are themselves released as open source software as part of a larger software distribution, while the domain-specific software required by clients is kept proprietary.

The interview subject in this case was a developer who also acted as the product manager and release engineer for RJD.

5.A.1 Product Description

In order to facilitate the easy use of the Python-based system and the additional packages produced by Company A, the company also creates their own distribution of Python, known as RJD. This distribution includes their own open source packages, other third-party packages, and some proprietary components. In total, RJD includes almost one hundred separate components integrated into a single released product.

Unlike the packages themselves, this distribution is *not* open source. RJD itself ships as a standalone distribution, but is also used as the platform for the software Company A provides to customers. Two different versions of the product are produced: RJD and RJD-Free. The former requires a paid license, whereas the latter is distributed for free, although it is not open source. RJD-Free provides a reduced set of functionality from the full version but also includes an upgrade path to users who want to upgrade to the full-featured RJD in the future. An RJD distribution must also support a number of different platforms, resulting in a number of distribution artifacts for a single release.

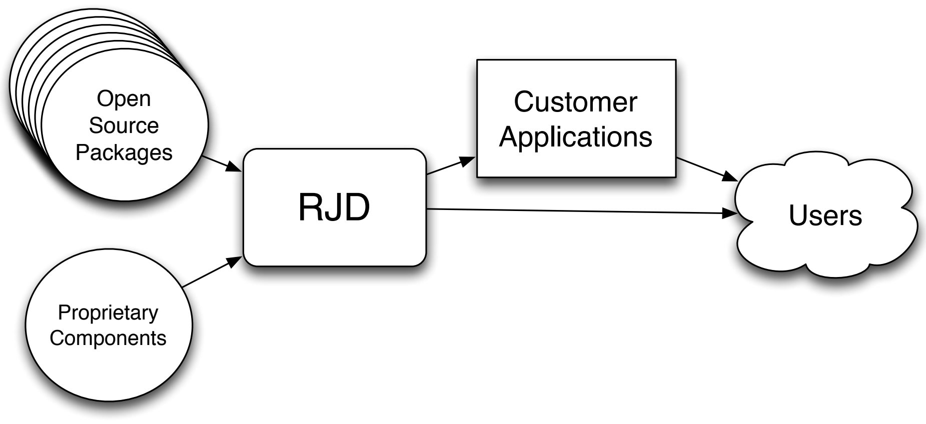

Figure 5.1: Upstream packages and downstream consumers of RJD

Thus, RJD has both *internal* and *external* consumers, though ultimately the software is destined for users outside of Company A. Figure 5.1 illustrates this relationship.

5.A.2 Release Timing

Company A tries to release RJD every three to five months, but this schedule varies. The needs of both upstream packages and downstream users can influence the timing of a release with their own release schedules. As a collection of packages, RJD is very modular, and the release manager can use this modularity to choose to update or hold back certain components from a particular release.

As proprietary customer applications are built upon the RJD platform, the timely release of RJD impacts not only consumers of the stand-alone distribution, but also applications Company A writes for their clients. Sometimes business reasons dictate that a release of RJD is brought forward to enable the in-house applications to take advantage of new functionality.

Similarly, when upstream packages are on the verge of releasing updated versions of their own projects, Company A may delay the release of RJD in order to incorporate these updated packages, both for their own benefit, as well as that of external users. Occasionally, the release manager for RJD will contact upstream packagers to help coordinate release schedules, so that RJD releases are both timely and fresh.

5.A.3 Release Team

The release team for RJD consists of a single individual whose role is to coordinate and manage releases. The release manager's role includes monitoring upstream packages as well as the needs of downstream consumers. He fills this role in addition to development and product management responsibilities.

5.A.4 Release Cycle

Similar to Subversion, RJD releases are cyclical. At the beginning of the cycle, the release manager collects package updates, additional requirements from internal downstream consumers, and other content that should be included in the next release. This process is aided by a "release dashboard," which shows the status of various input components to RJD and assists in estimating the time for the next release.

During this period, as packages are updated in the RJD distribution, a suite of integration tools runs to ensure compatibility between updated packages within the release. Since RJD is a collection of many components, this testing helps ensure these components will function together. Packages that pass testing are stored in a repository for later use.

After a major release of RJD, the release manager creates subsequent patch releases to address minor bugs or faults in the release. These patch releases only include minor changes so that users can safely upgrade to them in place. This work usually occurs in parallel with the beginning of the next release cycle.

5.A.5 Pre-release Testing

Prior to a release being made public, the artifacts are uploaded to an internal distribution site and then tested. These tests cover the various operating systems that RJD supports and target both the installation process as well as the various components that come as part of RJD. Since most of the emphasis of RJD focuses on the component packaging, the testing revolves primarily around the installation process, the provided graphical user interface, and the compatibility of the various component packages.

Almost all of this testing is automated, with manual testing being limited to brief tests for basic correctness. Throughout our interview, the subject referred to “the tests,” and made no distinction between regression, feature or other types of testing. He did describe the role of the testing as one of ensuring proper integration of the various dependency components and the proper production of the installer package.

5.A.6 Distribution

RJD is distributed to end users via a complete download from the Internet (along with associated license files) or through an in-product upgrade process.

5.A.7 Release Tools

To create a release, the release manager utilizes a suite of custom scripts and tools that build the various components of RJD and then combine them into a master distribution artifact. These tools can be scripted to perform all the required steps to produce a final release. Intermediate components that pass testing are stored in a repository, which the final tool then uses as input. This modular approach enables more rapid release artifact creation.

To enable this modular release process, each release begins with a release plan, which is then codified into a recipe. The various release tools make use of this recipe to determine which versions of packages should be built and included in the final release. Because many individual components of this recipe are orthogonal to each other, the release can be built incrementally, and small changes do not require massive artifact recreation.

The testing of the release is also automated. Manual testing is involved but only as a "sanity check," as described above.

5.A.8 Changes in the Release Process

At the time of our interview, the release manager mentioned that the release process had been changing. The interview subject had been the release manager for several months and had been transitioning the release process from a more complex one that a previous manager had used to a simpler one.

The previous release tools used a much more monolithic procedure, with the entire release being built using a single command. This required a complete and time-consuming re-build of the entire release whenever changes to the release or tooling were made. Thus, small changes to subcomponents would trigger a time- and resource-consuming rebuild of the entire release. Because of these requirements, this system required a team of five platform-specific people to manage releases.

With all the resources the previous process required, little time was left to perform post-release testing. The interview subject called the old system "a total disaster," where "everyone was trying to get *something* and when we had something, that's what we'd call the release." The current process requires fewer resources, allowing more time for release artifact testing.

5.A.9 Process Failures

Most of the failures in the RJD release process have been eliminated by moving to a more modular artifact build system, and the process has not recently experienced significant delays or problems. In some cases, delays are caused not by technical reasons, but by external ones, such as marketing and infrastructure needs. While they impact the release process, these issues are mostly about scheduling and package inclusion, and they are resolved through consensus by the several concerned stakeholders within the company.

5.A.10 Summary

In summary, key lessons learned from this case include:

- Complex processes took too much time and resources, leading to faulty releases.
- Communication with other teams helped to better meet their needs, as in the case of working with upstream packagers and downstream users.
- Modular processes and tools improved release quality.
- Modular processes decreased release team size.
- Business concerns, which are tangential to technical issues, also impact release content and schedule.

5.B Connect

The company studied as Case B produces and hosts an online social networking platform, known as Connect. The company produces its own software and also manages all the deployment and hosting of the software.

The interview subject was a member of the release engineering team, whose role had previously been one of managing releases, version control and continuous integration systems. As the company grew, these roles became parts of separate groups, and his role became to focus only on release engineering.

5.B.1 Product Description

The software for Connect is mainly written in Java and contains over 300 services that interact to produce the features displayed to customers. Releases consist of these services bundled as web applications to run in a J2EE container. The artifacts are these bundles, which include compiled code, and a configuration to be deployed on hardware owned by Company B. This follows an internal-customer release paradigm.

Even though the services could be deployed separately, the interconnected nature and high coupling of the services limits this ability. As a result, the interview subject characterized the software architecture is more monolithic than modular, which, combined with the desire to do rapid releases, can create a large amount of friction in the release cycle. This friction results from the need to release all components simultaneously.

5.B.2 Release Steps

To create a release artifact, a specific revision is first checked out, it is compiled and built, and the resulting artifact is published to a binary repository. These binaries are versioned for traceability and reproducibility. After testing, the artifacts are then deployed to production environments.

Because the software ships only to internal customers, and these steps are relatively lightweight, Connect can to be built and updated frequently.

Problems often occur not in the creation of the release artifacts themselves, but in deciding what should go into the release and finding software faults before they are published.

5.B.3 Release Cycle

Company B uses a “cadence” for Connect releases, wherein they attempt to release new versions every two weeks. The size and complexity of their software prohibits faster release cycles, but through experimentation, they have learned that two week cycles allow for enough time to responsively provide new features and bug fixes.

In the past, Company B attempted a quicker release cycle for only certain classes of low-impact changes, such as critical bug fixes. However, this was quickly co-opted by the product team to deploy new features, rather than just bug fixes. These features were often hastily done and not fully tested, which in turn led to the problem of “testing in production,” as users often uncovered software faults.

As a result of these experiences, Company B only considers bugs that are serious enough to impact revenue when breaking the regular two-week cadence cycle. Even in these instances, these exception cases are tightly controlled by product managers.

5.B.4 Tooling

In the course of our interview, the release engineer spent some time discussing tooling, both in a general sense, but also as it pertains specifically to Connect. Having written build and release frameworks in a number of languages for a number of different systems, his thoughts were illuminating.

The key insight is that managers should think of release engineering as a traditional manufacturing process, rather than a box with a collection of tools. A disparate collection of tools and skills results in a mindset where only certain people are trained for specific tools, and the tools are viewed individually instead of as a piece of the overall process. This method leads to a breakdown in the overall ecosystem.

Specifically for Connect, having multiple tools for various phases of the release process led to a scenario that was not scalable as the system grew larger. Lack of an overall model of the process drove this problem. In the interviewee's words: "you have to be able to model your process in order to automate it, and all of your process tooling and machinery has to be driven by that model." The interviewee suggested that working from a model, rather than a collection of tools, has led to an improved process.

The interviewee also observed that the use of tools can be a hindrance if they impose too much structure to the release process. His organization has seen some success by moving to a build system that allows for much more customization of the tasks to make a build, but such a system requires more discipline on the part of the release engineers. This discipline further shifts the issues of release engineering from the technical to the social domain: by introducing automation, resources become available for other tasks.

5.B.5 Release Process Failures and Deficiencies

When asked about deficiencies in the release process, the interview subject responded that the these failures had been one of intense debate within the company to that point. Even though there had been much discussion about the failings of the current processes and how they could be improved, no consensus had yet been reached. Release process improvement was a work-in-progress at Company B.

One of the key factors was ownership of the process. The subject remarked that in order to be successful, somebody has to "have ownership. There has to be a group, a role which owns the release process." The current Connect process does not have such an entity, with different aspects of the release process being split among a number of teams. The result is that the process is static and difficult to change since such change requires coordination between several entities.

This lack of ownership is manifest in a couple of different ways. In one case, the group that owns the resources to build release artifacts is separate from the people who are tasked with actually creating those artifacts, and there is not a feedback loop for effective communication between them.

The software architecture also influences the release process. The complexity of the software and its large degree of coupling means that dependencies between different modules are hard to untangle. A change to a specific module of Connect may induce a need to redeploy disparate parts of the system, and these parts may have dependencies themselves, which compounds this problem. The problem of deploying a specific module then becomes the sum of the problems of the transitive closure of all its dependencies.

Relatedly, during development every component is always built directly from source, but during staging and deployment, this is not the case. In those environments, cached versions of pre-built binaries are used to speed artifact creation, but since this process is different from that used by the developers, discrepancies in the resulting product may arise. Company B has largely moved away from "environment-specific" builds, but other problems, such as compatibility or configuration issues, still arise.

Differences between the various development, staging, and production environments still pose problems, however. For one example, the database size and quality is often different between them. The staging area does not contain data comparable to the production systems, and since most of Connect depends upon data to function, this difference means that testing often falls short.

Commenting on the general cause of release problems, the interviewee said that "very few of the challenges around release are technological" and that some of the most challenging items he had faced were due to social issues. Because development methods and developer behavior have such critical impacts on the release process, educating developers was an important part of working to improve the release process for Connect.

One of the additional social problems the interviewee mentioned was a desire to imitate other companies' release processes, without taking into account the unique aspects of the language, the technology stack or other components of one's own system. The subject remarked that there is not a one-size-fits-all approach to release engineering and that problems and solutions are often specific to a particular piece of software.

Finally, business concerns often impact the release process. Within Company B, there exists a constant attempt to balance the need to follow best practices with the business need to ship products in a time-sensitive manner. Previous release failures often resulted from attempts to short-circuit established release procedures in an attempt to ship features rapidly. In the interviewee's experience, this often resulted in delays and longer times to release.

Generally, to recover from release failures, the team tries to push forward in fixing the problem, rather than attempting to roll back to previous changes.

5.B.6 Summary

Key lessons learned from the study of the Connect release process include:

- Social and organizational problems in release engineering include:

  - A lack of ownership of the release process can impede attempts to improve it.
  - Attempts to imitate other release processes without considering issues unique to the target software product can lead to failure.
  - External business pressure to ship software for revenue purposes impacts release timing, content, and success.
  - The absence of feedback between developers and release engineers can also lead to release problems.
- Tool support for release engineering impacts releases in the these ways:
  - Automation can create reproducible release artifacts.
  - Tools should be envisioned as part of the process, rather than discrete components.
  - Tool support should be scalable, so they can continue to function as software systems grow.
- The monolithic architecture of the software project impacts the ability to release individual components.
- Differences between development, testing, and production environments creates opportunities for release failure.
- Many organizational stakeholders are interested in process improvement, but they lack consensus on how to achieve it.

5.C ForRent

The company studied as Case C is a service-oriented software company that runs a website allowing users to advertise and sell short-term property rentals of personal real estate.

The interview subject for this case is a Java developer who performs release engineering tasks for his team. This assignment was largely motivated by his desire to improve his own experience by improving the quality of the release tools he is required to use as a developer, along with prior experience at other companies performing a similar role.

5.C.1 Product Description

The software created by Company C is targeted to their own production environments; the only way that end users interact with the software is through a web interface. The higher-level layers of the complete product produce output that a user directly interacts with, while lower-level layers are services intended for consumption by other software systems.

The various services that ship as part of the application under study were previously contained in a monolithic release artifact but have since been split into separate deployable artifacts. The interview subject was quick to point out, though, that simply splitting code into separate artifacts does not change the monolithic character of the entire application and that a high level of coupling still exists between different artifacts.

The subject pointed out that creating a more modular architecture, as the company has attempted to do, presents its own set of challenges, both generally, and in the release processes. Maintaining a modular architecture implies maintaining a set of interfaces between components, which themselves have to be kept stable between releases, adding additional process and cost. Even if the code itself is modular, the interviewee pointed out that "you still end up releasing everything all at once, because that's the only way you know that everything works together."

5.C.2 Release process

Releases are deployed through test and staging environments. Testing environments are used as features are being developed and any member of the development team can promote changes to the test environment. The release team promotes versions of the software into the staging environment to test them before the releases flow into production.

For each release the team focuses for two weeks on a specific set of features. The features that are complete by the end of the two-week cycle are included in the release, while others may be postponed for the next release. At the end of the cycle, a release branch is created for the release, which the release engineers then promote into the staging environment. From that point, bug fixes may continue to get worked on in the release branch, while the cycle begins anew and new features are committed to the main *trunk* branch.

A separate Quality Assurance team tests the proposed release while it is in the staging environment. If the fault level is deemed acceptable, the release is deployed to production by an operations team. The entire process is coordinated through an issue tracking system, though often real-time communication with the developers is required throughout the process.

5.C.3 Diverse release processes

The history of Company C offers insight into the release processes there. Company C started as a small company but grew through purchasing other similarly-positioned organizations. As it acquired other groups, they continued to build and release their software in their own way, and even now, separate teams work independently and "do what they need to do," although the company is attempting to standardize upon tools and processes. These product teams are still distributed globally.

Some of this diversity stems from the inertia of the various tools each group is using. One example the interviewee cited was a group using the Ant build system, for which a company-wide standardization on a different tool, such as Maven, would prove too costly. For some groups, such a transition would be difficult but still doable, depending on the return such an investment would generate.

In some cases, various development teams use the same tools, but in different ways to meet their individual needs. As a result, the tools are customized for specific teams and products, and processes diverge from the intended standard. Though some of this is transparent at some level, it still creates problems as custom uses of specific tools generates little documentation for reuse by other teams.

5.C.4 Process Automation

This diversity of processes means that various teams use different levels of automation. The interviewee noted that automation is a highly valued trait of a release process but that the disparate requirements of various teams within a company limits that automation. Because certain teams manually create their automation tools, and these custom tools often lack documentation, problems arise when the tools are used or abused in unintended ways.

One of the difficulties to automation that the subject discussed was that releases consist of more than just code. Database schemas may need upgrading, which results in content migration, while other external data may also need attention as well. Often, these migrations are one-way, meaning that no safe method exists to revert the changes once they have been applied to production data. For small systems, it may be possible to duplicate the pre-release data, but for large systems this may not be tenable.

5.C.5 Process Failures

Release failures at Company C are rarely code-derived problems, but usually something tangential to the software itself. A configuration value may have been missed, the database updated incorrectly, or the release tools have changed in some way, all leading to process problems.

The interviewee identified a lack of automation in the process and an inability to track configuration changes as key causes of database and configuration problems. Because the process is manual, information must be communicated out-of-band, and omissions or inaccuracies lead to additional failure vectors. The solution, in the eyes of the interview subject, is better automation and communication.

Other failures are caused by changes to the environment the code is running in. Sometimes, these are the result of changes within the production environment that break assumptions made by the underlying code. Other times, it is the differences between the development, testing, staging, and production environments that lead to problems. Although the release steps are the same to deploy code to the staging or production systems, these environmental differences can cause release failure. A specific example of this type of failure was a change to the permissions of a disk mount in a production system, which caused a software update to fail. The solution to such problems, according to the interviewee, is to standardize and automate, both the process to deploy new code into production environments and the changes to that environment that accompany the new code.

Given the current lack of automation and standardization, current solutions to release problems often end up with a cluster of stakeholders physically gathered around one of the operations team members as he attempts to deploy the software. Developers, database administrators, and the operations team all try to help solve the issue, since each possesses a piece of the institutional knowledge needed to do so. While this may work in the short-term, it does not lend itself to long-term solutions.

For larger issues of automation and standardization, the interviewee identified the benefits to doing so, but mentioned that the costs of the existing collection of systems and tools had not yet become sufficient to motivate the business to invest the resources for standardization. He mentioned the very real costs in terms of lack of feature development, as engineers work on improving the tooling, rather than improving the product, and that in a competitive business environment, this may not be feasible.

5.C.6 Summary

Important lessons learned in from the ForRent case study include:

- Standardization between development groups depends on the proper use of release automation.
- In a heterogeneous environment, improved standardization requires acceptance from several groups, which can be difficult.
- Release failures do not have to be software faults, they can be faults in other parts of the system, such as a database or configuration.
- A monolithic software architecture impacts release by requiring all components to be release simultaneously.
- Differences between environments and tools leads to release failures.

5.D NetOS

The company studied as Case D is large manufacturer of networking equipment, and the release process studied is the embedded software that runs the devices. Even though this company manufactures a wide variety of devices, the software on these devices largely derives from a common code base, known as NetOS, which is itself a derivative of the open source operating system FreeBSD.

The interview subject for this case is the head of the release engineering team, whose responsibilities include nightly builds, builds required for testing new hardware, builds that are put on the hardware as part of the manufacturing process, and builds that end users can download to update their existing hardware installations.

5.D.1 Product Description

NetOS is a full-featured operating system responsible for network routing within industrial networking equipment; it is based upon FreeBSD. The software itself has a very monolithic architecture of tightly-coupled components, not unlike the Linux operating system [8]. For example, the NetOS modifications to FreeBSD are not limited to specific parts of the system, requiring a complete rebuild when creating releases.

Additionally, there is not a concept of code or module ownership, but all developers have access to all part of the source tree, with many different teams potentially modifying the same areas of the software. At the time of our interview, NetOS was undergoing a lengthy project to re-architect the code in an attempt to add more modularity and ownership—along with stricter access controls to the source code repository.

The release team for NetOS is responsible both for end-user facing builds, and for generating nightly builds from branches in various stages of development. The technical process to build both is similar, but the artifacts they produce may undergo different post-release procedures and they are often targeted toward different customers.

5.D.2 Release Team

At Company D, the release engineering and release management teams are distinct and have separate roles and responsibilities[1]. Release *management* is primarily responsible for the content of the release from a logical perspective: what bugs will be addressed and which features added. Release *engineering* is responsible for actually building the release and managing both the hardware and software resources to accomplish that goal. Our interview subject was the leader of the release engineering group.

Even though a specific group is tasked with release engineering, there is much interaction with other groups in accomplishing their tasks. Release engineers routinely interact with testing, release management, IT, and development groups, and a significant amount of social effort is required to maintain

[1]The terms used in this section are specific to this company. Throughout most of this dissertation "release engineering" and "release management" are used interchangeably

the trust required to create successful releases.

5.D.3 Release Cycle

NetOS has a cyclical release structure defined by periodic releases but potentially very long development cycles for individual features. Release lines are maintained for a number of years after shipping. Timing and content of releases are often driven by customer needs and market demand.

Development for multiple releases of NetOS occurs simultaneously, as various products require different levels of support. Some release lines may be in a state of feature improvement, while others are open only for bug fixes and other maintenance. The distinctions are maintained by branches within the version control system.

As stated above, the release management group determines which features will be in a specific release, and developers assigned to various tasks check their fixes into the appropriate branch of the version control system. Company D relies heavily upon their version control system to manage release components and their relationships to each other.

Unlike some of the other cases profiled heretofore, NetOS is an embedded system with much different maintenance and longevity requirements, which also impact the release process. Users can not be realistically expected to upgrade their router firmware every week, so releases must be solid. Because of this difficulty in upgrading, NetOS has fewer public releases compared to a web service, which may push out new releases every few days since the costs of doing so are so low.

When the software becomes ready for a release, the release management team creates a release branch in the version control system. The release engineering team then starts making release candidates from the branch. Release candidates use the same build steps and infrastructure as nightly builds and are intended to become releases if no serious faults are found. A critical part of the release process is the creation of a *manifest* file that describes the release and controls how it is distributed if it passes qualification.

5.D.4 Pre-release Testing

After a release is built, the produced images go to a system test group for testing. This testing is a combination of automated and manual test of the proposed images, which may differ depending on the type of build under consideration. If this group rejects the release, the developers fix the issues and the process begins again.

5.D.5 Distribution

Because the target users of NetOS are varied, the distribution model also varies. Builds intended for installation on newly created equipment are distributed through manufacturing, whereas subsequent updates are provided by download from the Internet or a service technician traveling with physical media to a customer location. The release manager, though, is largely removed from the mechanics of distributing the software.

The interview subject indicated that due to software architecture constraints, NetOS is always shipped completely: Company D is unable to distribute incremental patches to the software. A team working on this capability has spent over a year trying to implement it, but so far this capability has not been introduced.

In addition to shipping releases to manufacturing and existing customers, Company D must also adhere to escrow requirements, in that entire releases must be put on a DVD and shipped to a third-party escrow agency.

5.D.6 Artifact Creation

To create a build, the release engineering team, either manually or on an automatic schedule, creates a configuration, or *manifest*, for the release, which then goes into a database. These configurations, as well as the sandbox archived at the conclusion of artifact generation, provide all the requisite information to reproduce a release.

A *sandbox* is the build environment for a release, whether is a release candidate or a nightly build. Sandboxes are similar to the development environment and are created by checking out the branch and revision specified in the release configuration. The automated process then builds the artifacts from the sandbox and archives the sandbox for potential examination later. Sandboxes are used by the support organization to assist in finding and fixing problems for customers. By having access to the various nightly build sandboxes, support teams can more easily determine what changes may have caused or fixed an issue a specific customer is facing.

5.D.7 Release Tools

The NetOS process is heavily dependent upon automated tools for building and tracking release artifacts. When a trigger for a build is activated, either manually or through a time-automated process, the automated system begins checking out, building, and archiving the release. The tools also manage how the release is described and can be reproduced. The inputs, such as what version to build and the target hardware to build it for, are specified manually as part of the kickoff process. The progress of a particular build can be queried through a command line interface or a web page.

Both the inputs to the release, the environment the release is built in, and the log files generated from the release process are archived to allow for later analysis. Releases intended for end-user consumption are also archived on optical media. The entire process requires significant computing and storage resources, as the size of the release sandbox can be quite large, on the order of 150 GB.

To ensure timely completion of builds, the build and release processes are housed on a set of hardware completely distinct from everything else at Company D. A typical build takes several hours, though one of the goals of the attempt to introduce more modularity and ownership of those modules is to decrease this time by avoiding unneeded compilation of unchanged units.

5.D.8 Changes in the Release Process

In an effort to streamline development and make shipping code easier, the entire company that produces NetOS decided to shift from CVS to Apache Subversion for their source code management. The release engineer for NetOS said this change allowed developers to better use branches and other features of the version control system, which further improved the release process. This eliminated much of the friction of development and also release. This was one part of a wide-ranging change in process for the entire company.

Company D was also planning additional changes to the development process. Because the software produced in development is the input to the release process, changes in software architecture or process often impact the release process. The interview subject was unsure what those specific changes would be but was preparing to implement them as needed.

5.D.9 Weaknesses

The release manager who was our interview subject identified several weaknesses within Company D's existing release processes. These are outlined below.

5.D.9.1 Storage

One of the largest restrictions of the release process is the hardware resources used to create and manage the permanent archive. A technical restriction on the size of Company D backup storage limits volumes to sizes of 1

TB. The archived sandboxes are usually around 150 GB, which means only six or seven can fit in the same volume, requiring a significant amount of planning and coordination in creating and managing these volumes. This restriction limits the level of automation available to the release process.

The technical issue of more disk space is a real one for this release manager, but its foundations are more social than technical. Aquiring new storage requires business justification, which is difficult when release engineering is viewed as a cost center by the business. On several occasions during our interview, she emphasized the amount of space required for their sandbox archives and the difficulty in justifying its acquisition costs to her management.

To compound this issue, special releases can put extra demands on the archive system but are rarely accounted for when planning archive requirements. To combat this problem, the release engineering manager has been attempting to move the cost to provide the archive requirement to the units working on these special projects, shifting the burden of resource acquisition to those requiring it.

In many cases, the interviewee mentioned that release engineering can be seen as a drag on company resources, as it is typically considered a support organization and not a revenue-generating unit. The practical implication is that the release team does not have the budget or political influence to accomplish the objectives it has been tasked with.

5.D.9.2 Staffing

The release engineering manager also remarked about the chronic staffing problems that her group faces and the lack of formalized institutional memory. Much of the knowledge and nuances of the release process are known by only a few people, and their loss would negatively impact Company D's ability to produce releases.

5.D.9.3 Social Issues

During our interview, the NetOS release manager pointed out that many of the issues were ultimately social ones, in that they were caused by human decisions and failures, rather than technical ones. The role required a high-level of interaction with a number of different groups, any of which could cause problems with the process.

At times, release engineers would attempt to short-circuit the process by manual intervention and restart of release processes. Such attempts circumvented a number of the protections afforded by the release process and are discouraged. Education of those involved proves to be an ongoing task.

"Integration engineers need to be social," our interview subject pointed out, due to their need to interact with a large and disparate group of people. In Company D, such engineers are also in high demand, and the team is often required to have someone available even though they are only funded to be a business-day organization.

5.D.9.4 Software Architecture

The monolithic nature of NetOS also proved to be a source of trouble in the release management portion of the process. Since no team had clear ownership of specific parts of the software, developers are often making conflicting changes that requires significant time and cross-organizational effort to rectify. This is similar to the phenomenon observed in [35].

The release manager's proposed solution for this is would be group- or team-ownership of specific components of NetOS, which would hopefully lead to fewer conflicts than the current company-wide ownership paradigm. Work on a more modular architecture toward this end is ongoing.

5.D.10 Example of Anomalous Release Experience

A high-profile and potentially high-revenue customer wanted new features prematurely and wanted them in a supported set of release artifacts. A custom version of NetOS was produced for that customer, who proceeded to install it beyond the intended subset of trial scenarios. The features included in this custom release had not yet entered any shipping product and were considered experimental.

In producing the release artifacts, the release engineer had no technical way of differentiating the custom and experimental nature of this release versus a nightly build or a traditionally supported release. In essence, this was a development build, packaged as a supported release. Because no such ability was available as part of the release process, the release engineer was required to manually adapt the automated process to accomplish her goal, introducing delays and errors.

Additionally, because of the implied guarantees surrounding a release of NetOS, the manager of the release engineering group required the requesting group to get high-level approvals from within Company D before she would create such a special one-off release. This was both for the resource requirements involved, as well as the desire to indemnify herself should the release cause further problems in the organization.

In total, the special effort required for this release was significant, but the prospective customer decided not to purchase the product.

More generally, a desire to ship customized products from the same monolithic code base leads to many custom releases of features from various branches, which creates management logistics problems. Significant amounts of resources are being spent on these custom release targets.

5.D.11 Summary

Important issues about release engineering discussed in the above study include:

- Social difficulties with release engineering included:
  - Organizations often place unrealistic expectations of the release engineering teams.
  - Pressure from outside business units can cause release engineers to deviate from established process for custom releases.

    - Developers often have antagonistic feelings toward release engieneering teams.

- The monolithic software architecture constrained the ability to release independent components and bug fixes.
- The boundaries between release engineering and other part of the software development process are not well-defined.
- Sales, marketing, and support interactions all eventually influence the release process.
- Good automation can still lead to release failures if manual intervention is required.

5.E Publish

The company studied in Case E is a major provider of online news content. Writing their own software allows editors and authors to manage this content dynamically, giving them more editorial control and flexibility.

The interview subject for this case is one of the engineers responsible for creating the release pipeline used to move the software from development to production environments.

5.E.1 Product Description

This software controls the publishing platform for Company E's content, with requirements to make the content available in multiple languages and highly configurable. This software, known as Publish, exists to tie various components of Company E's ecosystem together to allow non-technical staff the ability to create and publish content. To do this, several separate software modules come together to create the final page the end user views.

The Publish software is responsible for orchestrating the interactions between these various pieces of software. Specifically, it exists to integrate the publishing tools, back-end API layer, the front-end, and previews into a single package. This package is deployed on servers owned and maintained by Company E, so this software's customers are entirely internal.

Even though the software has defined boundaries between itself and various subcomponents, features are often co-developed, resulting in coordination between various teams as the software components move through the release pipeline. This pipeline has several environments, each with their own dataset and hardware, and dependency components are expected to maintain these environments for consumers to use when testing. These environments include development, staging, and production.

5.E.2 Release Processes

The team that the interview subject works with has recently changed its release process from being a segmented part of the development process to something that occurs as part of development. Software release is visualized as *flowing* from one stage of development to another, with the result being that it ends up in production.

The interviewee pointed out that there is not a sense of being "done," as the software is being continuously adapted and improved. In this respect, there is never a completed software product, as new requirements continue to be added and implemented.

The lack of a sense of completeness, combined with the continuous flow of software, means there is not a definite meaning of a release *cycle*, as software components are packaged together and deployed to production servers *ad hoc*.

5.E.3 Tools

To accomplish the goal of software flow, new tools were recently written, including a custom build and release pipeline, which has the stated goal of delivering software every week or every day. To accomplish this, the pipeline supports the notion of "software hiding," which allows incomplete software to be deployed in an inert fashion with little consequence to production systems.

Developers check finished work into source control. The continuous integration system builds the subcomponent and runs the various static and dynamic checkers. Assuming the tests pass, the component is packaged and joins the pipeline to be assembled with other systems to go into production.

For release, all the various packages are bundled together into an immutable super-package that then traverses the pipeline as an atomic unit. The only exception to this atomicity property is the interface specification for the super-package, which changes depending upon the environment it is deployed to.

5.E.4 Testing

The continuous integration platform helps catch issues during development and prior to code entering the release pipeline. New code is continuously built and run through the automated tests prior to being promoted up the pipeline, and any faulty versions can easily be backed out of the pipeline.

One important part of pre-release testing is backward compatibility with the dependency modules. One of the features of the staging environment is the ability to test against both the current and *future* versions of a dependency package, so that when the future version is eventually promoted, Publish does not encounter backward incompatible changes. Agreements exist between the various teams to ensure that such incompatible changes are fixed within a short time window, usually on the order of a few hours.

5.E.5 Challenge with the Process Tools

While the above flow and tooling represent an ideal scenario, this goal has not yet been completely realized, for several reasons. First, not all dependency teams are set up to provide appropriate testing environments with the appropriate data. For instance, dependencies that have existed for a long time, such as user management, do not have the resources to fully implement a staging environment with full production data, due to both the scope of the data and the maturity of their existing platform.

Some dependency teams have the opposite problem, in that they are new and immature and thus do not have the appropriate response time or service level guarantees to enable the rapid finding of issues. This decreases the general effectiveness of the continuous deployment pipeline.

The current release tools are not yet properly componentized, which causes a lack of flexibility in the release process. The software is composed of multiple dependencies and components, and if a single component fails, the system will currently reject the entire bundle as faulty. The interview subject hopes to one day have a system that would simply exclude the faulty version of the component, using an earlier iteration to complete the release. Such a system will reduce the number of teams able to block the release flow with problems in their software.

5.E.6 Observations

The interview subject made some interesting observations regarding release engineering as it applies to his process. Specifically, he cited the example that when a developer checks in software that breaks other parts of the system, there are often additional people watching for that breakage, and the developer quickly fixes the problem due to the social convention of not allowing broken code into the system. Such principles, applied generally, help create the right incentives to prevent social-based release process failures.

The interview subject also emphasized how important it was to him to create a permissive environment for software release, rather than creating barriers to release. He emphasized balance between features and bugs and a concious acknowledgement that not all released software is fault-free. Strong functional tests ensure that major features work and the software continues to flow.

One of the key points he made was that software only has value when it is released and that providing this value to the business is what matters. Taking release processes into account should be seen as part of the process of maximizing developer productivity, not as counter to it. Defensively designed release processes can impede the release of software with the goal of ensuring better quality. However, such processes can often have negative effects when attempting to release timely fixes to software.

The subject also touched upon the impact of release engineering in the greater development cycle, mentioning that small changes to release and build processes can have large results since they impact a large number of people. In doing so, the subject cited the theory or constraints [18] as being an important, but often overlooked, part of designing release processes.

5.E.6.1 Previous Experiences

The release engineer also shared some experiences with failure in previous experiences with release process at other companies. Some of these include monolithic release blobs, problems with late integration of feature branches during development, and test systems built *ad hoc*. All of these issues led to more faults in the software and more resources spent fixing those faults.

Because the release process was not automated, testing was not a priority. The software had to deploy to over forty different types of environments, but only one or two of these environments were tested before the release because manual testing was too expensive.

5.E.7 Summary

The key lessons learned from the Publish case study are:

- Not all releases are discrete; Publish uses a continuous “flow” paradigm.
- Modular software architecture allows individual features to be released or held back.
- Maintaining a modular architecture requires discipline between teams to maintain and support appropriate interfaces between releases.
- Improvement to the release process can have wide-ranging impact because of the number of people affected.
- Social implications: release engineers should facilitate code release, rather than look to prevent it.

5.F ForRent Payments

The company studied for Case F is the same as that for Case C, namely a producer and provider of a social networking platform known as ForRent. In Case C, the interview subject noted that the company had a number of diverse teams working on various projects. The process studied herein as Case F is a separate set of code within ForRent, and while the overall structure remains similar, this case has its own set of unique failure modes.

The interview subject for this case was a member of the development team for the payment processing system. While not a dedicated release engineer, he does perform release duties on a rotating basis with the rest of the team. Within a release cycle, the then-current release engineer usually spends about twenty percent of his time working on release-related assignments.

5.F.1 Product Description

This product is a subsection of the aforementioned ForRent system (see Section 5.C), focusing specifically on the payment processing aspects of ForRent. The entire ForRent system is conceptualized as a collection of RESTful web services [17] interacting through a set of well-defined and understood interfaces.

5.F.2 Release Tools and Environments

The ForRent payment system uses the same hierarchy as the rest of ForRent, with one crucial difference. The development, testing, and staging environments are the same, but because this specific piece of software works with money transfers, the production environment is much more secure. As a result, the expense and resources of replicating this special production environment in lower levels of the release stack has not been made. In addition, these restrictions mean that developers have much less access to the production environments than a typical process, making diagnosing and fixing problems more difficult.

A set of automated tools assists in moving code between environments and making releases. The system tracks developer commits to the source code repository and automatically builds release artifacts. Developers and release engineers can select these builds for deployment on the testing system. The quality assurance team then has the opportunity to write and run functional tests against this new code in the testing environment for about a week, after which it is promoted to the staging environment.

The staging system is envisioned to be a mostly-stable environment where integration testing with other components occurs for another week before the software is release into production. The same tools accomplish all the promotion steps, with the developer/release engineer managing the first two steps, while the operations team coordinates the promotion to production.

5.F.3 Release Failures

The interview subject indicated that the largest set of release failures was due to the differences between the development, testing, staging, and production environments. Separate property and configuration files, which are not part of the standard release artifacts, define these differences on a per-environment basis. The interviewee mentioned that almost all of the release problems for his team are related to the inadvertent misapplication of this configuration information during the release. The problem is easily solved by updating the configuration, but preventing it is largely an issue of remembering the manual update step outside of the typical release process.

The other common failure mode occurs when the modularity and abstractions of the system break down. In the interview subject's words:

> You have this service oriented architecture, and then suddenly that's out the window, because it's not really specific services. It's just a bunch of services that are really just one big service, because they're all dependent upon each other.

Ensuring separate services is dependent upon maintaining a stable interface between those services. In the experience of the interviewee, when those interfaces work, it makes producing releases much easier since components can be deployed independently. Maintaining a stable interface requires effort, but his experience was that the results are usually worth it. Conversely, when the interfaces break down, the release process requires much more coordination between components.

5.F.4 Summary

In short, the ForRent Payments study addresses the following points:

- Non-standard production environments require non-standard, and potentially fragile, tools.
- Maintaining interfaces in a modular architecture requires effort, but is ultimately worth it as releasability improves.

5.G WebCan

The company studied as Case G manufactures networking applicance hardware to be deployed behind the firewall for large enterprise customers. This company also produces the software that runs the devices it sells, and it is this software which is under study in this case. This software runs the internal hardware and provides interfaces for external measurement and diagnosis of these appliances.

The developer interviewed for this case has spent several years in release engineering for a variety of different companies before filling his current role. At Company G, he works with the tools team, helping to manage the version control system and other infrastructure needed for both development and release. While he is not directly involved in release engineering, he works with the release management team and is familiar with their processes.

5.G.1 Release Process

Releases from one interaction to the next are generally incremental as bugs are found and reported, and development managers and release managers meet to discuss which issues should be addressed in subsequent releases. Using a checklist system, issues are tracked, and when the list of issues targeted for a specific release is completed, that release is the shipped.

Releases happen on the order of months, anywhere from two to four, depending upon the issues slated for that release and the ability to find and fix them. The release is then provided to the tech support group, which coordinates with their members in the field to get the release into the hands of customers.

5.G.2 Previous Experiences

In addition to his current role, the interviewee shared several insights based upon previous work doing release engineering at a number of his previous companies.

5.G.2.1 Process Design

The subject shared an experience at a previous employer when a committee was attempting to design a release process. One of the participants in that committee pointed out that the people carrying out the process are in fact human, and as such would make occasional mistakes. He felt that by creating complex processes, the committee was setting these people up for eventual failure. This insight led to more simplified release processes in that company.

The interviewee also recounted numerous instances when processes were altered or ignored, usually due to priorities dictated to release managers. These alterations typically led to failed releases, that had to be fixed at great cost to the company involved, negating the perceived benefit of altering the process initially.

5.G.2.2 Causes of Faults

The interviewee also described causes of faults he had seen among earlier employers. One of these was making assumptions in the process description, which led to omitted incorrect steps. Sometimes this itself is caused by sloppily adopting previous processes for the project at hand and not being zealous in updating the process for subsequent projects. One of the ways of coping with this problem is to ensure each build creates an installable artifact, which can then be tested to help maintain quality.

Another cause of faults that the interviewee identified occurs when people become "slaves to the Process." In other words when the process itself, and not the release, became the goal, and not the release, participants could overlook other causes of failure in their singular focus on the process. In addition, when faults did eventually occur, people were often more focused on finding and affixing blame, rather than fixing the problems, which then led to further problems.

The interviewee also drew the connection between software architecture and process design, in that software that is modular can reduce process scope and thus the potential for failure. Since components of a loosely-coupled software system can often be released independently, their release processes can themselves be compartmentalized, helping make the overall system process easier to conceptualize.

5.G.3 Summary

In summary, the main points emphasized by the interview subject for Case G were:

- Complex release processes provide more opportunities for process failure.
- Lack of automation can result in assumptions about the release process.
- Both developers and release engineers respond to social incentives to improve the release process.
- An open question exists as to whether release engineering is a technical or managerial role.

5.H Subversion Binary Packages

The company profiled in this case creates binary packages of the Apache Subversion open source project discussed in Chapter 3. Because the open source community does not distribute or endorse binary packages, the roles of producing and distributing artifacts suitable for most end users falls upon third parties. Company H is one such third-party who creates binary Subversion artifacts for a number of platforms as a means to increase adoption (and the market for their proprietary products).

The individual interviewed for this case fills several different roles within the company, but one of them is the release manager for Company H's Subversion packages. In this role, he monitors Subversion releases, writes and manages the automation software required to build and package Subversion distributions for various platforms, and coordinates the distribution of the artifacts to those platforms.

5.H.1 Release Process

As a redistribution of Apache Subversion, the release processes for these binary packages is somewhat time-dependent upon the release process of Apache Subversion itself, described previously in Section 3.1. Company H attempts to release the binaries as closely after the formal Subversion release is announced as possible, for both major and minor releases. In order to meet this time constraint, the binaries' release process begins when the open source project posts candidate Subversion release tarballs for pre-release testing and continues in parallel with the Subversion process.

To actually generate release artifacts, the release manager downloads the Subversion candidate tarballs and combines them with a set of scripts appropriate for the target platform. After editing the documentation included in the release artifact, the release engineer runs the scripts to build and package the release. After the candidate binaries are produced, they are staged to an internal distribution system for further testing and to await the announcement from the open source community to signal the final release.

The testing primarily focuses on the artifact packaging, while relying upon the upstream open source community to catch faults in the software itself. This testing step is not automated, and the release engineer identified the testing process as being a bottleneck in the release process.

5.H.2 Process Automation

The process described above is a mix of automatic and manual steps. Items such as compiling the binaries, running the tests, and packaging the results into a artifact for distribution are done with automated scripts, while documentation and other tasks are handled manually. The level of automation also varies depending upon the target platform and the knowledge of the release manager on that platform. For instance, the release engineer is more comfortable on POSIX-based systems, so the tooling is simpler when compared to Windows.

The interview subject specifiably mentioned the lack of continuous integration as a weak area in the process design. Instead of constantly building test binaries using the source code available in the public Subversion repository, the release engineer only builds binaries from release candidate code distributions. The primary reason for this is one of resources: all the binary building environments are virtual machines hosted on the same physical hardware, which would not be sufficient to handle the multiple parallel builds required by continuous integration.

When asked about the possibility of further automating the process, the release manager mentioned that several of the steps could be improved. However, he was not convinced that the effort required to implement the increased automation was worth the time he would save in the future.

5.H.3 Failures

One of the primary sources of failure mentioned by this release engineer was the unfamiliarity with Windows as a target platform. He spent some time setting up an initial build environment several releases prior to our interview, but since that environment is (mostly) functional, he just uses it as-is. When changes to Subversion require modifications to the Windows environment, the release engineer must spend a large amount of time attempting to adapt it. This lack of familiarity contributes to the lack of automation on this environment, as mentioned earlier.

In one case, our interview subject mentioned that the testing process is a key bottleneck in the release process because most of the testing is completely manual. Because of the time pressure and the lack of automation, sometimes problems are not caught in testing. The subject spoke of an instance when some of the lesser-used parts of the distribution had errors that were not caught in testing. The upstream Subversion testers do not test all the platforms that the third-party binaries are targeted for, so relying upon the upstream testing is not complete. Because the tests were not automated, less code was covered, and faulty software was released.

Another specific instance of failure occurred when releasing a new major version of the Subversion binaries. Typically, Subversion bug-fix releases differ slightly in their dependencies and packaging requirements between them, but major releases may have more fundamental differences. In the case of one major release, the engineer did not start working on the packaging scripts until only a few days before the actual release, even though he had been given significant lead time by the open source project. As a result, the binaries were not able to be released in concert with the upstream Subversion source code, resulting in significant angst within the business and a significant amount of extra work for the release engineer. Ultimately, both the release engineer and his manager identified the cause as a "project management failure."

Lastly, the largest problem with the current release process is that it depends upon the release engineer to do it. Because of the combination of manual and automatic steps and the lack of documentation, the current release engineer is the only one who can produce the release artifacts. As he put it: "I can't be on holiday during Subversion releases" because of the business goal for timely binary package releases. This creates additional stress on the entire team as well as the release engineer.

5.H.4 Summary

The key points covered in this case study are:

- Failed releases often are caused by lack of resources allocated by project management.
- Business needs can drive release timing and processes.
- Unfamiliarity with tooling and platforms can reduce the level of automation, increasing effort and the possibility of release failure.

5.I CodeBit

The company studied as case I produces a suite of data processing software exposed to customers via the Internet. This software runs on servers owned and operated by Company I and collects, stores, and processes customer-provided data streams internally. The software is released as a monolith to internal customers. In addition to externally-facing services, developers also create tools destined strictly for internal deployment, such as nightly data processing jobs.

The interview subject for this case primarily fills a development and architectural design role but also assists with releases. Within the company, the release responsibilities for CodeBit are shared by both the operations group and developers, and in this developer's current role, he has been working to help redesign the software architecture to better accommodate improved release processes.

5.I.1 Release Process and Artifacts

The CodeBit architecture is highly monolithic, consisting of a number of tightly-coupled modules, which requires the entire system to be released simultaneously. These releases are produced approximately every three months by using the then-current contents of the version control system. Often, the development and operations teams doing the release just use the most recent contents of the code from the source repository, but occasionally they will use a specific revision if other known issues exist.

The system uses the version control system and the RedHat Package Manager (RPM) to create and distribute releases. As developers write new code, they also create and update deployment scripts for the new software. All binaries are distributed as RPMs, and developers build and test these packages as they implement new features. Even though the software may be split across several discrete package artifacts, from the engineer's perspective it appears as a monolithic project because the various packages are so tightly coupled as to remove the possibility of releasing them independently.

As the time for release approaches, the development team enters a "code freeze" state, during which changes to the main source code branch are restricted. When the operations team is ready to update the servers with the new software, they checkout the latest code from the version control system, build the RPMs, and then push the RPMs to an internal package repository. From this repository they then update the production environments using the standard RPM tools.

Engineers do *ad hoc* testing of the packaging scripts and system during development, but the interview subject did not elaborate on additional software or artifact testing as part of the release process.

5.I.2 Release Failures

Several types of release failures were identified by the developer being interviewed. One of them was a social problem relating to the "code freeze" period before release. Specifically, the developer said that because operations pulls the latest code from the source code repository at the time of release, adhering to this quiet period is critical to ensure stable software, yet the social problem of enforcing developer discipline was difficult to solve. Developers can easily rationalize additional changes during the quiet period, and such collective behavior undermines the stability of the software being released.

As a result, another type of failure often occurs. When the operations group does pull the most recent set of changes from the source code repository, it is possible that the latest version has unresolved, or newly-introduced bugs. Recovery from this type of problem is relatively simple, in that the operations group can just use a previous version from the version control system. The main problem, though, is the downtime caused by the software faults, and the developer effort required to determine which previous revision does not contain them.

Finally, because the software is released as a monolith, release periods are large. This creates the temptation by developers to cherry pick parts of the software to release independently. The interview subject mentioned that most times when this happens, it results in failure, as the new subsystems do not interface correctly with the old ones. In some cases, the newer module would try to read data that did not exist, or access other systems in unanticipated ways. Recovery from these types of problems was difficult, often requiring additional development and an intermediate release to address the issues.

5.I.3 Attempts to Change the Architecture

The developer-architect we interviewed discussed the current effort to rearchitect the software in a more modular way, both for better software organizational purposes, and to facilitate improved release processes. While both benefits were valid in his mind, he said that project managers could actually understand the benefits of improved release processes more readily, and that was the basis upon which the development team got approval to undergo the change.

Although the team readily acknowledged the benefits to such a change, it was not without its challenges. For instance, the subject mentioned that for developers accustomed to envisioning the system as a single monolith, it was difficult then to transition to viewing it as a set of interconnected modules. This social issue was also compounded by the technical one that when releasing a monolith it is easy to see that everything works, while a more modular architecture and release structure could allow untested configurations to be released, resulting in the potential for unknown behavior.

5.I.4 Summary

In summary, our study of the CodeBit release process yielded the following insights:

- A monolithic software architecture requires the entire product be released simultaneously.

- Release failures often occur when engineers attempt to release subcomponents of the monolithic project independently.
- Social issues surrounding release include:
  - Developers may not be disciplined enough to enforce a pre-release "code freeze."
  - Changes to the architecture are resisted because they are seen to limit developer freedom.
- The benefits of a change to a more modular architecture were recognized, yet attempts to do so were resisted for cost reasons.

Chapter 6

Validity

The preceding case studies contain various threats to validity, which we discuss below. A complete discussion of validity is left to other sources (see [40] and [52]), but to be generally useful, this research should have construct, internal and external validity. Here we briefly discuss potential threats to the validity of these studies and its results.

6.1 Construct validity

Construct validity refers to whether specific measurements actually model independent and dependent variables from which the hypothesized theory is constructed. In other words, an empirical study with high construct validity would ensure the studied parameters are relevant to the research questions and indeed measure the abstract concepts intended to be studied.

The target class of interview subjects, along with the semi-structured nature of the interview process helps to provide construct validity in our studies. The interview subjects are practicing release engineers with many years of experience, who are well-positioned to provide insights into release process failure.

Interviewer bias could affect the construct validity of our studies, but we tried to stay as neutral as possible to avoid biasing the subject matter. We also allowed the interview subjects to “wander” and find topics of their own choosing beyond the fixed set of initial questions. This wandering meant that some topics were not fully addressed by all interview subjects, and that different subjects used different terminology to describe similar issues. Possible confusion over terminology, such as “release engineering” itself, could also impact the study validity.

6.2 Internal validity

Confounding factors represent a major threat to the internal validity in empirical studies. As the results in Section 4.1.2 show, selection bias is a prevalent problem in software engineering research, and the same problem could be present to limit the validity of this research. Internal invalidity can be difficult to counter since changes in the variable under observation may be attributed to the existence or variations in the degree of other variables that are related to the manipulated variable but not explicitly modeled variables.

Each of the organizations described as subjects in Chapter 5 are unique, with many different factors impacting their release processes. The common element between these organizations and our interview subjects was their involvement in the release process. That is the area we focused on in our interviews, but it does not remove the possibility that additional hidden factors impact the internal validity of this work.

Because the interviews dealt primarily with past mistakes, there may be some tendency for interview subjects to self-censor their recollection of events. They may downplay their own involvement in creating the failed releases, or they could omit details which may prove embarrassing to themselves or their company. From our experience with the interview subjects, it seemed they were eager to share these experiences as a type of therapeutic exercise, but the possibility for selective memory still exists.

6.3 External validity

External validity refers to the applicability of study or experimental results to domains beyond those under immediate observation. A study is said to have a high degree of external validity if the conclusions hold throughout the study domain. In most scientific disciplines, researchers prize studies with external validity, since the results can be widely applied to other scenarios.

While these results accurately describe the release processes in the organizations under study, one might argue that release processes vary so much both temporally and spatially so as to make these results difficult to generalize. Although this threat to validity is a concern, there are still several common aspects and lessons that are widely applicable and thus generalizable to a wider audience.

Even though the difference between proprietary and open source development organizations may not be as large as generally believed [27, 34], our decision to focus on proprietary software systems could also be a threat to external validity. We feel this threat is justified, but not eliminated, by the additional insights gain by using proprietary systems.

Chapter 7

Release Process Theories and Analysis

In this chapter, we discuss general observations on the release engineering processes described in Chapter 5. From these cases and observations, we also describe four theories of release engineering. Throughout, we present both common and unique failure modes as well as recovery mechanisms across the various cases under study.

7.1 Observations

Following are insights gleaned from studying the study cases, as well as observations given directly by the subjects themselves.

7.1.1 Team Organization

One of the more interesting aspects of release engineering, from an organizational perspective, are the methods various organizations use to organize their release engineering tasks and the teams to accomplish those tasks. For instance, some study subjects have dedicated release engineering teams, some delegate the assignment to specific individuals, while other subjects rotate the release duties among team members. In each instance, software complexity plays an important role, since complex systems, such as NetOS, require more resources than simpler ones. To paraphrase the subject from Case B, the software constrains the release process.

7.1.1.1 Team Structure

Divisions also occur along process lines, separating the questions of *what* should go into a release, from the issues of *when* a release is to be produced and the mechanics of *how* a release artifact is produced. While interdependence may exist between these functions, some organizations give each a large degree of latitude in their operation.

For instance, the NetOS team has completely separate teams for release management and release engineering, whose roles are to define and produce the release, respectively. RJD, on the other hand, has a single release engineer who both defines release schedule and contents, and has built much of the tooling and infrastructure needed to create release artifacts. Table 7.1 illustrates the team structure for each of the cases in Chapter 5.

For groups which had no redundancy, such as RJD or the Subversion binary packages, the lack of additional release team members represents a potential source of failure. In cases where the sole release engineer is not available, or no longer with the company, the institutional knowledge needed to create a release may be lost. Organizations can partially counter this through improved documentation, but in practice having multiple individuals familiar with release process is the most effective solution.

	A	B	C	D	E	F	G	H	I
Dedicated team		x		x	x		x		x
Dedicated part-time individual	x		x					x	
Rotating part-time individual						x			

Table 7.1: Release Team Composition

7.1.1.2 External Communication

Releases are not made in a vacuum, and release teams often have to communicate with a number of other groups and individuals to complete their assigned tasks. The various release engineers we interviewed spoke of interactions with several different groups, including sales, development, support, testing, manufacturing, operations, and IT. The NetOS engineer specifically mentioned that people who work on her team must have good social skills, since external interaction is frequent.

Often times, though, the release team is viewed by external groups as an impediment to their success, and hostile attitudes can develop, as sometimes happened in the past with NetOS. This environment is antithetical to a productive release processes in many cases.

7.1.1.3 Release Engineer Personalities

Several of the interview subjects, such as those from WebCan, NetOS, and Publish noted that release engineers generally fall into two categories: the technical and the managerial. Sometimes the traits are combined into a single individual, but often people who come to release engineering do so through either of those channels. These characteristics often apply to teams and their roles, in addition to just individual people.

The technically-derived individuals and teams, in the subjects' experiences, were often focused on the tools required to automate the release. In their view, these individuals are often pragmatists, treating release as just a necessary set of steps needed to get software deployed. The temptation for this type of individual is to blindly follow whatever checklist is provided, not accounting for any implied assumptions. While the presence of these assumptions may be an error in the process, an astute release engineer will take them into account when following it.

In contrast, the managerial release individual usually focuses more on the management tasks. In some respects, this person cares more about the process and its organization than the technical individual but may lack the ability to best implement that process. In the experience of the WebCan subject, this type of individual may then become focused just on the process itself, which adds overhead, complexity, and the potential for failures.

7.1.2 Software Domain

Other aspects of a software project that affect its release process are the domain of the software, including its usage model, and the development tools used to build and track the release environments and artifacts.

Several of the subjects we talked to produced software for in-house customers or deployment to servers controlled by their organization. Shipping to an in-house organization often means developers can release multiple times in a short period, even several times a day. This requires a low-overhead release process, but also allows for occasional process failures, as restarting the process is not a high-cost activity.

Shipping to internal customers is not without its drawbacks, however. As the ForRent Payments subject mentioned, internal customers are much less forgiving when failures occur because the feedback loop between customer and developer is very tight. From the developer's perspective, this actually increased the motivation to get things right in the release, even though process friction was low.

In contrast, software with a high amount of friction in the distribution mechanism is often more carefully tested before it is shipped. One subject we interviewed recalled working for a company whose software was over 80 GB in size and whose distribution model involved copying the software to a hard disk and sending that disk to a customer site with a technician to assist in the installation. In this scenario, shipping faulty software can require expensive measures to correct, so the release process is much more controlled.

7.1.3 Interactions Between Releasable Components

The software we studied was often a collection of components, or had strong dependencies upon other software packages. Because of this, the release process of one component did not exist in a vacuum but was impacted by the release processes and schedules of packages it was dependent upon. Likewise,

	A	B	C	D	E	F	G	H	I
I	x	x		x	x	x			x
II	x	x	x	x	x	x	x	x	x
III	x	x	x	x	x	x			x
IV	x	x	x		x		x		x

Table 7.2: Cases and Theories Matrix

packages at the top of the hierarchy often coordinated with downstream consumers to ensure compatibility and utility of their releases. Examples of these interactions include RJD and the Subversion binaries, which both depend upon upstream packages as inputs to their own release processes.

7.2 Theories of Release Engineering

From studying the cases presented in Chapter 5, we have developed the following theories surrounding release engineering. While no process conforms completely to each theory, these theories capture the essential elements of release engineering processes and problems in a way that is generally applicable.

The following areas of theory are supported by the case studies described in Chapter 5. Not all case studies supported every theory, as some interview subjects chose to focus more heavily on specific topics. Table 7.2 demonstrates which case studies supported which of the following ares of theory.

1. Identify release components
2. Prepare release schedule
3. Test release components
4. Create release artifacts
5. Test release artifacts
6. Distribute or deploy release artifacts
7. Iterate during maintenance phase

Table 7.3: Common Release Steps

7.2.1 Theory: The Structure of Release Processes

Of the release processes studied, there emerged a pattern as to common steps involved in a generic release process. These steps are summarized in Table 7.3 and discussed in more detail in the following sections. Not all release processes studied contain all these steps, ordering may not have strictly followed this form, and many subjects may not even explicitly acknowledge the steps, but this was the general form of a standard release process.

Each of these steps, and their relationship to the cases under study, are discussed below.

7.2.1.1 Identify Release Components

Many of the case study subjects indicated their method of selecting release components. Some, such as RJD tailored their release contents based upon the availability of upstream components and the needs of downstream users. Other organizations, like Connect and Publish, had release features dictated by market needs. In other circumstances, such as occurred with NetOS, special releases containing custom components were required for end user testing.

In every case, however, some individual or entity decided what features, subcomponents, or functionalities were to be part of the release. This usually happened while the product was being developed, but in some cases that are fully modular, such as RJD, release component selection could occur independent of developer activity. Additionally, some systems, like the continuous deployment pipeline for Publish, allowed immature features to be disabled prior to release, allowing maximum flexibility in determining what is eventually presented to the user.

7.2.1.2 Prepare Release Schedule

After the release components were identified, every case under study applied some type of scheduling metric to create those components. The schedule may have been purely clock-based, as in the two-week cycles for Connect, or variable such as with RJD. Even intermediate releases, such as the nightly builds of NetOS were done on a specific schedule.

The one exception for the schedule strategy was the continuous release pipeline described by the Publish release engineer. Because software is continuously "flowing" through the pipeline to the production system, releases are not orchestrated events.

7.2.1.3 Test Release Components

During development and during the release process, the various release features and components are tested, sometimes using a continuous integration system. This testing helps ensure components are functional and will be the appropriate building blocks for a release. In modular systems, such as RJD and some parts of Connect, once tested these components are archived for later inclusion in the final release.

Some organizations rely upon other groups or entities to test subcomponents, while only testing their own, such as with RJD. Other groups have specific requirements about how these subcomponents are tested and made available for testing further up the component stack. Connect is one example of this type of organization, where underlying services are expected to maintain a reliable level of service to facilitate testing.

7.2.1.4 Create Release Artifacts

Creating the actual release artifacts is one of the more variable steps of the process, is highly dependent upon the software domain, and depends on both the type of artifact created as well as the intended audience. For most of the cases studied, the release artifacts consist of some binary object or collection of objects with an intended target.

For processes with internal customers, the release artifacts are usually targeted to a specific hardware and configuration platform. Indeed, one of the failure modes mentioned in the ForRent payments system was a difference in the expected hardware and a failure to deploy the correct configuration information with the release artifact.

Processes with external customers, such as RJD, have to be more liberal in the platforms they target and the variety of artifacts produced. The release engineer for RJD produces and tests artifacts for a number of different platforms, and in past iterations of the process this caused a significant amount of additional work and resource usage.

Anecdotally from our studies, it appears the rise of the Software-as-a-Service business paradigm means that more organizations are creating software that ships to internal customers, which then impacts the release artifacts they need to produce.

7.2.1.5 Test Release Artifacts

After the release artifacts are produced, they go through a series of testing. The nature and scope of this testing varies widely among the processes studied, as well as the people and methods used to perform the tests. Many of the organizations studied in our cases use automatic testing tools to assist with this testing effort, with manual intervention as required.

The focus and extent of the release artifact testing varied widely between the different cases. For example, the RJD team focuses their testing on the artifacts and installation process, relying upon previously-completed component testing to ensure the contents are valid.

Conversely, the NetOS release process involves a whole team of testers employing both manual and automatic tests. In the case of the ForRent payments project, custom tests for new features are written as part of the release process, while the artifacts are in the test and staging areas.

7.2.1.6 Distribute or Deploy Release Artifacts

Upon completion of testing, or some other method deemed to declare the release artifacts suitable, the artifacts are then released or deployed. This step also varies widely depending upon the business needs and software domain. In the instance of NetOS, this process consists of providing images to manufacturing or making them available for end users to download. Company A also provides RJD artifacts as a download for end users. These are typically organizations with external customers, who require a self-contained package to install or update the software.

For many of the software-as-a-service providers we studied, including Connect, ForRent, and Publish, their customers are purely internal, so the concept of releasing the software usually means promoting the artifacts to a server with the ability to make changes to real user data. These organizations may even use the same tools to move releases to production as they do through other steps in the process, as is the case with ForRent.

Because there is increased friction, our observation is that organizations with external customers have longer release cycles, as in the case of RJD and NetOS. For these pieces of software, release failures are more costly, in that creating a new release and redeploying it to end users can be an expensive and time-consuming process.

Internal releases, while potentially much simpler, are not without their own negative consequences. The interviewee from the ForRent payments project mentioned that deploying to internal customers is less forgiving than external customers because demands are high and the feedback in the case of a release failure can be swift and harsh.

7.2.1.7 Iterate During Maintenance

Though not strictly part of the release process, most software organizations, particularly those shipping to external customers, use a maintenance phase. During this period, bug fixes and low-impact changes are made, but new features are often withheld for the next major release. While possible in some situations with internal customers, in our studies, we found the “release branch” phenomenon to be a feature of RJD and NetOS—both of which are systems with external customers.

7.2.2 Theory: Causes of Release Engineering Failures

Our theory of basic causes of release engineering failures as derived from the preceding case studies is as follows:

1. Social issues are often the dominant cause of release failures.
2. Lack of automation and inappropriate tool support can hinder successful release processes (and better tools can improve them).
3. Process complexity is the dominant internal product cause of release

failures.

7.2.2.1 Social Causes

While the release process can be highly dependent upon the architecture of the software being released, failures in the release process are rarely due to the software itself. Instead, failures can usually be attributed to configuration issues, lack of communication, poor infrastructure or social problems within the releasing organization.

Many of the individuals interviewed cited social issues as the root of their release engineering failures. Release teams can vary from being tightly integrated with developers, to operating on completely separate organizational units and schedules. For those who are disjoint from the developers, it becomes difficult for them to easily coordinate releases with the developers, which adds additional friction and potential for failure in the release process.

Often, release managers trying to improve this problem are rebuffed by their superiors who see little business sense in improving the project's ability to properly manage releases. One subject noted the lack of support and funding to improve the release infrastructure as a serious detriment to her ability to deliver proper releases.

This problem may also exist in open source communities, where participants self-select tasks and may stop trying to improve release management beyond some local optimum [29]. Each member of the team is satisfied with the state of the release process and has little incentive to improve it, even though such improvements may be generally useful to the project.

7.2.2.2 Automation and Tool Support

Effectively using tools to automate artifact creation helps to prevent process failures from happening and recover after they do. Issue trackers, version control systems, continuous integration and deployment systems all play a role in helping to create a release. The ease of use of these tools, and the extent to which they are put to use, can impact the quality and timeliness of the release product. Interview subjects in Cases A, B, D, E and I all discussed the impact of automation and tools on their processes.

The NetOS release engineer had recently switched version control systems and in doing so indicated that the switch allowed the team to better control feature development and release. By improving the tools the release engineer used, she was able to deliver more targeted releases in a more effective manner, ultimately improving the quality of the software produced.

Many of these tools provide opportunities for scripting and automation of release tasks. We found that of the cases studied, the amount of automation varied from fully-automated to processes that involved several manual steps. The manual steps were often perceived as points of weakness, where unintended process changes could enter, causing failures.

Several of the developers and release engineers we spoke with also expressed an interest in improving the automation of their systems, but they mentioned the social issues associated with doing so, such as overcoming established developer habits and convincing management of the profitability of the improvement. In many cases, project management was hesitant to invest the required resources to improve the release process automation. Although the benefits were readily apparent to the release engineering staff, such improvements were difficult to sell to higher layers of management.

Finally, a high amount of tooling and automation helps make the release process and artifacts reproducible. Several of the release engineers we spoke with emphasized the need to be able to reproduce a given release artifact on demand, primarily for debugging or recovery purposes. Some of the subject organizations had such an ability and went to great lengths to maintain it, while others lamented the fact they did not, and such had been the cause of problems in their organizations.

7.2.2.3 Process Complexity

Process complexity, usually a result of the software architecture (as described below) often led to process failures, either directly or by incentivizing participants to skip or alter steps. In cases such as NetOS, processes with many steps and participating individuals sometimes led to perceived special circumstances during a release, and these perceptions justified altering the established release process.

When such process deviations occur, they often have to be done manually, since no automation support exists for them. This manual intervention leads to increased ability for human error to affect process validity and more opportunities for process failure. Combined with the obvious issues of omitting important steps, process alteration leads to a high probability of failure.

To counter this type of problem, several of the release engineers we spoke to suggested a simpler process, with few steps and actors, along with a more modular software architecture, would help to solve these issues. However, they noted that instituting simpler process was itself a difficult proposition, given existing institutional inertia for the current system.

7.2.3 Theory: Relationship Between Architecture and Process Complexity

Our theory regarding the relationship between software architecture and process design and complexity is as follows:

1. Monolithic architectures induce complex release processes.
2. Modular, loosely-coupled architectures allow much simpler, possibly incremental, release processes.
3. Modular architectures may still develop cross-component dependencies that necessitate an all-in-one release process.

Many of the organizations and individuals we spoke with described their software release as being heavily tied to the architecture of the system. This relationship is analogous to the long held belief that communication structures often mirror the organizations that generate them, also known as Conway's Law [11].

Systems that exhibited a large, tightly-coupled design and code base often require large and complex release processes since many modules of the code are interdependent. This interdependence practically dictates that release processes are highly coordinated affairs, requiring large amounts of effort, involving many individuals, and only occurring with low frequency. They can also be high-risk efforts, as a failed release results in the sunk cost of a large amount of resources.

In contrast, a smaller, more modular software system with fewer interdependencies often requires less effort to release, leading to more frequent releases and individual releases of various subcomponents of the entire system. This often resembles the agile software development methodology, which encourages modular features and incremental releases [22]. Anecdotal evidence of this phenomenon is observable in open source ecosystems where the mantra “release early, release often” historically correlates with small, modular software components and rapid release cycles [38].

Loosely-coupled systems are not without their own risks during release, however, as compatibility interfaces must be maintained and supported for use by older dependent components. The design of these interfaces, as well as their continued maintenance can introduce significant overhead in the development process, even though it may make releases easier to perform.

Likewise, cross-module dependencies may evolve in complex feature scenarios, thus breaking down the interfaces between modules and negating the benefits of a more modular software architecture. As the ForRent subject put it: “you still end up releasing everything all at once, because that’s the only way you know that everything works together.”

The trend in several of the cases we studied was to move toward more modular architectures, with one of the perceived benefits of being more flexibility in their release process. Each of the interview subjects in these cases hoped that a more modular process would improve both the technical aspects of creating artifacts, and the social aspects of fixing faults and issuing subsequently improved releases. Interestingly, this contrasts with a well-known example of a move to a monolithic architecture and the complications arising therefrom [44].

7.2.4 Theory: Process Improvement

Our theory of release process improvement as derived from our case studies can be summarized as follows:

1. Release processes can be improved through modularization of both the process itself and the software architecture.
2. Organizations recognize these facts, but implementing improvements is often difficult for technical or social reasons.

Most of the interview subjects we spoke to had recently been through, or were currently experiencing an attempt to improve their release process, with mixed amounts of success. From our interviews, there appears to be a sense that release engineering can be a pain point within an organization, and that concerted effort is required to improve it.

The NetOS release engineer took a different approach, pragmatically recognizing the limitations of process improvement: “People keep trying to make software development easy; I don’t think it’s ever going to be easy.” Even with this attitude, she was supportive of her organization’s efforts to rearchitect their system in an attempt to improve both development and release processes. In other words, she felt that product improvement would also result in process improvement.

The improvements to the RJD process resulted in one which was more modular, in that the process itself could be started and stopped in known states. As a distribution of packages, the RJD release manager tests and stores known-good packages for later use in building an artifact, rather than performing all steps of the release sequentially at one time. This may add some complexity, but also improved the ability of RJD to build release artifacts in a timely manner.

The results of these process improvements have largely been positive. The RJD team was able to decrease the amount of resources required to create a release, while at the same time increasing test coverage. In the case of NetOS, the change in version control tools has allowed the release engineers to better plan and manage the contents of releases, thought some problems still remain. The continuous release pipeline of the Publish team resulted in an improvement in the time-to-release, with the hopes that such improvements will continue as the system matures.

Chapter 8

Conclusion

This dissertation has presented work relevant to the area of release engineering. Specifically, we have outlined a series of case studies conducted via semi-structured interview and the resulting analysis of those interviews, which show ways that release processes are commonly structured, how they often fail, and how organizations recover from these failures.

This work is significant for several reasons: release engineering processes have wide-reaching effects on the overall quality of a software product; release processes are a critical, but often overlooked part of the software life-cycle; and an understanding of common process failure modes will help prevent them, improving software quality and decreasing development costs. The goal of this dissertation, then, has been to increase both the state of the art, and the state of the practice.

In conclusion, we outline the research contributions, potential future work in this important research area, as well as three recommendations to practicing release engineers based upon the results of our studies. We believe that our work provides a solid foundation for both future researchers and practicioners to build upon.

8.1 Contributions

This dissertation addresses the following research questions:

I What is the common form of release processes?

II What process faults and failures commonly occur?

III What strategies or techniques can help prevent these faults and failures in the future?

These questions address areas of concern for both practicing release engineers as well as software engineering researchers. To gain insight into these areas, our case study interviews focused on these subjects with practicing release engineerers. The results of this work are four theories of release engineering, specifically:

I The structure of release engineering processes

II Common release engineering failure modes

III The relationships between software architecture and release processes

IV Release process improvement

These theories develop a framework for reasoning about release engineering processes as well as practical knowledge that can be applied by release engineers currently in industry.

8.2 Future Work

The work described in this dissertation does not seek to be the final word on the topic of release engineering or release processes and management. While answering some important questions, several additional areas of potential research have become apparent, and I feel that these are worth mentioning here. Some of these include the use of formal process analysis to better understand release process structure and efforts to assist in process standardization.

8.2.1 Formal Process Analysis

Our work has primarily focused on *qualitative* measures of release processes, and our early results demonstrate that release processes vary widely across organizations. Our interviews have indicated a need for more formal *quantitative* methods for reviewing, comparing, and analyzing release processes. Process modeling languages, such as Little-JIL [47] or Interact [36], may prove useful to aid in reasoning about process interactions and properties [32]. However, due to release process complexity, capturing a complete release process with its many exceptions may require significant resources.

Release process modeling and analysis would also benefit from an understanding of where release engineering fits in the comprehensive software development cycle [33]. Such process unification would be beneficial to both release engineers, and those who look to integrate their efforts into a wider development process.

8.2.2 Process Standardization

As release processes continue to evolve and change, some degree of standardization among them will occur. Future research into release processes can assist such standardization before bad practices become entrenched and difficult to dislodge.

Process standardization could also help to encourage a set of best practices for the release engineering industry. Such knowledge is currently buried within organizations, with very little ability for discussion and learning across release groups. Developing a repository for release process information could help these groups better communicate and standardize their processes.

8.3 Recommendations

Based upon our interviews and analysis, we present the following three areas, which release engineers and their managers can explore to improve release process, reducing failures and their attendant costs. Broadly, these areas are increased automation, more modular process design, and simpler processes with more external support of release engineering teams. While none of these changes may be easy to adopt in a particular organization, our research indicates they will yield long-term benefits.

The end result of these recommendations is to decrease friction in the release process, allowing for more frequent releases, which helps to negate transitory faults in the release process. If releases are occurring at regular and frequent intervals, recovering from problems may often be simply a case of waiting until the next release is due.

8.3.1 Improved Automation

Almost all of the subjects we interviewed cited automated tooling as a method they have used to improve their processes, while those who did not expressed a desire to do so. Automating release processes ensures that all the steps in the process have been appropriately captured in a repeatable way, so that artifacts can be reproduced in a reliable fashion. This suggestion does not claim to dictate which tools should be used or how they should be implemented, only that organizations should strive for as much automation as possible.

By automating the process, an organization also makes releasing easier, reducing the friction of a release and increasing the potential for more frequent releases. Even if these artifacts are not provided to consumers, creating a "push-button" release process helps engineers practice the art of creating release artifacts. Frequent artifact generation also allows for more frequent testing of the release artifacts, helping find packaging and integration problems outside of a typical release cycle.

Automation also serves to capture assumptions in the process, helping ensure that all the institutional knowledge related to the process of creating a release artifact is captured. While automation is not a replacement for adequate documentation and training, it does supplement them. Automation may also allow release engineering resources to be better utilized in handling other aspects of the release process.

8.3.2 Modular Process Design

Likewise, improving the software architecture was often mentioned by our interview subjects as a goal of their organization. For instance, the NetOS team is currently involved in a major effort to introduce a more loosely-coupled system, in part to improve the release experience. These groups hope that smaller releasable pieces of a larger system will allow individual components, features, and fixes to reach the hands of their users more quickly.

Modularity comes at a cost, including the effort required to maintain stable interfaces between components. However, while the introduction of these well-defined interfaces and module boundaries requires developer discipline and effort, we feel, as did our interview subjects, that such effort will be rewarded by improved release processes.

8.3.3 Improved Organizational Support

Perhaps the most important recommendation to improve release processes is to improve the organizational support for release engineering. Release process improvements often come as a result of long-term application of discipline and resources and may lack the immediate payoff resource allocators within a company desire. At the same time, market and business forces may dictate that investing in release engineering personnel and infrastructure is not currently tenable.

However, for organizations that are in a position to do so, they would be well-served by making the effort to improve both the tools and resources available to release engineering teams.

Appendices

Appendix A

Pre-Interview Questionnaire

“Release Engineering Faults and Failures”
Pre-Interview Questionnaire

Thank you for your willingness to participate in this research study. I look forward to visiting with you either in person, or via teleconference. In preparation for our interview, please consider the following questions regarding your experiences with your product’s release management procedures. You do not have to return this document; it will be used as the basis for our interview. As previously disclosed, all responses will be appropriately anonymized prior to any publication.

Definitions:
- **Release Process**: the part of the software lifecycle from when a product is declared “feature complete” and when it is actually deployed or shipped.
- **Process Failure**: A failure to follow established processes procedures, whether documented or undocumented (traditional). In other words, a *deviation* from existing process
- **Process Fault**: A deficiency in the process itself. Process faults represent opportunities for process improvement in future iterations.

I. What are the typical release processes in your organization?
  A.Who participates?
  B.What steps are involved?
  C.What is the approximate timeline?

II.What are some instances when your organization encountered process **failures** during the release process?
  A.What prior plans were made for working around these failures?
  B.Why did these failures occur?
  C.What tools or personnel were involved in recovering from the failures?

III.What are some instances when your organization encountered process **faults** during the release process?
  A.How did the faulty process impact the release?
  B.What changes did you make after the release to prevent a similar experience?
  C.Who was responsible for identifying and implementing these changes?

Appendix B

Sample Subject Solicitation Mail

```
To: xxxx@apache.org

From: xxxx@xxxxx.org

Subject: Soliciting Release Engineering experiences

Date: 17 May 2011


Hello fellow ASFers:


I'm in the midst of doing a research project on release
engineering.  As part of the research, I'm collecting
information about release processes, and how those processes
fail, either via technical or human means.  To do so, I am
interviewing release managers / engineers from both open
source and proprietary development organizations.


If you are a release manager, or know somebody that works
in release management / engineering for you project or
company, I'd be interested in interviewing you.  The
interview shouldn't last more than an hour, and can be
```

scheduled at your convenience. All results will be properly anonymized prior to publication, and would be welcome to review them. (I'm hoping to interview two individuals from each organization, but one-man groups are also of interest.)

Please contact me off-list if you are interested in participating or have additional questions.

Thanks,
-Hyrum

Bibliography

[1] Apache Subversion. `http://subversion.apache.org/`.

[2] Choosing the FreeBSD Version That Is Right For You. `http://www.freebsd.org/doc/en/articles/version-guide/`.

[3] Hacker's Guide to Subversion. `http://subversion.tigris.org/hacking.html`.

[4] Subversion Merge Tracking Notes. `http://subversion.tigris.org/merge-tracking/`.

[5] Barry W. Boehm. Software engineering economics. *Software Engineering, IEEE Transactions on*, SE-10(1):4 –21, jan. 1984.

[6] Barry W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.

[7] I.T. Bowman, R.C. Holt, and N.V. Brewster. Linux as a case study: Its extracted software architecture. In *Proceedings of the 21st international conference on Software engineering*, pages 555–563. ACM, 1999.

[8] Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster. Linux as a case study: its extracted software architecture. In *Proceedings of the*

21st international conference on Software engineering, ICSE '99, pages 555–563, New York, NY, USA, 1999. ACM.

[9] Pär Carlshamre. Release planning in market-driven software product development: Provoking an understanding. *Requirements Engineering*, 7:139–151, 2002.

[10] M. Cataldo and J.D. Herbsleb. Factors leading to integration failures in global feature-oriented development: an empirical analysis. In *Proceeding of the 33rd international conference on Software engineering*, pages 161–170. ACM, 2011.

[11] M.E. Conway. How do committees invent? *Datamation*, 14(4):28–31, 1968.

[12] J. Corbet. Waiting for Emacs 22. `http://lwn.net/Articles/234593/`, 2007.

[13] K. Crowston and J. Howison. The social structure of free and open source software development. *First Monday*, 10(2), 2005.

[14] M. Diaz and J. Sligo. How software process improvement helped motorola. *Software, IEEE*, 14(5):75–81, sep/oct 1997.

[15] Tadashi Dohi, Yasuhiko Nishio, and Shunji Osaki. Optimal software release scheduling based on artificial neural networks. *Annals of Software Engineering*, 8:167–185, 1999.

[16] Justin R. Erenkrantz. Release Management Within Open Source Projects. In *Proceedings of the ICSE 3rd Workshop on Open Source Software Engineering*, May 2003.

[17] Roy T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000.

[18] Eliyahu M. Goldratt. *Essays on the Theory of Constraints*. North River Press, 1998.

[19] R.E. Grinter, J.D. Herbsleb, and D.E. Perry. The geography of coordination: dealing with distance in r&d work. In *Proceedings of the international ACM SIGGROUP conference on Supporting group work*, pages 306–315. ACM, 1999.

[20] R.S. Hall, D. Heimbigner, André van der Hoek, and Alexander L. Wolf. The Software Dock: A Distributed, Agent-based Software Deployment System. Technical Report CU-CS-832-97, University of Colorado, Dept. of Computer Science, February 1997.

[21] Donald E. Harter, Mayuram S. Krishnan, and Sandra A. Slaughter. Effects of process maturity on quality, cycle time, and effort in software product development. *Management Science*, 46(4):pp. 451–466, 2000.

[22] Jim Highsmith. *Agile software development ecosystems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[23] R.H. Hou, S.Y. Kuo, and Y.P. Chang. Optimal release times for software systems with scheduled delivery time based on the HGDM. *IEEE Transactions on Computers*, 46(2):216–221, 1997.

[24] C.Y. Huang and M.R. Lyu. Optimal release time for software systems considering cost, testing-effort, and test efficiency. *IEEE transactions on reliability*, 54(4):583–591, 2005.

[25] Yiu-Wing Leung. Optimum software release time with a given cost budget. *Journal of Systems and Software*, 17(3):233 – 242, 1992.

[26] K. D. Levin and O. Yadid. Optimal release time of improved versions of software packages. *Information and Software Technology*, 32(1):65–70, 1990.

[27] A.D. MacCormack, J. Rusnak, C.Y. Baldwin, and Harvard Business School. Division of Research. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science*, 52(7):1015, 2006.

[28] Martin Michlmayr. Quality Improvement in Volunteer Free Software Projects: Exploring the Impact of Release Management. In *Proceedings of the First International Conference on Open Source Systems*, pages 309–10, 2005.

[29] Martin Michlmayr. *Quality Improvement in Volunteer Free Software Projects: Exploring the Impact of Release Management.* PhD thesis,

University of Cambridge, 2007.

[30] Martin Michlmayr, F. Hunt, and D. Probert. Release Management in Free Software Projects: Practices and Problems. *International Federation for Information Processing*, 234:295, 2007.

[31] Kazu Okumoto and Amrit L. Goel. Optimum release time for software systems based on reliability and cost criteria. *Journal of Systems and Software*, 1:315–318, 1980.

[32] Lee Osterweil. Modeling processes to effectively reason about their properties. In *Proceedings of the ProSim '03 Workshop*, 2003.

[33] Lee J. Osterweil. Unifying microprocess and macroprocess research. In *Unifying the Software Process Spectrum*, Lecture Notes in Computer Science, pages 68–74. Springer Berlin / Heidelberg, 2006.

[34] J.W. Paulson, G. Succi, and A. Eberlein. An empirical study of open-source and closed-source software products. *Software Engineering, IEEE Transactions on*, 30(4):246–256, 2004.

[35] D.E. Perry, H.P. Siy, and L.G. Votta. Parallel changes in large-scale software development: an observational case study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 10(3):308–337, 2001.

[36] Dewayne E. Perry. Enactment control in interact/intermediate. *Software Process Technology*, pages 107–113, 1994.

[37] Dewayne E. Perry and C. Stieg. Software faults in evolving a large, real-time system: a case study. *Software Engineering–ESEC'93*, pages 48–67, 1993.

[38] Eric S. Raymond. The Cathedral and the Bazaar. *Knowledge, Technology, and Policy*, 12(3):23–49, 1999.

[39] J.W. Reeves. What is software design. *C++ Journal*, 2(2), 1992.

[40] R. Rosenthal and R. Rosnow. *Essentials of behavioural research.* McGraw, 1991.

[41] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of the 9th International Conference on Software Engineering*, pages 328–338. IEEE Computer Society Press Los Alamitos, CA, USA, 1987.

[42] S. Sawyer. Packaged software: implications of the differences from custom approaches to software development. *European Journal of Information Systems*, 9(1):47–58, 2000.

[43] John Stark. Product lifecycle management. In *Product Lifecycle Management*, Decision Engineering, pages 1–16. Springer London, 2011.

[44] Nancy A. Staudenmayer. *Managing multiple interdependencies in large scale software development projects.* PhD thesis, Massachusetts Institute of Technology, 1997.

[45] André van der Hoek, R. S. Hall, D. Heimbigner, and Alexancer. L. Wolf. Software release management. *ACM SIGSOFT Software Engineering Notes*, 22(6):159–175, 1997.

[46] Michiel van Genuchten. Why is software late? an empirical study of reasons for delay in software development. *IEEE Transactions on Software Engineering*, 17(6):582–590, 1991.

[47] A. Wise, A. G. Cass, B. S. Lerner, E. K. McCall, Lee J. Osterweil, and S. M. Sutton Jr. Using little-jil to coordinate agents in software engineering. In *Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering*, pages 155–163. IEEE, 2000.

[48] Alexander L. Wolf and David S. Rosenblum. A Study in Software Process Data Capture and Analysis. In *Proceedings of the Second International Conference on the Software Process*, pages 115–124, 1993.

[49] Hyrum K. Wright, Miryung Kim, and Dewayne E. Perry. Validity Concerns in Software Engineering Research. In *Proceedings of the Workshop on the Future of Software Engineering Research*, November 2010.

[50] Hyrum K. Wright and Dewayne E. Perry. Subversion 1.5: A Case Study in Open Source Release Mismanagement. In *Proceedings of the ICSE 2nd Emerging Trends in FLOSS Research and Development Workshop*, May 2009.

[51] S. Yamada and S. Osaki. Cost-reliability optimal release policies for software systems. *Reliability, IEEE Transactions on*, 34(5):422–424, 2009.

[52] R. Yin. *Case study research: Design and methods.* Sage Pubns, 2008.