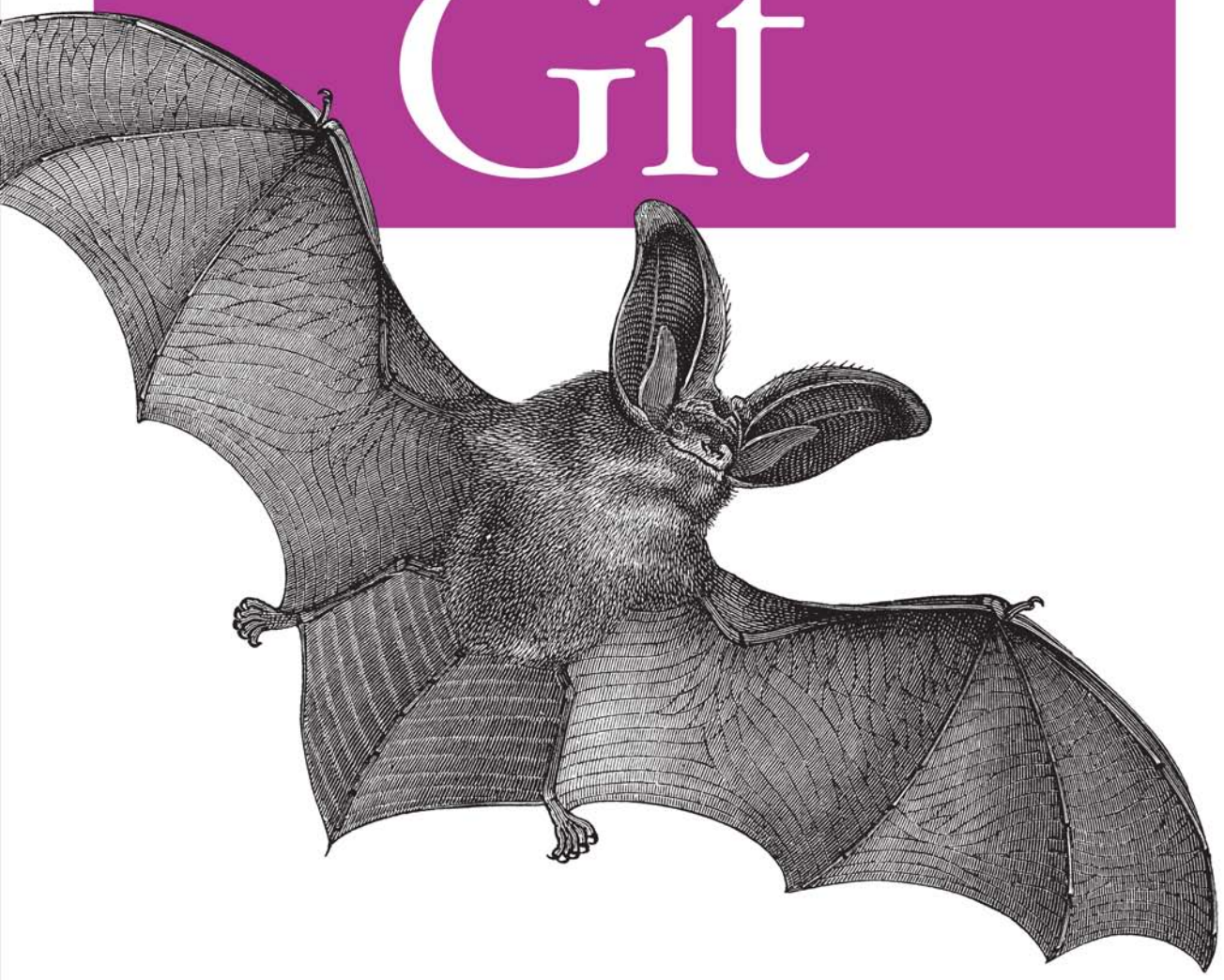
Powerful Techniques for Centralized and Distributed Project Management

Version Control with Git



O'REILLY®

Jon Loeliger

Software Engineering

O'REILLY®

Version Control with Git

Version Control with Git takes you step-by-step through ways to track, merge, and manage software projects, using this highly flexible, open source version control system.

Git permits a virtually infinite variety of methods for development and collaboration. Created by Linus Torvalds to manage development of the Linux kernel, it's become the principal tool for distributed version control. But Git's flexibility also means that some users don't understand how to use it to their best advantage. *Version Control with Git* offers tutorials on the most effective ways to use it, as well as friendly yet rigorous advice to help you navigate Git's many functions.

With this book, you will:

- Learn how to use Git in several real-world development environments
- Gain insight into Git's common-use cases, initial tasks, and basic functions
- Understand how to use Git for both centralized and distributed version control
- Use Git to manage patches, diffs, merges, and conflicts
- Acquire advanced techniques such as rebasing, hooks, and ways to handle submodules (subprojects)
- Learn how to use Git with Subversion

Git has earned the respect of developers around the world. Find out how you can benefit from this amazing tool with *Version Control with Git.*

"This is a book I'm going to keep within easy reach."

—Don Marti, editor, correspondent, and conference chair

Jon Loeliger is a freelance software engineer who contributes to open source projects such as Linux, U-Boot, and Git. He has given tutorial presentations on Git at many conferences, including Linux World, and has written several papers on Git for *Linux Magazine.*

oreilly.com

US $34.99 **CAN $43.99**

ISBN: 978-0-596-52012-0

9 780596 520120 53499

Version Control with Git

Version Control with Git

Jon Loeliger

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

Version Control with Git
by Jon Loeliger

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Andy Oram
Production Editor: Loranah Dimant
Proofreader: Katie Nopper DePasquale
Production Services: Newgen North America
Indexer: Fred Brown
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Printing History:
May 2009: First Edition.

ISBN: 978-0-596-52012-0

[M]

1242320486

Table of Contents

Preface

Audience

While some familiarity with revision control systems will be good background material, a reader who is not familiar with any other system will still be able to learn enough about basic Git operations to be productive in a short while. More advanced readers should be able to gain insight into some of Git's internal design and thus master some of its more powerful techniques.

The main intended audience for this book should be familiar and comfortable with the Unix shell, basic shell commands, and general programming concepts.

Assumed Framework

Almost all examples and discussions in this book assume the reader has a Unix-like system with a command-line interface. The author developed these examples on Debian and Ubuntu Linux environments. The examples should work under other environments, such as Mac OS X or Solaris, but the reader can expect slight variations.

A few examples require root access on machines where system operations are needed. Naturally, in such situations you should have a clear understanding of the responsibilities of root access.

Book Layout and Omissions

This book is organized as a progressive series of topics, each designed to build upon concepts introduced earlier. The first 10 chapters focus on concepts and operations that pertain to one repository. They form the foundation for more complex operations on multiple repositories covered in the final six chapters.

If you already have Git installed or have even used it briefly, you may not need the introductory and installation information in the first two chapters, nor even the quick tour presented in the third chapter.

The concepts covered in Chapter 4 are essential for a firm grasp on Git's object model. They set the stage and prepare the reader for a clearer understanding of many of Git's more complex operations.

Chapters 5 through 10 cover various topics in more detail. Chapter 5 describes the index and file management. Chapters 6 and 10 discuss the fundamentals of making commits and working with them to form a solid line of development. Chapter 7 introduces branches so that you may manipulate several different lines of development from your one local repository. Chapter 8 explains how Git derives and presents "diffs."

Git provides a rich and powerful ability to join different branches of development. The basics of branch merging and resolving merge conflicts is covered in Chapter 9. A key insight into Git's model is the realization that all merging performed by Git happens in your local repository in the context of your current working directory.

The fundamentals of naming and exchanging data with another, remote repository are covered in Chapter 11. Once the basics of merging have been mastered, interacting with multiple repositories is shown to be a simple combination of an exchange step plus a merge step. The exchange step is the new concept covered in this chapter; the merge step is covered in Chapter 9.

Chapter 12 provides a more philosophical and abstract coverage of repository management "in the large." It also establishes a context for Chapter 13 to cover patch handling when direct exchange of repository information isn't possible using Git's native transfer protocols.

The remaining three chapters cover advanced topics: the use of hooks, combining projects and multiple repositories into a superproject, and interacting with Subversion repositories.

Git is still evolving rapidly because there is an active developer base. It is not that Git isn't mature enough to be used for development; rather, ongoing refinements and user interface issues are being enhanced regularly. Even as this book was being written, Git evolved. Apologies if I was unable to keep up accurately.

I do not give the command `gitk` the complete coverage that it deserves. If you like graphical representations of the history within a repository, you should explore `gitk`. Other history visualization tools exist as well, but they are not covered here either. Nor am I able to cover a rapidly evolving and growing host of other Git-related tools. I'm not even able to cover all of Git's own core commands and options thoroughly in this book. Again, my apologies.

Perhaps, though, enough pointers, tips, and direction can be found here to inspire readers to do some of their own research and exploration!

Conventions Used in This Book

The following typographical conventions are used in this book:

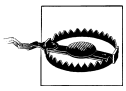
Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, useful hint, or a general note.



This icon indicates a warning or caution.

Furthermore, you should be familiar with basic shell commands to manipulate files and directories. Many examples will contain commands such as these to add or remove directories, copy files, or create simple files:

```
$ cp file.txt copy-of-file.txt
$ mkdir newdirectory
$ rm file
$ rmdir somedir
$ echo "Test line" > file
$ echo "Another line" >> file
```

Commands that need to be executed with root permissions appear with a `sudo` operation:

```
# Install the Git core package

$ sudo apt-get install git-core
```

How you edit files or effect changes within your working directory is pretty much up to you. You should be familiar with a text editor. In this book, I'll denote the process of editing a file by either a direct comment or a pseudocommand:

```
# edit file.c to have some new text

$ edit index.html
```

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Version Control with Git*, by Jon Loeliger. Copyright 2009 Jon Loeliger, 978-0-596-52012-0."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

When you see a Safari® Books Online icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at *http://my.safaribooksonline.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/9780596520120/

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our website at:

http://www.oreilly.com

Acknowledgments

This work would not have been possible without the help of many other people. I'd like to thank Avery Pennarun for contributing substantial material to Chapters 14, 15, and 16. He also contributed some material to Chapters 4 and 9. His help was appreciated. I'd like to publicly thank those who took time to review the book at various stages: Robert P. J. Day, Alan Hasty, Paul Jimenez, Barton Massey, Tom Rix, Jamey Sharp, Sarah Sharp, Larry Streepy, Andy Wilcox, and Andy Wingo.

Also, I'd like to thank my wife, Rhonda, and daughters, Brandi and Heather, who provided moral support, gentle nudging, Pinot Noir, and the occasional grammar tip. And thanks to Mylo, my long-haired dachshund, who spent the entire writing process curled up lovingly in my lap. I'd like to add a special thanks to K.C. Dignan, who supplied enough moral support and double-stick butt tape to keep my behind in my chair long enough to finish this book!

Finally, I would like to thank the staff at O'Reilly as well as my editors, Andy Oram and Martin Streicher.

CHAPTER 1

Introduction

Background

No cautious, creative person starts a project nowadays without a back-up strategy. Because data is ephemeral and can be lost easily—through an errant code change or a catastrophic disk crash, say—it is wise to maintain a living archive of all work.

For text and code projects, the back-up strategy typically includes version control, or tracking and managing revisions. Each developer can make several revisions per day, and the ever-increasing corpus serves simultaneously as repository, project narrative, communication medium, and team and product management tool. Given its pivotal role, version control is most effective when tailored to the working habits and goals of the project team.

A tool that manages and tracks different versions of software or other content is referred to generically as a version control system (VCS), a source code manager (SCM), a revision control system (RCS), and with several other permutations of the words "revision," "version," "code," "content," "control," "management," and "system." Although the authors and users of each tool might debate esoterics, each system addresses the same issues: develop and maintain a repository of content, provide access to historical editions of each datum, and record all changes in a log. In this book, the term *version control system* (VCS) is used to refer generically to any form of revision control system.

This book covers Git, a particularly powerful, flexible, and low-overhead version control tool that makes collaborative development a pleasure. Git was invented by Linus Torvalds to support the development of the Linux Kernel, but it has since proven valuable to a wide range of projects.

The Birth of Git

Often, when there is discord between a tool and a project, the developers simply create a new tool. Indeed, in the world of software, the temptation to create new tools can be deceptively easy and inviting. In the face of many existing version control systems, the decision to create another shouldn't be made casually. However, given a critical need, a bit of insight, and a healthy dose of motivation, forging a new tool can be exactly the right course.

Git, affectionately termed "the information manager from hell" by its creator is such a tool. Although the precise circumstances and timing of its genesis are shrouded in political wrangling within the Linux Kernel community, there is no doubt that what came from that fire is a well-engineered version control system capable of supporting worldwide development of software on a large scale.

Prior to Git, the Linux Kernel was developed using the commercial BitKeeper VCS, which provided sophisticated operations not available in then-current, free software version control systems such as RCS and CVS. However, when the company that owned BitKeeper placed additional restrictions on its "free as in beer" version in the spring of 2005, the Linux community realized that BitKeeper was no longer a viable solution.

Linus looked for alternatives. Eschewing commercial solutions, he studied the free software packages but found the same limitations and flaws that led him to reject them previously. What was wrong with the existing VCS systems? What were the elusive missing features or characteristics that Linus wanted and couldn't find?

Facilitate distributed development
: There are many facets to "distributed development," and Linus wanted a new VCS that would cover most of them. It had to allow parallel as well as independent and simultaneous development in private repositories without the need for constant synchronization with a central repository, which could form a development bottleneck. It had to allow multiple developers in multiple locations even if some of them were offline temporarily.

Scale to handle thousands of developers
: It isn't enough just to have a distributed development model. Linus knew that thousands of developers contribute to each Linux release, so any new VCS had to handle a very large number of developers, whether they were working on the same or on different parts of a common project. And the new VCS had to be able to integrate all of their work reliably.

Perform quickly and efficiently
: Linus was determined to ensure that a new VCS was fast and efficient. In order to support the sheer volume of update operations that would be made on the Linux Kernel alone, he knew that both individual update operations and network transfer operations would have to be very fast. To save space and thus transfer time, compression and "delta" techniques would be needed. Using a distributed model

instead of a centralized model also ensured that network latency would not hinder daily development.

Maintain integrity and trust
: Because Git is a distributed revision control system, it is vital to obtain absolute assurance that data integrity is maintained and is not somehow being altered. How do you know the data hasn't been altered in transition from one developer to the next, or from one repository to the next? For that matter, how do you know that the data in a Git repository is even what it purports to be?

  Git uses a common cryptographic hash function, called *Secure Hash Function* (SHA1), to name and identify objects within its database. Although perhaps not absolute, in practice it has proven to be solid enough to ensure integrity and trust for all of Git's distributed repositories.

Enforce accountability
: One of the key aspects of a version control system is knowing who changed files, and if at all possible, why. Git enforces a change log on every commit that changes a file. The information stored in that change log is left up to the developer, project requirements, management, convention, etc. Git ensures that changes will not happen mysteriously to files under version control because there is an accountability trail for all changes.

Immutability
: Git's repository database contains data objects that are *immutable*. That is, once they have been created and placed in the database, they cannot be modified. They can be recreated differently, of course, but the original data cannot be altered without consequences. The design of the Git database means that the entire history stored within the version control database is also immutable. Using immutable objects has several advantages, including very quick comparison for equality.

Atomic transactions
: With atomic transactions, a number of different but related changes are performed either all together or not at all. This property ensures that the version control database is not left in a partially changed (and hence possibly corrupted) state while an update or commit is happening. Git implements atomic transactions by recording complete, discrete repository states that cannot be broken down into individual or smaller state changes.

Support and encourage branched development
: Almost all VCSs can name different genealogies of development within a single project. For instance, one sequence of code changes could be called "development" while another is referred to as "test." Each version control system can also split a single line of development into multiple lines and then unify, or merge, the disparate threads. As with most VCSs, Git calls a line of development a *branch* and assigns each branch a name.

  Along with branching comes merging. Just as Linus wanted easy branching to foster alternate lines of development, he also wanted to facilitate easy merging of

those branches. Because branch merging has often been a painful and difficult operation in version control systems, it would be essential to support clean, fast, easy merging.

Complete repositories
: So that individual developers needn't query a centralized repository server for historical revision information, it was essential that each repository have a complete copy of all historical revisions of every file.

A clean internal design
: Even though end users might not be concerned about a clean internal design, it was important to Linus and ultimately to other Git developers as well. Git's object model has simple structures that capture fundamental concepts for raw data, directory structure, recording changes, etc. Coupling the object model with a globally unique identifier technique allowed a very clean data model that could be managed in a distributed development environment.

Be free, as in freedom
: 'Nuff said.

Given a clean slate to create a new VCS, many talented software engineers collaborated and Git was born. Necessity was the mother of invention again!

Precedents

The complete history of version control systems is beyond the scope of this book. However, there are several landmark, innovative systems that set the stage for or directly led to the development of Git. (This section is selective, hoping to record when new features were introduced or became popular within the free software community.)

The Source Code Control System (SCCS) was one of the original systems on Unix and was developed by M. J. Rochkind in the very early 1970s.[*] This is arguably the first VCS available on any Unix system.

The central store that SCCS provided was called a repository, and that fundamental concept remains pertinent to this day. SCCS also provided a simple locking model to serialize development. If a developer needed files to run and test a program, she would check them out unlocked. However, in order to edit a file, she had to check it out with a lock (a convention enforced through the Unix filesystem). When finished, she would check the file back into the repository and unlock it.

In the early 1980s, Walter Tichy introduced the Revision Control System (RCS)[†] RCS introduced both forward and reverse delta concepts for efficient storage of different file revisions.

[*] "The Source Code Control System," *IEEE Transactions on Software Engineering* 1(4) (1975): 364–370.

[†] "RCS—A System for Version Control," *Software Practice and Experience* 15 (7) (July 1985): 637–654.

The Concurrent Version System (CVS), designed and originally implemented by Dick Grune in 1986 and then crafted anew some four years later by Berliner et al., extended and modified the RCS model with great success. CVS became very popular and was the de facto standard within the open source community for many years. CVS provided several advances over RCS, including distributed development and repository-wide change sets for entire "modules."

Furthermore, CVS introduced a new paradigm for the lock. Whereas earlier systems required a developer to lock each file before changing it and thus forced one developer to wait for another in serial fashion, CVS gave each developer write permission in his private working copy. Thus, changes by different developers could be merged automatically by CVS unless two developers tried to change the same line. In that case, the conflict was flagged and the developers were left to work out the solution. The new rules for the lock allowed different developers to write code concurrently.

As often occurs, perceived shortcomings and faults in CVS eventually led to a new version control system. Subversion (SVN), introduced around 2001, quickly became popular within the free software community. Unlike CVS, SVN committed changes atomically and had significantly better support for branches.

BitKeeper and Mercurial were radical departures from all the aforementioned solutions. Each eliminated the central repository; instead, the store was distributed, providing each developer with his own shareable copy. Git is derived from this peer-to-peer model.

Finally, Mercurial and Monotone contrived a hash fingerprint to uniquely identify a file's content. The name assigned to the file is a moniker and convenient handle for the user and nothing more. Git features this notion as well. Internally, the Git identifier is based on the file's contents, a concept known as a content-addressable file store. The concept is not new. (For example, see "The Venti Filesystem," (Plan 9), Bell Labs, *http://www.usenix.org/events/fast02/quinlan/quinlan_html/index.html*.) Git borrowed the idea immediately from Monotone, according to Linus.‡ Mercurial was implementing the concept simultaneously with Git.

Time Line

With the stage set, a bit of external impetus, and a dire VCS crisis imminent, Git sprang to life in April 2005.

Git became self-hosted on April 7 with this commit:

```
commit e83c5163316f89bfbde7d9ab23ca2e25604af29
Author: Linus Torvalds <torvalds@ppc970.osdl.org>
Date:   Thu Apr 7 15:13:13 2005 -0700
```

‡ Private email.

```
    Initial revision of "git", the information manager from hell
```

Shortly thereafter, the first Linux commit was made:

```
commit 1da177e4c3f41524e886b7f1b8a0c1fc7321cac2
Author: Linus Torvalds <torvalds@ppc970.osdl.org>
Date:   Sat Apr 16 15:20:36 2005 -0700

    Linux-2.6.12-rc2

    Initial git repository build. I'm not bothering with the full history,
    even though we have it. We can create a separate "historical" git
    archive of that later if we want to, and in the meantime it's about
    3.2GB when imported into git - space that would just make the early
    git days unnecessarily complicated, when we don't have a lot of good
    infrastructure for it.

    Let it rip!
```

That one commit introduced the bulk of the entire Linux Kernel into a Git repository.[§] It consisted of the following:

```
17291 files changed, 6718755 insertions(+), 0 deletions(-)
```

Yes, that's an introduction of 6.7 million lines of code!

It was just three minutes later when the first patch using Git was applied to the kernel. Convinced that it was working, Linus announced it on April 20, 2005 to the Linux Kernel Mailing List.

Knowing full well that he wanted to return to the task of developing the kernel, Linus handed the maintenance of the Git source code to Junio Hamano on July 25, 2005, announcing that "Junio was the obvious choice."

About two months later, version 2.6.12 of the Linux Kernel was released using Git.

What's in a Name?

Linus himself rationalizes the name "Git" by claiming "I'm an egotistical bastard, and I name all my projects after myself. First Linux, now git."[‖] Granted, the name "Linux" for the kernel was sort of a hybrid of Linus and Minix. The irony of using a British term for a silly or worthless person was not missed either.

In the meantime, others had suggested some alternative, more palatable interpretations: the Global Information Tracker seems to be the most popular.

[§] See *http://kerneltrap.org/node/13996* for a starting point on how the old BitKeeper logs were imported into a Git repository for older history (pre-2.5).

[‖] See *http://www.infoworld.com/article/05/04/19/HNtorvaldswork_1.html*.