Community Experience Distilled

Bazaar Version Control

A fast-paced practical guide to version control using Bazaar

Janos Gyerik

www.allitebooks.com

Bazaar Version Control

A fast-paced practical guide to version control using Bazaar

Janos Gyerik

BIRMINGHAM - MUMBAI

www.allitebooks.com

Bazaar Version Control

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: June 2013

Production Reference: 1300513

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84951-356-2

www.packtpub.com

Cover Image by Andrew Caudwell (acaudwell@gmail.com)

www.allitebooks.com

Credits

Author
Janos Gyerik

Reviewers
Alexander Belchenko
John Arbash Meinel
Yavor Nikolov

Acquisition Editors
Mary Jasmine Nadar
Llewellyn Rozario

Lead Technical Editor
Ankita Shashi

Technical Editors
Jalasha D'costa
Amit Ramadas
Lubna Shaikh

Project Coordinator
Amey Sawant

Proofreaders
Kate Elizabeth
Clyde Jenkins

Indexer
Monica Ajmera Mehta

Production Coordinator
Conidon Miranda

Cover Work
Conidon Miranda

www.allitebooks.com

About the Author

Janos Gyerik is a Software Engineer living in Paris, France. He has been using Bazaar since its early releases to manage his personal projects, some of which are open source and available on Launchpad (`https://launchpad.net/~janos-gyerik`). Janos is passionate about Bazaar, and although he embraces other version control systems as well, he wouldn't miss a chance to uphold Bazaar's values over competitive solutions. Janos spends most of his free time on various personal projects, and he is always up to something, which you can read about on his blog at `http://janosgyerik.com/`.

I would like to thank my wife for putting up with my late night writing sessions. I also give deep thanks and gratitude to my brother, Matyas Fodor, and my friends, Hugues Merlen, Alain Vizzini, Ivan Zimine, and Pierre-Jean Baraud, whose critical comments and support has helped me greatly in writing and improving the quality of this book.

I also would like to thank the reviewers Yavor Nikolov, John Meinel, and Alexander Belchenko for their criticism and support, it was a real pleasure working together. Finally, I thank Packt Publishing for this great opportunity.

www.allitebooks.com

About the Reviewers

Alexander Belchenko is a software developer from Ukraine. He worked on hardware and software designs of embedded systems and radio-electronic devices as a Radio Engineer and Software Developer. In his free time, Alexander contributes to open source projects. In 2005, he started contributing to the Bazaar VCS project, and later worked on GUI tools for Bazaar VCS.

John Arbash Meinel is a software developer currently living in Dubai, United Arab Emirates. He was one of the primary developers of Bazaar, and is currently working on cloud technologies. He was employed by Canonical Ltd.

I would like to thank Martin Pool for bringing the vision for such a wonderful version control system, and my wife and son for bringing a balance to my life outside work.

Yavor Nikolov is a software professional living in Sofia, Bulgaria. His professional background is mostly in Oracle Database technologies and data warehousing, and being involved in software development, database administration, tweaking server OS, and technical consulting.

Yavor's interests are in bettering everything in the software/knowledge world—from personal level to team, products, and organizations as a whole. He is trying to bring innovation and good practices in tools, technologies and infrastructure, process of work, project management, collaboration and learning culture.

www.allitebooks.com

As a proponent of Kanban, Lean, Agile, Scrum methods, approaches, and related practices, Yavor has been actively involved in the local communities, which have emerged around these topics (most notably Scrum Bulgaria—`http://scrumbulgaria.org/`).

Yavor often uses open source software. He uses Linux as his main OS on his computer at work and at home. He's also been contributing to a few open source projects, most notably DbFit—`http://benilovj.github.io/dbfit`, and pbzip2—`http://compression.ca/pbzip2`.

Yavor discovered Bazaar and Launchpad in his way while trying to find an online collaboration platform and source control repository for the previously mentioned pbzip2 project. He loved the power and flexibility of Bazaar and since then has been using it in some other projects and personal work.

When not at work, Yavor loves spending time with nature and is often found hiking in nearby mountains.

> I would like to thank the author, Janos Gyerik, and Packt Publishing for their effort in making this great book. Thanks for involving me in its review—particular thanks to Amey Sawant and Leena Purkait of Packt Publishing for their professional attitude. I've been glad to help and to be part of this project!

www.allitebooks.com

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

http://PacktLib.PacktPub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

www.allitebooks.com

www.allitebooks.com

Table of Contents

Preface **1**

Chapter 1: Getting Started **7**

Version control systems **7**
Reverting a project to a previous state 8
Viewing the log of changes 8
Viewing the differences between revisions 9
Branching and merging 10
Acronyms related to version control 12
Centralized version control systems (CVCS) **13**
Distributed version control systems (DVCS) **14**
What is Bazaar? **16**
Installing Bazaar and its plugins **17**
GNU/Linux 17
Ubuntu, Debian, and derivatives 18
Red Hat, Fedora, CentOS, and derivatives 18
openSUSE and derivatives 18
Installing Bazaar using pip 18
Other installation methods 19
Windows 19
Mac OS X 20
Bazaar in a shared hosting environment 21
Interacting with Bazaar **22**
Using the command-line interface 22
Using the graphical user interface 23
Using the two user interfaces together 26
Upgrading Bazaar to the latest version **26**
Uninstalling Bazaar **26**
Getting help **27**
Summary **28**

www.allitebooks.com

Chapter 2: Diving into Bazaar 29
Understanding the core concepts 29
Revision 30
Repository 31
Branch 32
Working tree 32
Putting the concepts together 34
Storing Bazaar's data in the filesystem 34
Introducing the user interfaces 35
Using the command-line interface (CLI) 35
Using Bazaar Explorer 36
Configuring Bazaar 36
Configuring the author information 36
Configuring the default editor 37
Other configuration options 38
Performing the basic version control operations 38
Putting a directory under version control 39
Using the command line 39
Using Bazaar Explorer 40
Checking the status of files and directories 42
Using the command line 43
Using Bazaar Explorer 44
Adding files to version control 45
Using the command line 45
Using Bazaar Explorer 46
Recording a new revision 49
Using the command line 49
Using Bazaar Explorer 50
Ignoring files 53
Using the command line 54
Using Bazaar Explorer 54
Checkpoint 56
Deleting files 56
Using the command line 57
Using Bazaar Explorer 57
Undoing changes 58
Using the command line 58
Using Bazaar Explorer 58
Editing files 60
Using the command line 60
Using Bazaar Explorer 60
Viewing differences in changed files 61
Using the command line 62

Using Bazaar Explorer 64
Checkpoint 65
Renaming or moving files 66
Using the command line 66
Using Bazaar Explorer 68
Checkpoint 68
Viewing the revision history 68
Using the command line 68
Using Bazaar Explorer 70
Restoring files from a past revision 71
Using the command line 71
Using Bazaar Explorer 72
Putting it all together 72
Making different kinds of changes 72
Understanding the backup files created by Bazaar 74
Understanding the .bzr directory 75
How often to commit? 75
Beyond the basics 76
Mastering the command line 76
Common flags 76
Common behavior in all the commands 76
Using shorter aliases of commands 77
Quick reference card 77
Using tags 77
Specifying revisions 78
Specifying a single revision 78
Specifying a range of revisions 79
Viewing differences between any two revisions 80
Viewing differences between any revision and the working tree 80
Viewing differences between any two revisions 81
Viewing differences going from one revision to the next 82
Cloning your project 82
Summary 83
Chapter 3: Using Branches 85
What is a branch? 86
A single branch with a linear history 86
Multiple diverged branches 86
Branches with non-linear history 87
Unrelated branches 88
What can you do with branches? 88
Creating branches 88
Comparing branches 89
Merging branches 89
Mirroring branches 90

Why use more than one branch? 90
Separating the development of new features 91
Switching between tasks 92
Experimenting with different approaches 93
Maintaining multiple versions 94
Understanding core terms and concepts 94
trunk, master, and mainline 94
The tip of a branch 94
Source and target branches 95
Parent branch 95
Diverged branches and the base revision 95
Storing branch data 96
Using a shared repository 96
Using the command line 98
Using Bazaar Explorer 98
Basic branching and merging 99
Getting the example project 99
Using the command line 99
Using Bazaar Explorer 100
Creating a feature branch 101
Using the command line 101
Using Bazaar Explorer 101
Working on a branch 102
Starting another branch 102
Merging the bugfix branch 103
Using the command line 104
Using Bazaar Explorer 104
Viewing merged revisions in the log 106
Using the command line 106
Using Bazaar Explorer 107
Using the branch command 108
Creating branches based on an older revision 109
Using the command line 109
Using Bazaar Explorer 109
Viewing basic branch information 109
Comparing branches 110
Using the command line 110
Viewing missing revisions between branches 111
Viewing the differences between branches 112
Using Bazaar Explorer 114
Viewing the tree of branches 114
Viewing missing revisions between branches 115
Viewing the differences between branches 116

Merging branches **116**
Performing a three-way merge 117
Completing the merge 118
Committing the merge 118
Aborting the merge 119
Resolving conflicts 119
Resolving text conflicts 120
Resolving content conflicts 124
Redoing the merge 125
Resolving other types of conflicts 126
Merging a subset of revisions 126
Merging up to a specific revision 126
Merging a range of revisions 127
Cherry-picking 128
Understanding revision numbers 128
Merging from multiple branches 130
Mirroring branches **130**
Mirroring from another branch 131
Mirroring from the current branch 132
Summary **134**
Chapter 4: Using Bazaar in a Small Team **135**
Collaborating with others **136**
Working with remote branches 136
Implementing simple workflows 137
Sharing branches over the network **138**
Specifying remote branches 138
Using URL parameters 139
Using remote branches through a proxy 139
Sharing branches using a distributed filesystem 140
Sharing branches over SSH 142
Using individual SSH accounts 142
Using individual SSH accounts with SFTP 143
Using a shared restricted SSH account 143
Using SSH host aliases 145
Using a different SSH client 145
Sharing branches using bzr serve 145
Sharing branches using inetd 146
Sharing branches over HTTP or HTTPS 147
Working with remote branches **147**
Working with remote branches directly 148
Using local mirror branches 148
Creating a local mirror 150
Using a shared repository 150
Updating a local mirror 150

Using remote mirror branches 151
Creating a remote mirror 151
Using a shared repository 152
Updating a remote mirror 152
Using branches without a working tree 152
Creating a local branch without a working tree 152
Creating or removing the working tree 153
Reconfiguring working trees in a shared repository 154
Creating remote branches without a working tree 154
Slicing and dicing branches 155
Implementing simple workflows 156
Using independent personal branches 157
Merging from branches repeatedly 158
Handling criss-cross merges 159
Viewing the history from different perspectives 160
Using feature branches and a common trunk 161
Merging into a common remote trunk 162
Merging feature branches in lock-step 163
Doing "code reviews" 164
Summary 164
Chapter 5: Working with Bazaar in Centralized Mode 165
The centralized mode 166
Core operations 166
The centralized workflow 167
Checkout from the central branch 167
Making changes 168
Committing changes 168
Updating from the server 169
Handling conflicts during update 170
Advantages 170
Easy to understand 170
Easy to synchronize efforts 171
Widely used 171
Disadvantages 171
Single point of failure 171
Administrative overhead of access control 171
The update operation is not safe 172
Unrelated changes interleaved in the revision history 172
Using Bazaar in centralized mode 172
Bound branches 172
Creating a checkout 173
Using the command line 173
Using Bazaar Explorer 174
Updating a checkout 176
Using the command line 177

Using Bazaar Explorer 177
Visiting an older revision 178
Committing a new revision 178
Practical tips when working in centralized mode 179
Working with bound branches 180
Unbinding from the master branch 180
Binding to a branch 181
Using local commits 182
Working with multiple branches 184
Setting up a central server 184
Using an SSH server 185
Using the smart server over SSH 185
Using individual SSH accounts 186
Using a shared restricted SSH account 186
Using SFTP 188
Using bzr serve directly 188
Using bzr serve over inetd 189
Creating branches on the central server 189
Creating a shared repository without working trees 190
Reconfiguring a shared repository to not use working trees 190
Removing an existing working tree 191
Creating branches on the server without a working tree 191
Practical use cases 191
Working on branches using multiple computers 192
Synchronizing backup branches 193
Summary 194
Chapter 6: Working with Bazaar in Distributed Mode 195
The distributed mode in general 195
Collaborators work independently 197
The mainline branch is just a convention 198
Collaborators write only to their own branches 198
The distributed mode gives great flexibility 199
Encouraging feature branches 199
The revision history depends on the perspective 200
The human gatekeeper workflow 202
Overview 203
Setting guidelines to accept merge proposals 204
The role of the gatekeeper 205
Creating a merge proposal 206
Using a Bazaar hosting site 206
Sharing the branch URL with the gatekeeper 206
Sending a merge directive 207

Rejecting a merge proposal 211
Accepting a merge proposal 212
Reusing a branch 213
Commander/Lieutenant model 214
Switching from the peer-to-peer workflow 215
The automatic gatekeeper workflow 218
Patch Queue Manager (PQM) 218
Revision history graph 219
The shared mainline workflow 219
Updating the mainline using push operations 220
Updating the mainline using a new local mirror 220
Re-using an existing local mirror 221
Updating the mainline using a bound branch 222
Updating the mainline using a new checkout 222
Reusing an existing checkout 223
Choosing a distributed workflow 224
Summary 225
Chapter 7: Integrating Bazaar in CDE 227
What is a CDE? 227
Working with Launchpad 228
Creating a Launchpad account 229
Creating an account 229
Associating bzr with Launchpad 232
Testing your setup 232
Hosting personal branches 233
Uploading personal branches 234
Using personal branches 235
Deleting branches 236
Hosting a project 236
Using the Sandbox site 236
Creating a project 237
Uploading project branches 238
Viewing project branches 238
Viewing your own branches 239
Setting a focus branch 239
Using series 242
Viewing and editing branch details 242
Using merge proposals 243
Creating a merge proposal 243
Viewing and editing a merge proposal 245
Approving / rejecting a merge proposal 246
Using the e-mail interface to handle a merge proposal 247
Browsing the content of a branch 248

Using the bug tracking system 250
Reporting bugs 250
Linking commits to bugs 251
Useful tips when using Launchpad 251
Deleting or renaming a project 251
The karma system 251
Hosting private projects 251
Integrating Bazaar into Redmine 251
Integrating Bazaar into Trac 253
Enabling the plugin globally 253
Enabling the plugin for one project only 254
Browsing Bazaar branches 254
Getting help 255
Linking commits to bug trackers 256
Configuring bug trackers in Bazaar 257
Linking to public bug trackers 259
Linking to Launchpad 259
Linking to Bugzilla 259
Linking to Trac 260
Linking to other bug trackers 260
Advanced integration with bug trackers 260
Web-based repository browsing with Loggerhead 261
Installing Loggerhead 261
Running Loggerhead locally 262
Running Loggerhead in production 264
Summary 264
Chapter 8: Using the Advanced Features of Bazaar 265
Using aliases 266
Undoing commits 267
Shelving changes 269
Putting changes "on a shelf" 270
Listing and viewing shelved changes 273
Restoring shelved changes 274
Using shelves to revert partial changes in a file 275
Using shelves to commit partial changes in a file 275
Using lightweight checkouts 275
Creating a lightweight checkout 276
Converting a checkout to a lightweight checkout 277
Converting a branch to a lightweight checkout 278
Converting from a lightweight checkout 278

Re-using a working tree **278**
Setting up the example 279
Preparing to switch branches 280
Switching to another branch using core commands 280
Switching to another branch by using switch 282
Using a lightweight checkout for switching branches 283
Using stacked branches **283**
Signing revisions using GnuPG **284**
Configuring the signing key used by Bazaar 285
Setting up a sample repository 285
Verifying signatures 286
Signing existing revisions 286
Signing a range of commits 288
Signing new commits automatically 288
Configuring a hook to send an e-mail on commit **290**
Setting up the example 290
Installing the email plugin 290
Enabling commit emails 291
Testing the configuration 291
Customizing the plugin 292
Summary **293**
Chapter 9: Using Bazaar Together with Other VCS **295**
Working with other VCS in general **295**
Working with foreign branches 296
Installing plugins 296
Installing plugins in Windows or Mac OS X 297
Installing plugins in Linux 298
Installing plugins using Pip 298
Installing additional requirements 298
Understanding the protocol overhead 298
Using shared repositories 299
Limitations 299
Issues and crashes 299
Using Bazaar with Subversion **300**
Installing bzr-svn 300
Supported protocols and URL schemes 301
Using the example Subversion repository 301
Understanding branches in Subversion 302
Branching or checkout from Subversion 303

Preserving Subversion metadata 304
Preserving original revision numbers 304
Preserving versioned properties 305
Preserving revision and file IDs 305
Pulling or updating from Subversion 306
Committing to Subversion 307
Pushing to Subversion 307
Merging Subversion branches 308
Merging local branches into Subversion 309
Binding and unbinding to Subversion locations 312
Using lightweight checkouts 313
Browsing the logs 313
Limitations of bzr-svn 314
Final remarks on bzr-svn 314
Using Bazaar with Git 315
Installing bzr-git 315
Supported protocols and URL schemes 316
Using the example Git repository 316
Branching from git 317
Preserving version control metadata 318
Preserving Git revision ids 319
Preserving merged branches and revisions 320
Pulling from Git 321
Pushing to Git 322
Merging Git branches 322
Merging local branches into Git 324
Limitations of bzr-git 326
Final remarks on bzr-git 327
Migrating between version control systems 328
Installing bzr-fastimport 328
Exporting version control data 328
Exporting Subversion data 329
Exporting Git data 329
Exporting Bazaar data 330
Exporting other VCS data 331
Importing version control data 331
Querying fast-import files 332
Filtering fast-import 332
Summary 333

www.allitebooks.com

Chapter 10: Programming Bazaar 335
Using Bazaar programmatically 335
Using bzrlib outside of bzr 336
Accessing Bazaar objects 336
Accessing branch data 337
Accessing branch configuration values 338
Accessing revision history 338
Accessing the contents of a revision 339
Formatting revision info using a log format 340
More examples 341
Locating BZRLIB 342
Creating a plugin 342
Using the example plugins 343
Using the summary plugin 343
Using the customlog plugin 344
Using the appendlog plugin 344
Naming the plugin 345
Creating the plugin directory 345
Implementing the plugin 346
Writing the README file 346
Creating __init__.py 347
Setting help and documentation texts 347
Declaring the API version 348
Declaring the plugin version 348
Verifying the loaded module name 349
Registering new functionality 349
Registering a test suite 353
Performance considerations 353
Writing unit tests 354
Creating setup.py 357
Browsing existing plugins 358
Registering your plugin 359
Creating a hook 360
Hook points, hook classes, and hook types 360
Registering hooks 361
Activating hooks 362
References 362
Summary 363
Index 365

Preface

A version control system enables you to track your changes, view the history of your revisions, revert to previous states if necessary, and allows you many other very practical operations. Bazaar is such a system, and although these tasks are complicated and can be really difficult to accomplish, Bazaar makes makes all this as easy for you as possible.

I have been using Bazaar since its early days. At the time I was a happy user of Subversion. Although I could not do everything that I wanted with it, I was not looking for something better. I don't remember what compelled me to try Bazaar, but I do remember that soon after I tried it, very quickly (and very easily!) I migrated all my projects, without ever looking back.

I found my way around Bazaar little by little, mostly by reading its built-in help pages. Based on my previous experiences with version control systems, I often used operations the "hard way" at first, only to learn later that Bazaar had a much easier, much more intuitive way to accomplish the same thing. I had to unlearn many things, and again and again I was surprised by how predictable this tool was. I could guess how some complex operations would work in a situation I have never experienced before, and to my surprise Bazaar would prove me right.

Although Bazaar has excellent documentation both built-in and online, the idea behind the structure of this book is to lead you on, step by step, through more and more logically complex scenarios that you might find yourself in when working on any project. When you start using a version control tool, you will probably try it first by yourself, in a simple project you have, or something completely new. As the project shapes up, you might want to share your work with your friends or colleagues, get some feedback from them, or better yet, get actual implementations of real improvements. The idea is to not to just go over all the possible operations like a bullet-point list, but to put them in practical, realistic contexts, jam-packed with good examples. The book gradually reveals the power of Bazaar, while constantly highlighting the common intuition behind all the operations.

Using a version control system skillfully is not easy at all, and the subject should not be taken lightly. I truly hope that this book will help you gain a solid understanding of version control with Bazaar, and that you will become fully comfortable and effective using this fantastic tool.

What this book covers

Chapter 1, *Getting Started*, explains the concept of version control and how to install Bazaar.

Chapter 2, *Diving into Bazaar*, explains all the most important core operations by using the command-line interface and the GUI.

Chapter 3, *Using Branches*, explains all the various branch operations.

Chapter 4, *Using Bazaar in a Small Team*, explains how to work together with others in a small team, by branching and merging from each other.

Chapter 5, *Working with Bazaar in Centralized Mode*, explains the principles of the centralized mode and how to work in this mode by using Bazaar.

Chapter 6, *Working with Bazaar in Distributed Mode*, explains common distributed workflows and how to implement them by using Bazaar.

Chapter 7, *Integrating Bazaar in CDE*, explains how to integrate Bazaar into various collaborative development environments.

Chapter 8, *Using the Advanced Features of Bazaar*, explains practical tips that are not essential to using Bazaar, but can be very useful and make you more productive.

Chapter 9, *Using Bazaar Together with Other VCS*, explains how to use Bazaar to interact with other version control systems.

Chapter 10, *Programming Bazaar*, explains how to interact with Bazaar programmatically, and how to extend it by implementing plugins.

What you need for this book

You will need a computer where you can install Bazaar. The content of this book was tested in Windows, GNU/Linux, and Mac OS X systems, but Bazaar should work in any system where a supported version of Python is installed—`2.4`, `2.5`, `2.6`, or `2.7`.

Who this book is for

This book is designed for anyone who may be new to version control systems. If you are a programmer or a system administrator, you can benefit greatly by using Bazaar in your projects. To those who are already familiar with version control systems, this book should serve as a fast and easy way to understand Bazaar, and take advantage of its unique features.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Commands and code words in text are shown as follows: "You can check the status of the working tree by using the `bzr status` command."

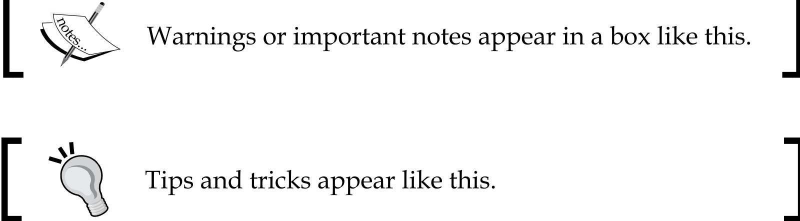
A block of code is set as follows:

```
from bzrlib.commands import plugin_cmds
plugin_cmds.register_lazy(
    'cmd_summary', [], 'bzrlib.plugins.summary.cmd_summary')
```

Any command-line input or output is written as follows:

```
$ bzr status
added:
  .bzrignore
unknown:
  Thumbs.db
  maps/Thumbs.db
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "In Windows, another way to launch Bazaar Explorer is from **Program Files** | **Bazaar** | **Bazaar Explorer**."



Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message. If there is a topic that you have expertise, and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Janos can be reached at `info@janosgyerik.com`.

Bugs in the examples can be reported at `https://bugs.launchpad.net/bzrbook-examples`.

Questions about the examples can be posted at `https://answers.launchpad.net/bzrbook-examples`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1 Getting Started

This chapter will get you started with the concept of version control, and explain why it is indispensable for anybody working with files, regardless of the project. It will introduce the core features of version control in general, and the basics and key differences between centralized, and distributed version control. Finally, we will get Bazaar up and running on your system, learn the very basics of the command-line and graphical interfaces, and how to get help using the built-in documentation.

The following topics will be covered in this chapter:

- What is a version control system and why you should care
- What is centralized version control
- What is distributed version control
- What is Bazaar
- How to install Bazaar and its plugins
- How to interact with Bazaar using the command-line interface
- How to interact with Bazaar using the graphical interface
- How to upgrade Bazaar
- How to uninstall Bazaar
- How to get help

Version control systems

A **version control system** (**VCS**) is essentially a tool to organize and track the history of changes to files in a project. This is more than just good book-keeping. A version control system can change the way you work and make you more productive. How, exactly? This will become clearer after considering the core features of a version control system and its implications.

Reverting a project to a previous state

A version control system enables you to record your changes to the files in a project, effectively building up a history of revisions. Having a complete history of changes in your project enables you to switch back-and-forth between revisions if you need to. For example:

- Restoring a file to a previous state; for example, to the point right before you deleted something important from it
- Restoring files or directories that you deleted at some point of time in the past
- Undoing changes introduced by specific revisions, affecting one or more files

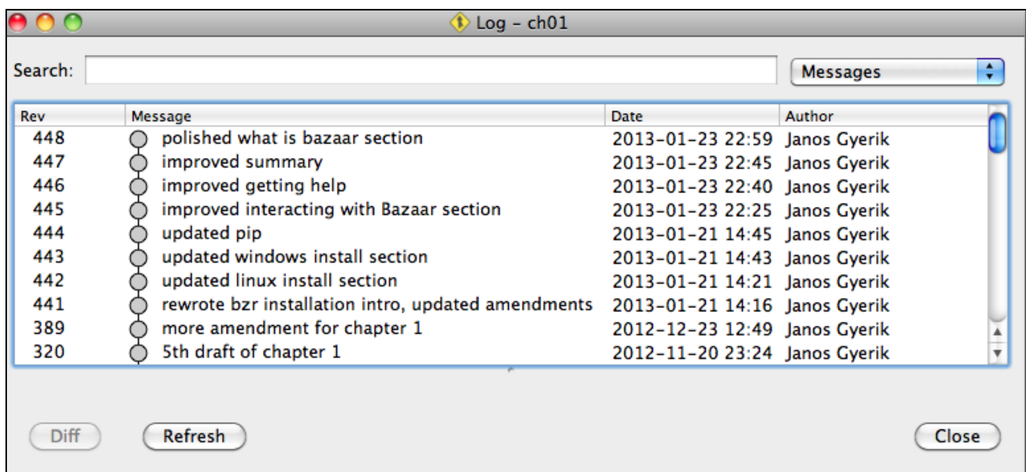
These are the most obvious benefits of keeping the history. However, there is a very powerful hidden benefit too—knowing that you can easily switch back to any previous state liberates your mind from worries that you might break something. Being able to return to any previous state means that you cannot really break anything. Once a revision is recorded in the history, you can always return to that state. Revisions are like snapshots, or milestones that you can return to anytime.

As a consequence, you can go ahead and make even drastic changes with bold confidence. This is a crucial point. This key feature enables you to focus on the real work itself, without the fear of losing anything.

Have you ever made a copy of a file or a directory and added a timestamp to the original one, so that you could make experimental changes? With a version control system, you can stop making copies and avoid getting lost in the sea of timestamped files and directories. You are free to experiment, knowing that you can return to any previous state at any time.

Viewing the log of changes

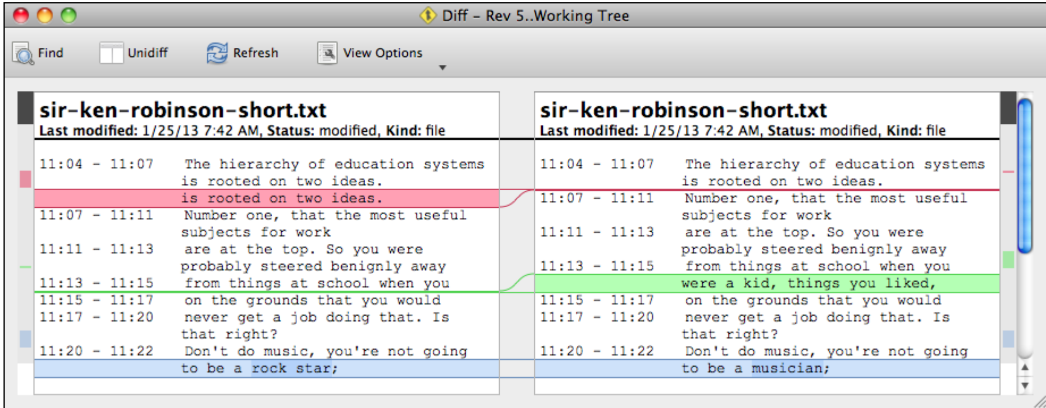
Having a full history of revisions is one thing. It is also important to have a simple way of viewing the history of changes; for example, an overview of what has changed from revision to revision, as follows:



This way, in case you need to retrieve something from a past revision, the log messages help to identify the exact point to jump to in the history. In a version control system, this typically works by entering a brief summary when recording a new revision. Often, the easiest way to find a particular past revision is by reading or searching the log of these summary messages, which should serve as a readable timeline or "changelog" of the project.

Viewing the differences between revisions

Being able to view files at any past state is great, but often what is even more interesting is the difference between two states. With a version control system, it is possible to make comparisons between any two states of specific files, directories, or the entire project. For example, the difference between two revisions of a text file can be displayed as follows:



www.allitebooks.com

Let's call the compared revisions **base** and **target**. The left-hand side shows the file as it was at the base revision, while the right-hand side is at the target revision. The coloring indicates what has changed, going from the base state to the target state:

- Lines with the red background in the left panel have been deleted
- Lines with the green background in the right panel have been added
- Lines with the blue background in both the panels have been changed; the changed part is highlighted with a deeper shade of blue

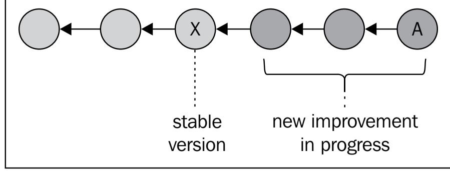
However, this kind of a detailed view of the differences is only possible for text files. In case of binary files, such as images, Word, or Excel files, the differences are binary and therefore are not human readable. In case of these and other binary formats, the only way to see the differences is to open both revisions of the file, and to compare them side by side.

Viewing the differences is most useful in projects with mostly plaintext files, such as software source code, system administration scripts, or other plaintext documents.

Branching and merging

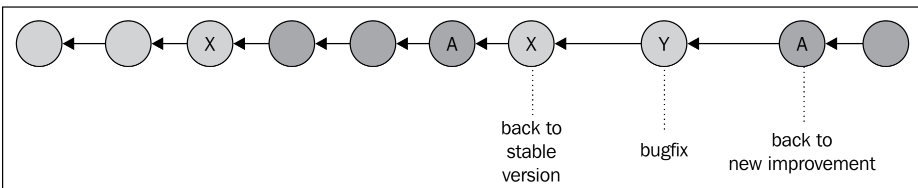
Being able to revert a project's files to any previous state gives you the freedom to make bold changes. What is even better, though, is if instead of completely undoing a set of experimental changes, you can work on multiple experimental improvements or ideas in parallel and switch between them easily.

Take, for example, a software project that is stable and works well at revision X. After revision X, you can start working on a new feature. As you progress, you can record a few revisions, but the feature is not complete yet. In fact the software is not stable at the moment until you finish the feature. At this point, the revision history may look something similar to the following:



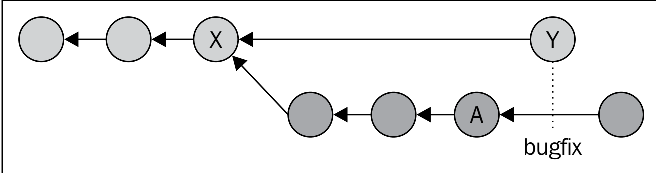
During this time, users use the stable version of the software based on revision X, and discover a serious problem that had been overlooked. Your current version of the project is incomplete, but you must fix the problem urgently and release a new stable version of the software. What can you do?

One solution is to revert to revision **X**, fix the problem, release the fixed version for the users, restore your work on the new improvement, and continue. While this is possible and the version control system helps by minimizing your effort, this solution is tedious and makes the revision history confusing to follow:



Effectively, we have confined ourselves to a linear history. Although it works, the result is awkward. Also, keep in mind that at some point you will want to reach a state that includes both the completed new improvement and the bugfix you did in revision **Y**, further confounding the revision history.

A much better and more natural solution is to break the linearity of the history and introduce a new branch, as follows:



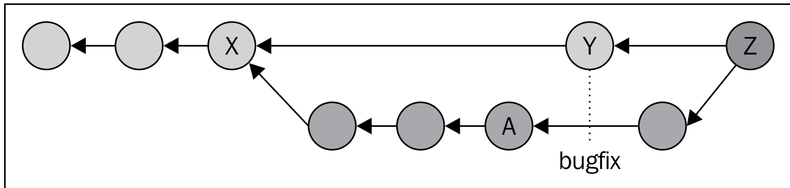
That is, instead of reverting your ongoing work on the new feature, create a new branch that is isolated from your current work and fix the problem of the stable version in that branch. A version control system can do this efficiently, using minimal additional disk space in the process.

Now, you have two parallel versions of the project—one that is stable and another that is a work in progress. The version control system makes it easy to switch between the two. You could have even more branches if needed. In reality, it is all too common that your current work must be interrupted for some reason, and branching is a practical solution in many situations. For example:

- You realize that you need more input from a colleague or another department to complete the current improvement you are working on
- A high priority task has come up that you have to switch to immediately
- You realize that your current approach might not be the best solution and you would like to try another method without throwing away what you've done so far, reserving the possibility to return later if needed

Our work is interrupted every day. Being able to work on multiple branches and switch between them easily can help a lot, minimizing the impact of interruptions and thereby saving us time and increasing our productivity.

Although being able to work on branches is great, what is even more important is bringing the various branches together, which is called **merging**. In the preceding examples and in most practical situations, having multiple branches is not the end goal, and most of the time, branches are temporary and short-lived. The end goal is to have all the improvements done on a project, unified in a single place, on a single branch, as follows:



Revision **Z** is the result of merging the two branches—the stable branch and the branch of the completed new improvement, and it should include all the work done in these branches.

Merging is a complicated and error-prone operation. It is an important job of a version control system to make merging as painless as possible, and intelligently apply the changes that were recorded in the branches you are trying to merge. However, when there are conflicting changes in two branches; for example, one branch modified a file and another branch deleted the same file, then the version control system cannot possibly figure out the right thing to do. In such relatively rare cases, a user must manually resolve the conflict.

Branching and merging does not have to be an advanced operation reserved for power users. A version control system can make this relatively easy and natural. Once you become comfortable with this feature, it will boost your productivity, allowing you to work on multiple ideas in parallel in an organized way. Branching and merging are especially crucial in collaboration. Without branching and merging, it is not possible to work in parallel; collaborators will have to work in lockstep, with only one person recording new revisions at the same time, which can be inefficient and unnatural.

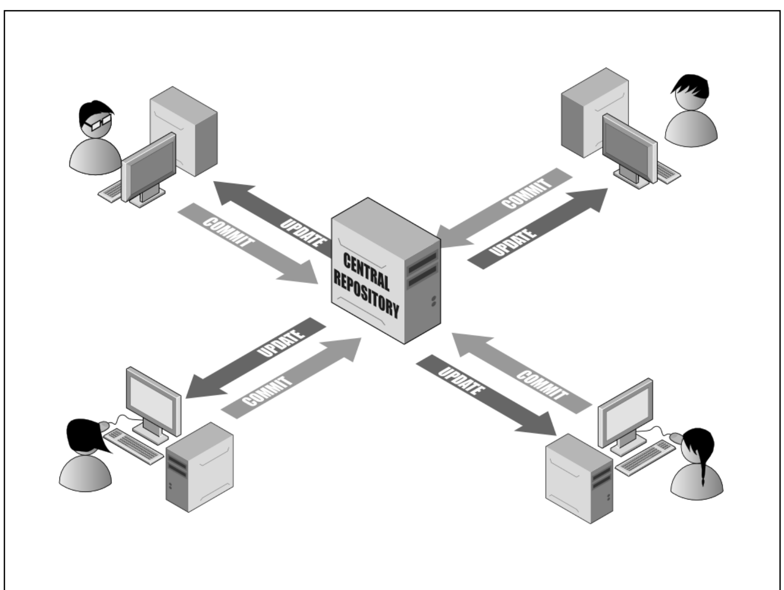
Acronyms related to version control

There are many acronyms and names related to version control that can be confusing sometimes, so it's probably worth clarifying them here:

- **Revision Control System (RCS)** is exactly the same as **Version Control System (VCS)**
- **DVCS** may be spelled as **Distributed VCS** or **Decentralized VCS**, and they both mean exactly the same thing
- **Distributed Revision Control System (DRCS)** is the same as **DVCS**
- **Source Code Management (SCM)** is **VCS** specifically applied to the source code in software development projects

Centralized version control systems (CVCS)

Centralized version control systems were created to make it possible for multiple collaborators to work on projects together. In these systems, the history of revisions is stored on a central server, and all the version control operations by all collaborators must go through this server. If a collaborator records a new revision, then all other collaborators can download and apply the revision in their own environments to update their project to the same state as the central server:



To avoid conflicting changes on the same file by multiple collaborators, such as concurrent modifications to the same lines, collaborators have to work in lockstep—after collaborator A has made some changes, collaborator B must first download those changes before he can add any new changes of his own.

Thanks to its simplicity, this is still a very popular workflow today, used by many large and famous projects and organizations. However, despite their popularity, centralized systems have serious drawbacks:

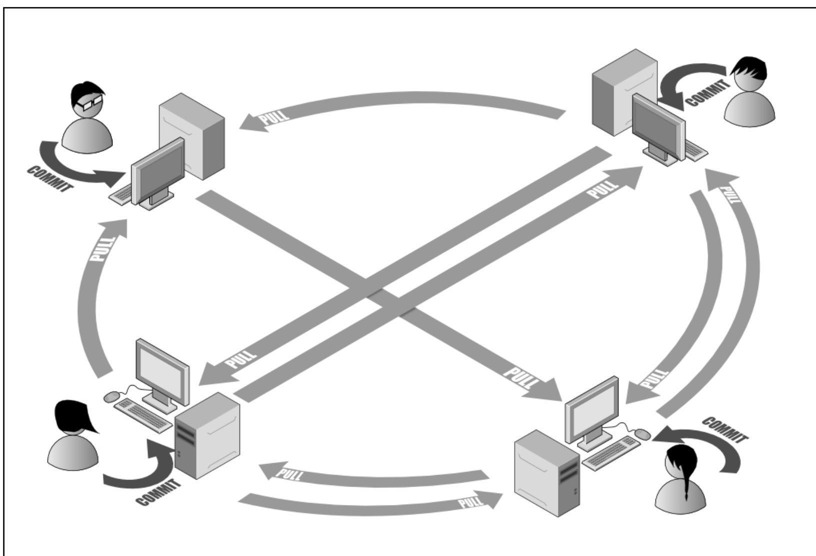
- Network access to the central server is required for all the operations that change the repository or access the revision history. As such, network outage and slowness can seriously impact productivity.
- The central server is a single point of failure—if the server is unavailable or lost, so is the revision history of the entire project.
- Administrative overhead—to prevent unauthorized access, user account and permission management must be configured and maintained.

Distributed version control systems (DVCS)

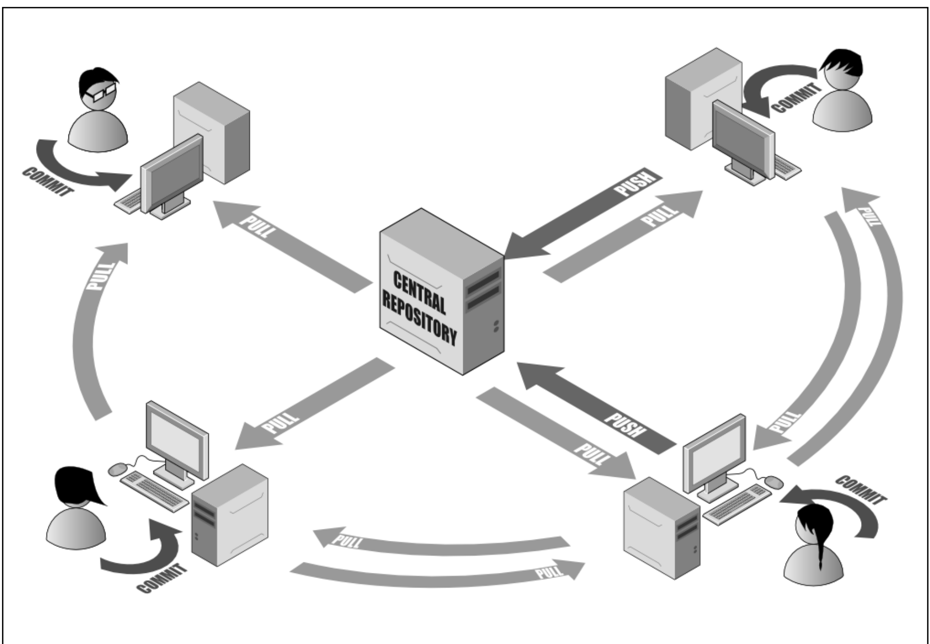
Distributed version control systems were created to make collaboration possible without a central server, and thus overcome many of the common issues with CVCS. This can work based on a few core principles:

- Each collaborator has the full revision history
- Collaborators can branch and merge from each other easily

The result is an architecture where there is no technical center, and any participant can potentially be the center:



Instead of a central server with the complete revision history, each collaborator has the full history in his/her own personal branches. Although technically there is no need for a central server, typically there is a designated common "official" public branch aggregating the work of all collaborators:



In general, a DCVS can do everything that a CVCS can, and enable many additional features. One of the most interesting added features is the many possible workflows for exchanging revisions between collaborators, such as:

- Merging revisions peer-to-peer
- Centralized—a branch is designated as the "official" branch, which can be used by collaborators in exactly the same way as in centralized version control
- Centralized with gatekeepers—the "official" branch is accessible by designated maintainers of the project, who merge changes peer-to-peer and publish releases in the "official" branch

Distributed version control is especially suitable for large teams with physically disconnected collaborators, such as most open source projects. However, it can be just as useful at smaller scales too, even in a solo project.

Distributed version control has important implications in terms of keeping backups of the project. By design, it is very easy to replicate the full revision history on a remote location or even on a local backup disk, thus providing a simple and consistent backup method. Considering that every collaborator begins working on new revisions by first grabbing the full history of the project, the vast majority of the revision history is very difficult to lose; the full history can only get lost if all the collaborators lose all their work. On the other hand, since the changes of all the collaborators are not necessarily at a single central location but distributed across all their local environments, there is also no single place to back up all the work done in the project. Thus, it is up to each individual collaborator to make sure that their local changes don't get lost before they are merged into the official branch or into other collaborator branches. Fortunately, this is not difficult to achieve, and we will provide examples to demonstrate how you can enjoy the benefits of distributed version control and at the same time stay safe by replicating your new revisions at another location.

What is Bazaar?

Bazaar is a distributed version control system, and as such one of the most powerful version control tools that exists today. At the same time, it is friendly, flexible, consistent, and easy to learn. It can be used effectively from very small solo projects, to very large distributed projects, and everything else in between.

Bazaar is written in Python, it is open source and completely free, and is an official GNU project, licensed under GPLv2. It is sponsored by Canonical, and used by many large projects, such as the Ubuntu operating system, Launchpad, MySQL, OpenStack, Inkscape, and many others. The official website for hosting Bazaar projects is **Launchpad** (`http://launchpad.net/`), where you can find many interesting projects that use Bazaar.

This book will explain how to make the most out of version control, and how to accomplish all the features outlined earlier with Bazaar and much more. The next chapters will explain how to use Bazaar in increasingly advanced use cases. Each scenario will build on the previous one, gradually revealing the added benefits of each increasingly sophisticated setup, and how they will improve your productivity, whether you are working solo or as part of a large team.

Installing Bazaar and its plugins

Bazaar is implemented in Python, therefore it should work on any system where a supported version of Python is installed (2.4, 2.5, 2.6, or 2.7). The core module of Bazaar consists of `bzrlib`, a Python library that implements the core functionality of Bazaar, and `bzr`, the command-line interface. Bazaar is highly extensible, and a wide selection of official and unofficial plugins exist enriching its functionality. For the purpose of this book, the most important plugins to include are:

- **explorer**: Bazaar Explorer is the graphical user interface of Bazaar
- **qbzr**: This is a Qt-based frontend for Bazaar, which provides a graphical user interface for most core `bzr` commands
- **svn**, **git**, and **fastimport**: These plugins help to interoperate with foreign repositories

On Windows and Mac OS X, the official installer includes the core module and a good selection of commonly used plugins by default. On GNU/Linux and other systems, the core module and each plugin are packaged separately, and you must install them individually.

Visit the official download page to find the right installer and installation instructions for your system at `http://wiki.bazaar.canonical.com/Download`.

Here, we explain only the most typical and simple installation options. For more advanced scenarios, please refer to the download page for details.

GNU/Linux

Most modern GNU/Linux distributions include Bazaar in their official binary repositories, in the package `bzr`. This package typically includes only the core functionality of Bazaar, and plugins are found in separate packages. The most important plugin we will use throughout the book is the Bazaar Explorer plugin, usually in the package `bzr-explorer`.

You can discover other plugins with additional functionality in packages starting with `bzr-` in their name.

Ubuntu, Debian, and derivatives

Use your favorite package manager tool, or the following command:

```
$ sudo apt-get install bzr bzr-explorer
```

> **Downloading the example code**
>
> You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Red Hat, Fedora, CentOS, and derivatives

Use your favorite package manager tool, or the following command:

```
$ sudo yum install bzr bzr-explorer
```

openSUSE and derivatives

Use your favorite package manager tool, or the following command:

```
$ sudo zypper install bzr bzr-explorer
```

Installing Bazaar using pip

Keep in mind that generally it is recommended to install Bazaar using the official binary repository of your distribution, in order to benefit from the advanced package management features of your operating system, such as automatic security update notifications and software upgrades.

Pip is the next generation Python package management tool. The benefit of using pip to install Bazaar is that it provides the latest stable, and unstable versions of Bazaar, whereas the official binary repository of your operating system may be a bit out of date. If you prefer to have the latest version, then using pip can be a good option. Another potential benefit of using pip is that it allows you to install Bazaar inside your home directory rather than system-wide, thus it makes it possible to install Bazaar even if you don't have administrator rights in a system.

If you don't already have pip, you can install it using the graphical or the command-line package manager of your distribution; for example, in Ubuntu:

```
$ sudo apt-get install pip
```

If you don't have administrator rights, another way to install pip is using easy_install, which is the legacy package manager utility of Python:

```
$ easy_install --user pip
```

Once you have pip, you can install Bazaar system-wide to make it available to all users, as follows:

```
$ sudo pip install bzr bzr-explorer
```

Or install only for your user (into ~/.local/), as follows:

```
$ pip install --user bzr bzr-explorer
```

To discover other Bazaar plugins with additional functionality, search for packages starting with bzr- in their name, as follows:

```
$ pip search bzr-
```

Other installation methods

There is a more detailed explanation on the Bazaar download page, which can be useful if you are not using the latest version of these distributions, if you are using another distribution, or if you prefer to build and install Bazaar from source:

http://wiki.bazaar.canonical.com/DistroDownloads

Windows

The download page offers different types of the Bazaar installers, such as standalone or Python-based at http://wiki.bazaar.canonical.com/WindowsDownloads.

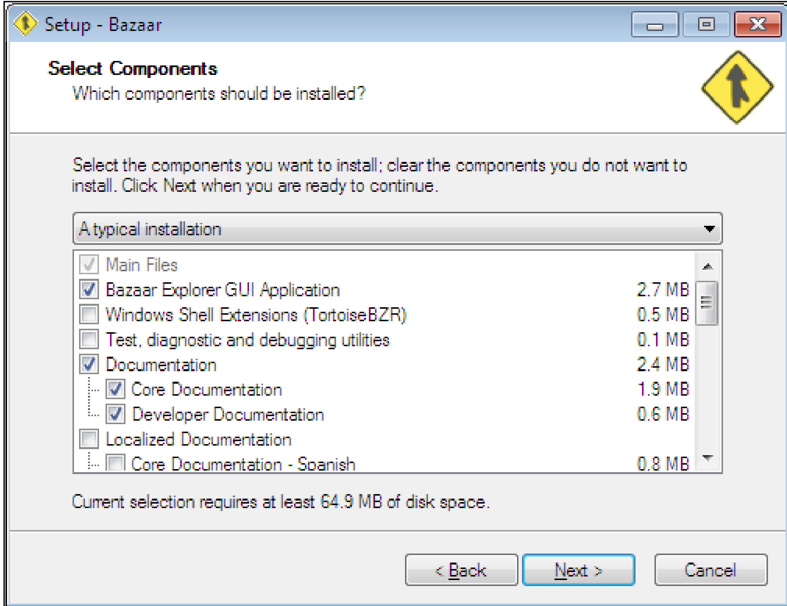
The standalone installer includes all the dependencies of Bazaar, most notably a Python interpreter. This installer is about 20 MB in size, and will use between 50 MB to 70 MB disk space on your computer, depending upon the components and plugins you select during installation. If you are not sure which installer to choose, then choose this one.

The Python-based installers assume that you already have a specific version of Python installed. This can be a good option if you want to save disk space. However, these installers do not include some dependencies of Bazaar, and you will have to install them by yourself. See the following documentation for details:

http://wiki.bazaar.canonical.com/BzrWin32Installer#bzr-dependencies

Depending upon the type of installer you choose, there may be different releases of Bazaar available. It is recommended that you pick up the latest stable release.

During installation, you can choose the components to install. The default selection includes the Bazaar Explorer, the documentation, and a good set of additional plugins. You may simply accept the defaults for now. If you want to install additional components later, simply run the installer again:



If you prefer to install Bazaar using Cygwin, you can use the standard Cygwin installer `setup.exe` file and look for the package `bzr`.

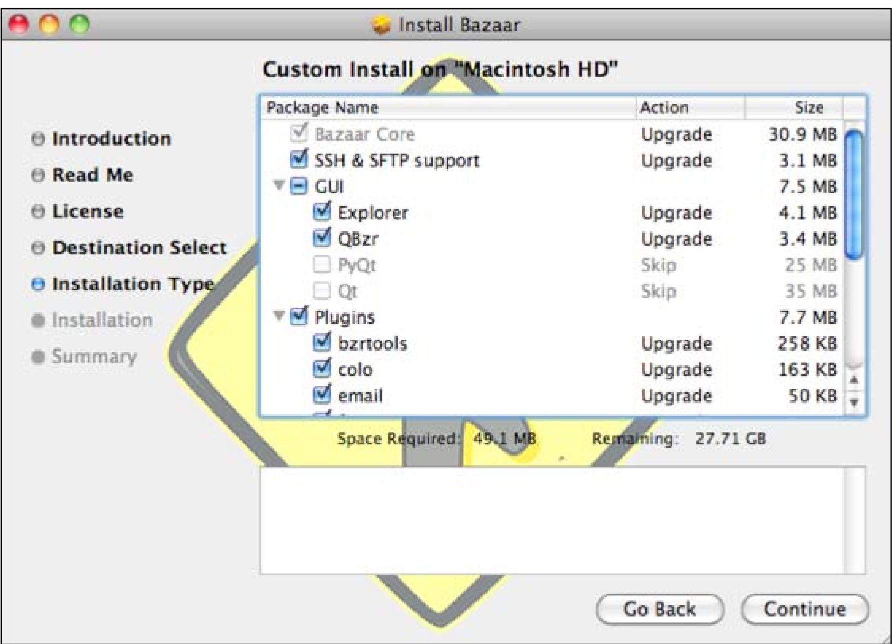
Mac OS X

The download site offers `dmg` packages for recent versions of Mac OS X; it is recommended to choose the latest stable release of Bazaar:

`http://wiki.bazaar.canonical.com/MacOSXDownloads`

At the time of this writing, the latest platform is Snow Leopard; there are no installers specifically for Lion or Mountain Lion. If your Mac OS X is Lion or above, use the Snow Leopard installer.

During installation, you can choose the components to install. By default, all the components and plugins are selected, including the Bazaar Explorer and documentation, which will take up about 50 MB disk space on your computer. If you deselect some components now, you can install them later by running the installer again:



The download page explains more advanced installation options, such as using Homebrew, MacPorts, Fink, or source.

Bazaar in a shared hosting environment

If you want to install Bazaar in a shared hosting environment, where you have shell access, then the easiest way may be using Python package management tools such as `pip` or `easy_install`.

`pip` is the next generation Python package manager. If it is not installed in your shared hosting environment, you can try to install it with `easy_install`:

```
$ easy_install --user pip
```

Before installing Bazaar itself, it is recommended to install `pyrex` and `paramiko`:

```
$ pip install --user pyrex
$ pip install --user paramiko
```

At the time of this writing, when installing Bazaar with pip, it chooses the latest beta release instead of the latest stable release. If that is not what you want, you can specify the version like this:

```
$ pip install --user bzr==2.5 bzr-explorer
```

Interacting with Bazaar

The most straightforward way to interact with Bazaar is the command-line interface. In this book, we will cover both the command-line interface and Bazaar Explorer, which is the official graphical user interface, but keep in mind that the latter is still beta status.

Using the command-line interface

A good way to confirm that the installation was successful is checking the version. Open a terminal application, such as DOS prompt in Windows or terminal in other operating systems and run the following command:

```
$ bzr version
```

The output should look something similar to the following:

```
Bazaar (bzr) 2.5.0
 Python interpreter: /usr/bin/python 2.6.6
 Python standard library: /usr/lib/python2.6
 Platform: Linux-3.2.0-2-amd64-x86_64-with-debian-wheezy-sid
 bzrlib: /usr/lib/python2.7/dist-packages/bzrlib
 Bazaar configuration: /home/jack/.bazaar
 Bazaar log file: /home/jack/.bzr.log

Copyright 2005-2012 Canonical Ltd.
http://bazaar.canonical.com/

bzr comes with ABSOLUTELY NO WARRANTY. bzr is free software, and
you may use, modify and redistribute it under the terms of the GNU
General Public License version 2 or later.

Bazaar is part of the GNU Project to produce a free operating system.
```

In addition to the version number, the command prints other useful information, such as the location of the Python interpreter used, the Bazaar libraries (`bzrlib`), and the user's configuration directory.

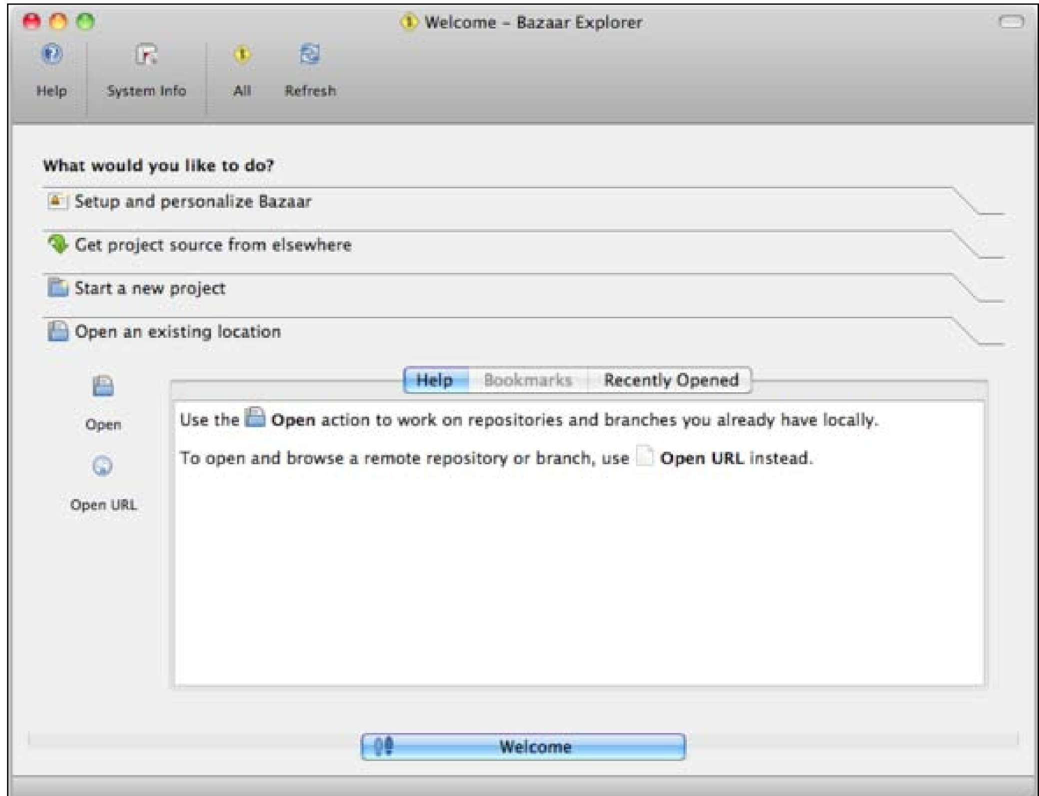
Using the graphical user interface

The graphical user interface is called Bazaar Explorer. It is included in the `explorer` plugin. In all the systems, you can start Bazaar Explorer using the following command:

```
$ bzr explorer
```

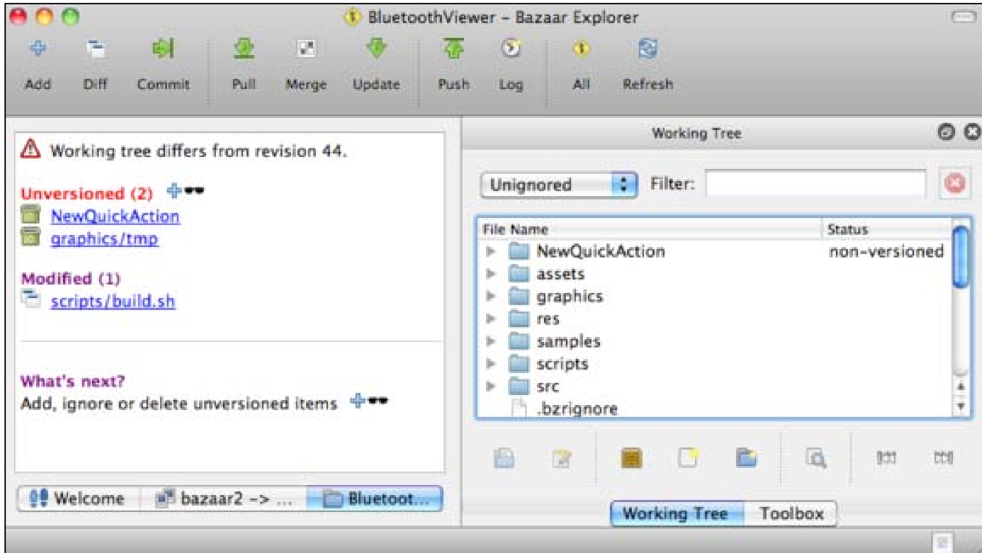
In Windows, another way to launch Bazaar Explorer is from **Program Files | Bazaar | Bazaar Explorer**.

Bazaar Explorer will open with the **Welcome** view as follows:

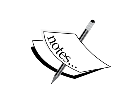


The top part is a toolbar with buttons to perform the most common version control operations. The main part of the screen shows some typical operations you might want to perform, such as open existing projects, start a new project, or customize Bazaar. All of these options will be explained in the next chapter; for now, we just wanted to confirm that it works.

Bazaar Explorer is similar to a regular file explorer, except that it is specialized for viewing Bazaar project directories. When you open an existing Bazaar project, the Working Tree panel on the right looks just like a regular file explorer, showing the list of files and subdirectories in the project:

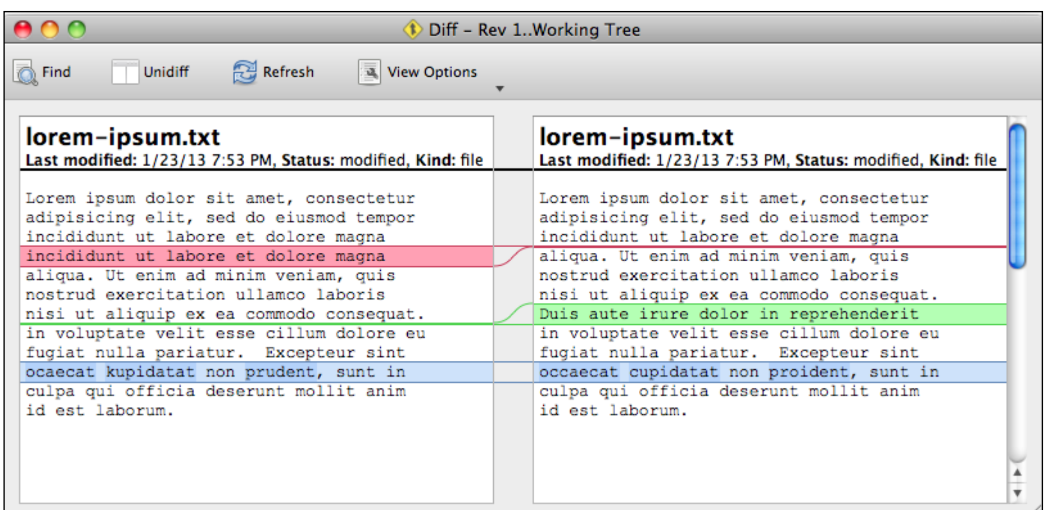


In addition, the **Status** column in the Working Tree panel and the left panel indicates files that have not been added to version control yet (nonversioned), or files that have been modified since the last recorded revision.

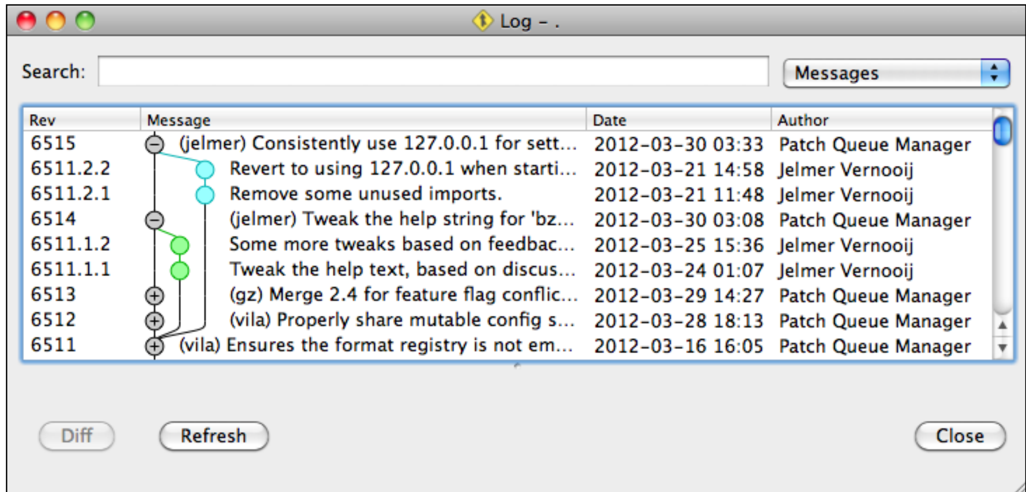


The "working tree" is the main method to interact with Bazaar and perform the various version control operations on a project. This concept will be explained in detail in the next chapter.

One of the nice features of Bazaar Explorer is the graphical visualization of differences in text files, for example:



Another very practical use case is browsing the change history, with the various branches of the project presented in a nicely formatted way:



Using the two user interfaces together

You can perform the most common operations with whichever interface, but each will have some advantages and disadvantages depending upon the situation. In general, the command-line interface can be faster and more efficient when you are already familiar with Bazaar's commands. On the other hand, typically for viewing operations such as browsing or searching in the history, or comparing revisions, Bazaar Explorer is often more practical. In this way, the two user interfaces complement each other.

Throughout this book, we will focus more on the command-line interface, mainly for the sake of clarity. Command-line expressions tend to be more accurate and unambiguous in general. For this reason, understanding the command-line interface is essential, while the graphical user interface is optional.

Upgrading Bazaar to the latest version

The procedure to upgrade Bazaar to the latest version depends upon your operating system:

- **Windows and Mac OS X**: Simply download and run the latest version of the installer. It will replace your existing installation.
- **GNU/Linux**: Use the package manager of your distribution.
- **pip**: Use the `--upgrade` flag of the `install` command; for example, `pip install --upgrade bzr bzr-explorer`.

Uninstalling Bazaar

The procedure to uninstall Bazaar depends upon your operating system:

- **Windows**: Use the application wizard (`appwiz.cpl`) to remove Bazaar with all its plugins
- **Mac OS X**: At the time of this writing, there is no standard way of uninstalling Bazaar on Mac OS X systems
- **GNU/Linux**: Use the package manager of your distribution
- **pip**: Use the `pip uninstall bzr bzr-explorer` command

Getting help

Bazaar has a superb built-in help system. Simply type `bzr` in a terminal without any parameters, and it will give you a list of the basic commands and how to get more detailed help:

```
$ bzr
Bazaar 2.5.0 -- a free distributed version-control tool
http://bazaar.canonical.com/

Basic commands:
 bzr init     makes this directory a versioned branch
 bzr branch    make a copy of another branch

 bzr add      make files or directories versioned
 bzr ignore    ignore a file or pattern
 bzr mv      move or rename a versioned file

 bzr status    summarize changes in working copy
 bzr diff     show detailed diffs

 bzr merge     pull in changes from another branch
 bzr commit    save some or all changes
 bzr send     send changes via email

 bzr log      show history of changes
 bzr check     validate storage

 bzr help init   more help on e.g. init command
 bzr help commands list all commands
 bzr help topics  list all help topics
```

You can use the `bzr help` command to read the documentation of any Bazaar command, as well as common topics.

- `bzr help`: This command provides a brief summary of the basic commands (same as simply `bzr` without parameters)
- `bzr help some_command`: This command provides a detailed help on `some_command`
- `bzr help commands`: This command lists all commands
- `bzr help topics`: This command lists all topics

All the commands accept the `-h` or `--help` flag consistently, and print the same help message as the `bzr help some_command` command.

If you cannot find something in the built-in documentation, then you can explore the official online documentation, which contains everything from short tutorials to complete in-depth references, FAQ, glossary, and further resources at `http://doc.bazaar.canonical.com/en/`.

Summary

You should have a good idea why it is so important to use a version control system, and expect that Bazaar will make this very easy for you. With Bazaar, you will be able to revert to any past state, work on multiple ideas in parallel, and effectively use the various operations of version control. Having Bazaar installed on your computer, you are now ready to dive in and learn how to use all these features of version control.

In the next chapter, you will learn to convert any directory on your computer to a Bazaar repository, record changes, view the history of changes, and revert to a previous state. You will be able to apply these steps to any of your existing projects, and begin to enjoy the benefits of version control.

2

Diving into Bazaar

This chapter will explain and demonstrate, with examples, the basic version control operations using Bazaar in the simplest possible scenario—working solo on a single branch. Although the scenario is simple, the operations you will learn here are the very core functionality of version control, and will remain relevant throughout your experience with Bazaar, even in more complex setups with multiple collaborators and branches.

The main topics in this chapter are as follows:

- Introducing the command-line interface and Bazaar Explorer
- First-time setup—configuring the author setting
- Performing the basic version control operations
- Beyond the basics—performing other practical operations

Understanding the core concepts

Before diving into Bazaar, it is essential to understand its four core concepts:

- **Revision**: This is a snapshot of the project's files
- **Repository**: This is a store of revisions recorded in the project
- **Branch**: This is an ordering of revisions – a history
- **Working tree**: This is a directory tree in your filesystem, which contains your files and subdirectories associated with a branch and a revision

These concepts are central to Bazaar and appear everywhere in the documentation. Therefore, it is crucial to understand them well.

www.allitebooks.com

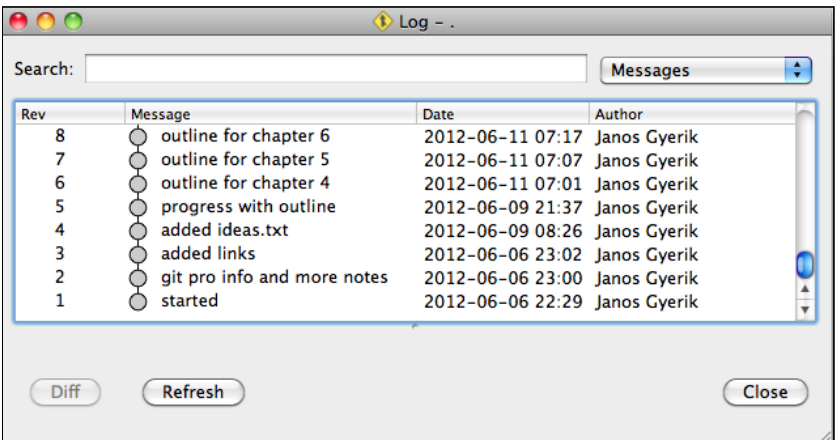
Revision

A revision is a snapshot of the tree of files and directories in a project recorded at some specific point in time. As you make changes to the files in the project, you record a new snapshot, which is a new revision. In this way, the revisions will represent logical steps forward in the evolution of the project. Recording a revision is called **committing**. In addition to the state of the project, a revision contains metadata such as:

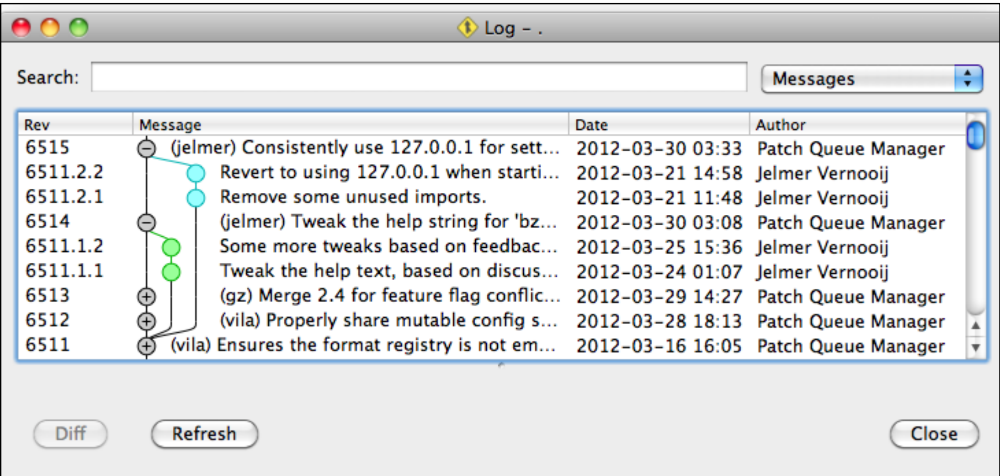
- **Committer**: This represents the user who recorded the revision
- **Timestamp**: This represents the date and time the revision was recorded
- **Message**: This represents a short description of what has changed in the revision, often called the **commit log**

In Bazaar, revisions on a linear timeline (on a single branch) are numbered with an integer sequence starting from 1, and incremented by one for each new revision recorded.

The following is an example of a commit log, which shows the aforementioned metadata and revisions numbers:

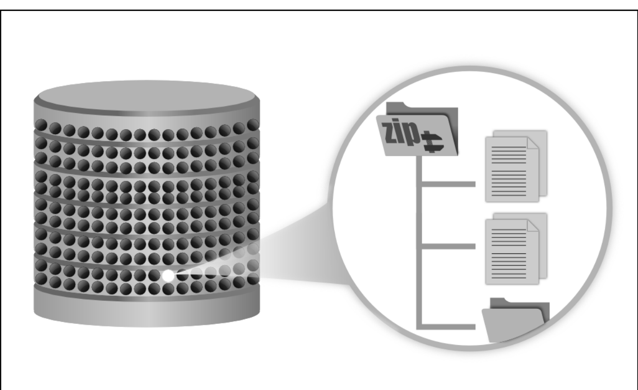


When working on multiple branches, the timeline becomes non-linear and a dotted notation is used. This will be explained later, but to give you an idea, the following is an example of a commit log with multiple branches:



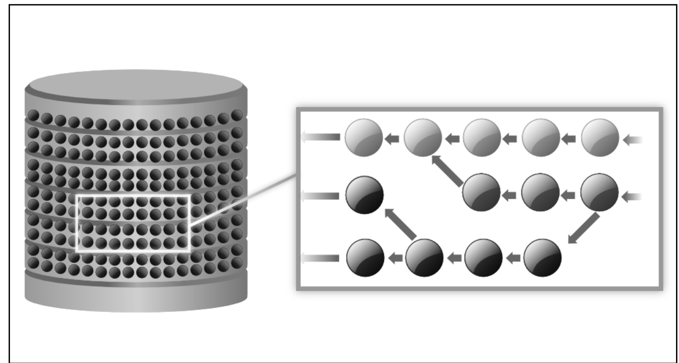
Repository

A repository is simply a store of revisions. The content and metadata of revisions are stored inside a repository in a compact and efficient way. Often, the total size of the repository, including all revisions, is smaller than the total size of all the files of the latest revision in an uncompressed form:



Branch

A branch represents the ordering of revisions; in other words, how the revisions follow each other in the history.

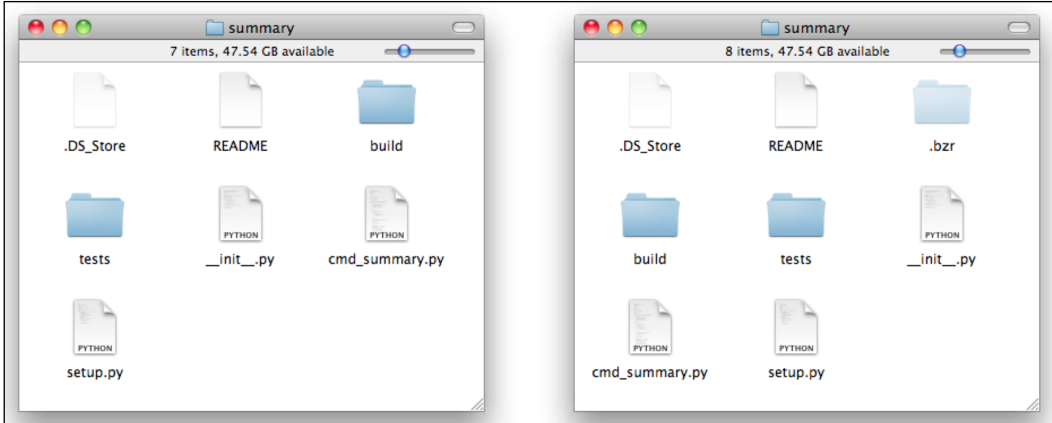


Each revision contains a reference to its parent revision, or multiple references if there are multiple parents, as is the case when a revision is created by the merging of two or more other revisions. A branch has precisely one latest revision, called the tip. By following the parent relationships from the tip, revisions form a directed acyclic graph, representing the ordering of revisions within the branch.

Working tree

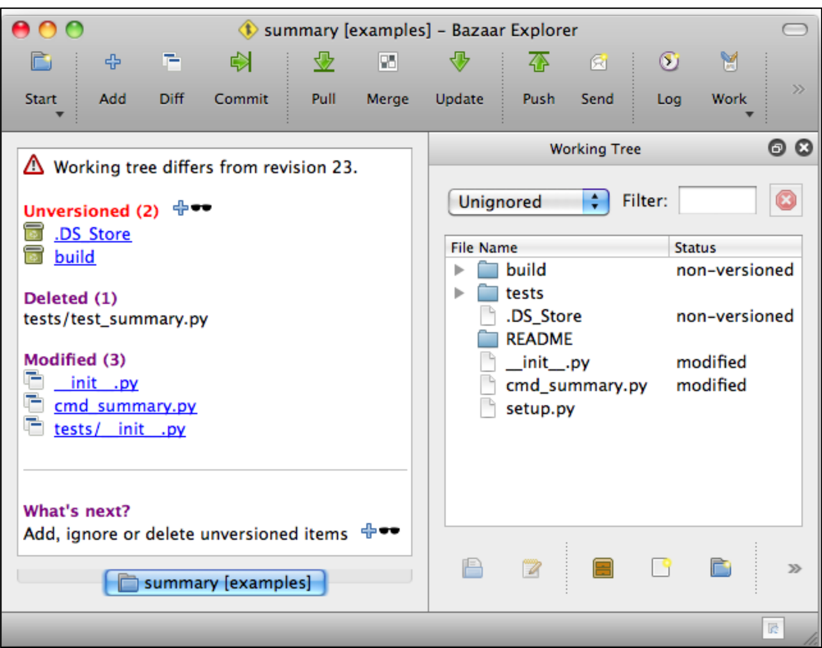
A working tree is associated with a branch. It contains the files and subdirectories of the project at a specific revision of the branch. It looks and behaves exactly like any other regular directory in the filesystem. In addition to the project's files, a working tree contains a hidden subdirectory named `.bzr`, which is used by Bazaar to track the state of the project's files relative to the associated revision.

To illustrate with an example—at the left is a regular directory, and at the right is the same directory converted to a working tree. The only difference is the presence of a hidden `.bzr` directory:



A working tree can be set to any revision of its associated branch, effectively resetting its content to the files and subdirectories as they existed at the given revision. In this way, you can browse the content of the project at any past revision.

The most common use of the working tree is to view the project's files at the latest revision of the branch, so that you can work with those files—edit their content, add new files, or delete them. At any time, you can either revert some or all of your pending changes to their original state (of the last revision), or commit your changes to record the current state as a new revision of the project:



Putting the concepts together

When you work with a project under version control, you will use all of these concepts together, as follows:

- You use a working tree to access and modify the files of your project. The working tree looks and behaves like any other ordinary directory in your filesystem, with an added hidden `.bzr` directory to store Bazaar's files.
- The working tree is linked to a Bazaar branch, which in turn is linked to a Bazaar repository. The branch represents the order in which revisions are recorded, and the repository stores the content of revisions as snapshots in a compressed form.
- As you make changes in the working tree and record them as a new revision, the content of the revision is added to the repository as a new snapshot, and the branch tip is updated to point to the new revision.

The four core concepts are vital in most version control operations. Therefore, it is good to understand what they are and how they are linked together.

Storing Bazaar's data in the filesystem

Bazaar stores its files in one or more hidden `.bzr` directories, depending upon the configuration.

In this chapter, we will focus on the most simple configuration, where the working tree data, the branch data, and the repository data are all stored in a single hidden `.bzr` directory in the working tree, as follows:

```
/path/to/working/tree/.bzr
|-- branch      # branch data (ordering of revisions)
|-- checkout    # working tree data (track pending changes)
|-- repository  # repository data (content of revisions)
```

This configuration is called a **standalone tree**.

Just to give you an alternative example, another common configuration is the **shared repository**, which will be explained in the next chapter. In this configuration, the repository data is stored in a `.bzr` directory at a higher level, as follows:

```
/path/to/shared/repo/
|-- .bzr
|   |-- repository      # repository data
```

```
|-- some_branch
|   |-- ...            # project files in the working tree
|   |-- .bzr           # hidden .bzr directory
|       |-- checkout   # working tree data
|       |-- branch     # branch data
|-- another_branch
|   |-- ...            # project files in the working tree
|   |-- .bzr           # hidden .bzr directory
        |-- checkout   # working tree data
|       |-- branch     # branch data
|-- yet_another_branch
|   |-- ...
```

In this configuration, there are multiple working trees, each in a separate subdirectory, corresponding to different branches. The `.bzr` directory of each working tree contains the working tree data and the branch data, but not the repository data. The repository data is stored in the `.bzr` directory of the parent directory. In this setup, the repository data is shared by the branches.

Throughout the book, we will explain various configurations and their practical use cases in detail. For now, it is enough just to be aware that the working tree data, the branch data, and the repository data may be stored at different locations depending upon the use case.

Introducing the user interfaces

There are two primary ways to interact with Bazaar—running `bzr` commands in a terminal program, or using the Bazaar Explorer graphical user interface. Both the interfaces have their advantages and disadvantages, but we will try to cover them equally throughout the book whenever possible.

Using the command-line interface (CLI)

`bzr` is the **command-line client** and official user interface of Bazaar. It is the clearest way to explain all the version control operations, and often the fastest way to accomplish tasks.

To use `bzr`, you need a terminal application (or DOS prompt in Windows systems), and basic familiarity with shell commands of your local system, in order to be able to navigate in your local filesystem.

Using Bazaar Explorer

Bazaar Explorer is the **graphical user interface** (**GUI**) of Bazaar. In Windows, you can launch it from **Program Files** or the **Start** menu. In other systems, you can launch it by running the `bzr explorer` command in a terminal application.

With Bazaar Explorer, it is easy to see an overview of all the files in a project and pending changes. In addition, it has many practical features for filtering, searching, and history browsing that simply wouldn't be possible by using the command-line interface. However, not all operations are available in Bazaar Explorer. For this reason, it is important to understand the command-line interface too.

Configuring Bazaar

Before we can really dive in, it is good to configure the following user settings:

- `email`: This is the author information, which is recorded in each revision
- `editor`: This represents the default text editor used to write the revision log messages and edit files in Bazaar Explorer

These settings are stored in the `bazaar.conf` file in the Bazaar configuration directory, which you can find in the output of the `bzr version` command. You can view and edit the configuration using the command-line interface or Bazaar Explorer.

Configuring the author information

The author is part of the metadata recorded, together with the content of the revision. In Bazaar, the author is a string in the following format:

```
NAME <EMAIL>
```

For example, `Janos Gyerik <janos@example.com>`.

You can check the current setting by running `bzr whoami` without any parameters:

```
$ bzr whoami
Janos Gyerik <janos@testvm>
```

Depending upon your system, Bazaar may figure out a sensible default value based on your user account information. In this example, the e-mail address was auto-detected by Bazaar as my username in the system and the hostname of the system.

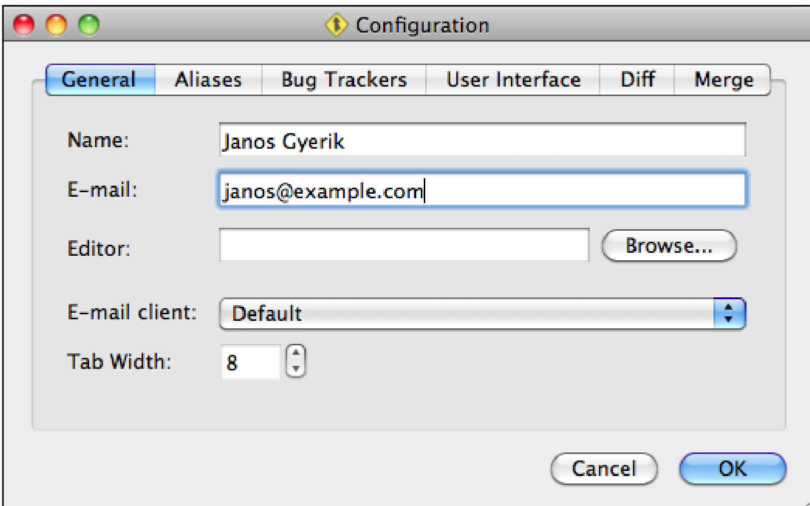
It is recommended that you use a real e-mail address in this setting, that collaborators can reach you by. You can update the setting by specifying the new value as a parameter. For example:

```
$ bzr whoami 'Janos Gyerik <janos@example.com>'
```

You can change this setting anytime, however, this will only affect future revisions you record; the author information cannot be changed in the past revisions.

The `bzr whoami` command effectively shows or sets the value of the e-mail setting in the user configuration file.

In Bazaar Explorer, you can find the author settings in the **Setup and personalize Bazaar** tab's **Configuration** item, or by selecting **Settings** | **Configuration** | **User Configuration**. The most important are the **Name** and **E-mail** fields in the **General** tab:



Configuring the default editor

The default editor is used to enter a log message when recording a revision, and when opening files in Bazaar Explorer for editing. It is stored in the editor setting in the user configuration file.

You can view the value of the setting by using the following command:

```
$ bzr config editor
/usr/local/bin/edit
```

You can change the value of the setting by using the following command:

```
$ bzr config editor=/usr/local/bin/edit --scope=bazaar
```

Make sure to use a plaintext editor, such as Notepad or Notepad++ in Windows; TextEdit, TextMate, or TextWrangler in Mac OS X; gedit, gvim, or vim in GNU/Linux.

Another way to confirm and edit this setting is by using Bazaar Explorer in the same way as you would while editing the author information—in the **Setup and personalize Bazaar** tab, click on **Configuration**, or select **Settings | Configuration | User Configuration** in the **Application** menu.

Other configuration options

For more efficient screenshots in this book, we have disabled **Toolbox** in the **Status** view, as its content is static and not relevant in the examples. If you want to do the same, open the **Preferences** view, and on the **Appearance** tab, uncheck the **Show the toolbox** checkbox on the **Status** view.

Performing the basic version control operations

In this section, we will show how to put any regular directory under version control using Bazaar, and demonstrate the basic version control operations on it, such as the following:

- Checking the status of files and directories in the project
- Adding files
- Recording a new revision
- Ignoring files
- Deleting files
- Undoing changes
- Editing files
- Viewing differences in changed files
- Renaming or moving files
- Viewing the revision history
- Restoring files from a previous revision

We will demonstrate each operation in the context of an example—planning a dinner party using plaintext files. The important point of the example steps is not the content of the files but the nature of the changes and the version control operations that we will perform. You can repeat the steps similarly on any directory on your computer, or download this ZIP file with the sample directory we used:

`https://launchpad.net/bzrbook-examples/trunk/examples/+download/dinner-party.zip`

Extract the ZIP file anywhere on your computer; for example, to `/sandbox/dinner-party` or `C:\sandbox\dinner-party`. The examples in this chapter will assume this path of the sample project, but you may choose any other path you prefer.

Putting a directory under version control

To manage a directory using version control, we need three things:

- A repository to store revisions
- A branch to represent the ordering of revisions
- A working tree to view and edit the project files and interface with the repository and the branch

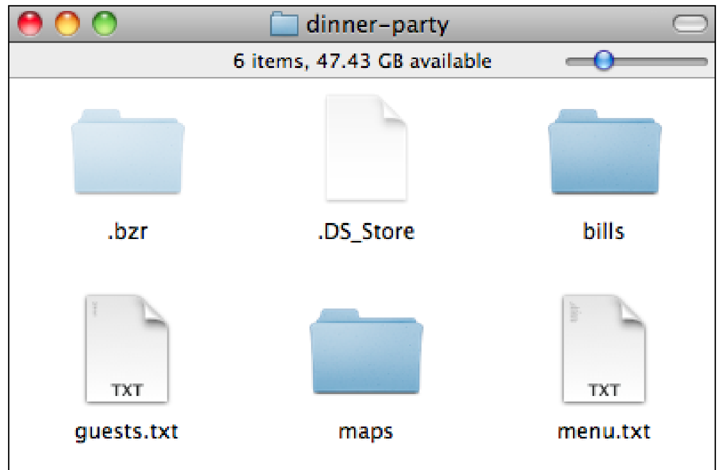
Bazaar will do all this by converting the directory to a working tree. This involves creating a hidden `.bzr` directory inside the selected directory, which will store the repository, the branch, and the files for tracking the state of the working tree. The files that existed in the directory will not be changed in any way.

Using the command line

You can convert an existing regular directory into a working tree using the `bzr init` command. You can either specify the directory as a parameter, or first change into the directory by using the `cd` command, then run `bzr init` without parameters::

```
$ cd /sandbox/dinner-party
$ bzr init
Created a standalone tree (format: 2a)
```

The following screenshot confirms that a hidden `.bzr` directory has been created:



Other than the hidden `.bzr` directory, nothing else has changed. If at this point, you remove the `.bzr` directory, you will be back to where we started.

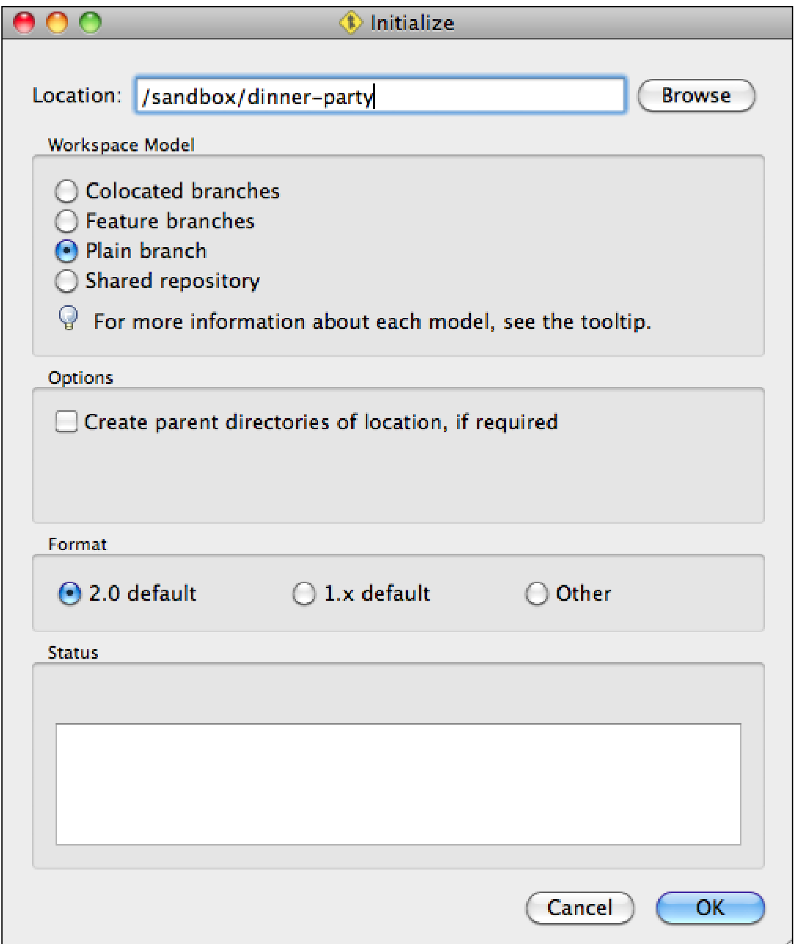
In the output of `bzr init`, Bazaar tells us that it has created a standalone tree, using the storage format `2a`. A standalone tree is when the `.bzr` directory contains both a repository and a branch. We will see other configurations later, where the repository and the branch are at different locations. The `2a` storage format is the default in the Bazaar 2.x series; it is not important for the scope of this book. Since we have not recorded a revision yet, the repository and branch are empty at this point.

Using Bazaar Explorer

In Bazaar Explorer, you can convert an existing regular directory to a working tree using the **Initialize** view. You can open this view in several ways:

- From the menu, select **Bazaar** | **Start** | **Initialize**
- From the **Welcome** view, in the **Start a new project** tab, select **Initialize**

Using the shortcuts *Ctrl* + *N* (Windows, Linux) or *Cmd* + *N* (Mac OS X). In the **Location** text box, you can either type the path to the directory to convert, or click on the **Browse** button and navigate to it. In the **Workspace Model** box, select **Plain branch**:

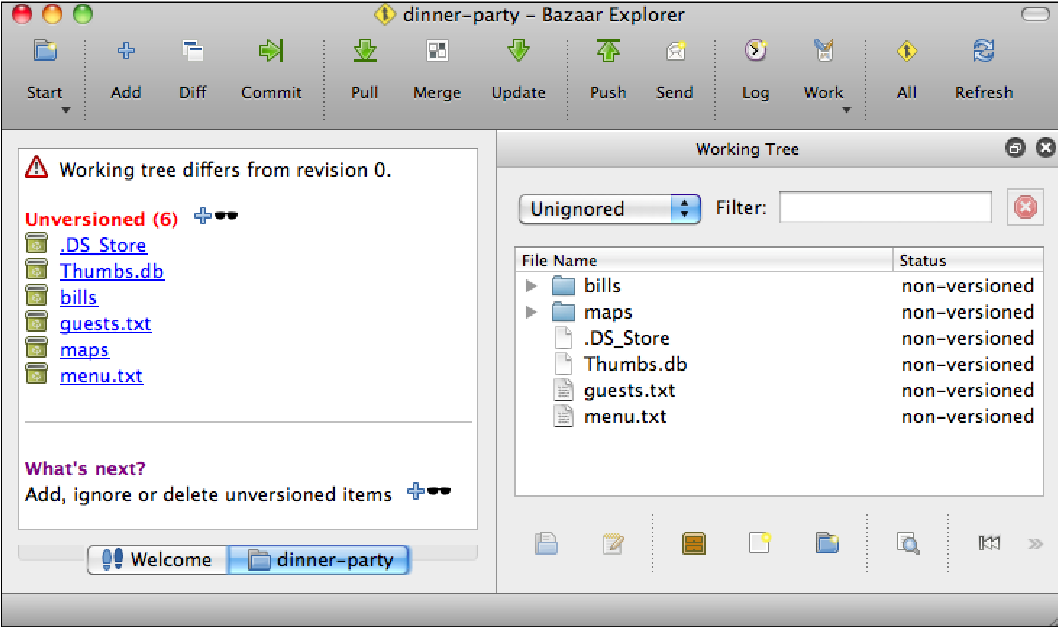


After you click on **OK**, the **Status** box will show the `bzr` command that was executed and its output. For example:

```
Run command: bzr new --plain-branch --format 2a /sandbox/dinner-party
Created branch at /sandbox/dinner-party
```

Typically, the commands executed by Bazaar Explorer are slightly more verbose than what we normally use on the command line, mainly because Bazaar Explorer spells out options with default values that normally can be omitted, such as the `--format` parameter in this example.

After you click on **Close** to dismiss the **Initialize** view, the **Status** view will open, showing the files and directories in the working tree with their status:



Checking the status of files and directories

Checking the status of the working tree is one of the most important operations when using version control. The `status` command reports the outstanding changes in the project's files and directories as compared to the last revision, such as:

- Files that have been modified
- Files and directories that have been deleted from the working tree
- Files and directories that have been renamed or moved
- Files and directories that have been explicitly added to the project
- Files and directories that are unknown—they exist in the working tree but have not been explicitly added to the project
- Change in the execution bit of files

Using the command line

You can check the status of the working tree by using the `bzr status` command:

```
$ bzr status
unknown:
  .DS_Store
  Thumbs.db
  bills/
  guests.txt
  maps/
  menu.txt
```

Although we have converted our sample directory into a working tree, all the files in it are reported as unknown. This is because we have not told Bazaar that these files should be part of the project and it does not make such assumptions.

As we demonstrate different kinds of changes to the project, we will use the status command extensively, so you will see many more interesting examples.

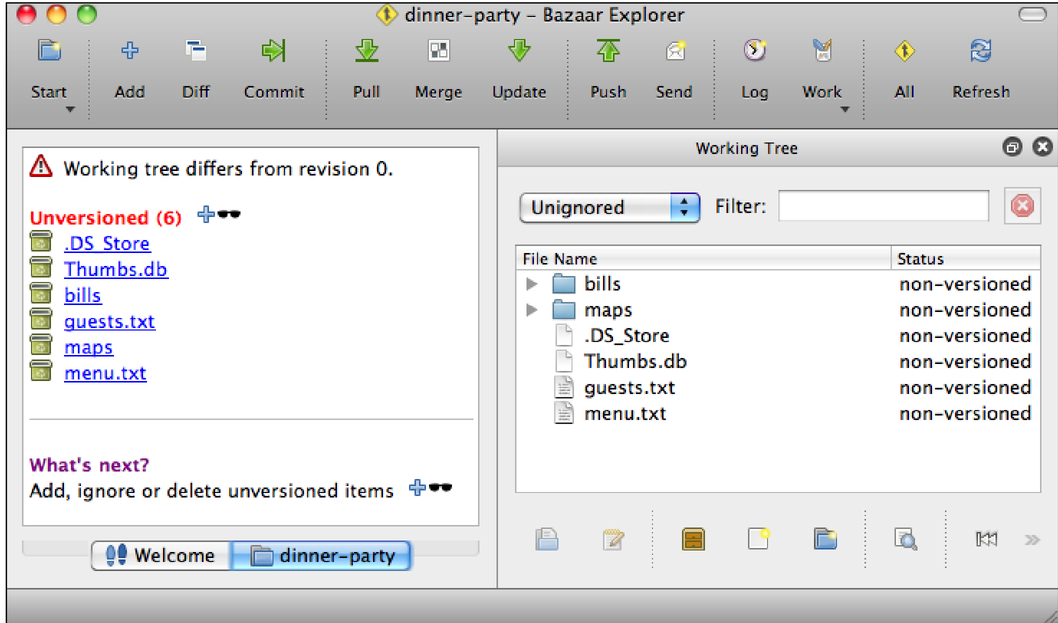
When used without parameters, the status command tells you the status of the entire working tree. If you want to see the status of only a set of files or directories, simply specify them as parameters, for example:

```
$ bzr status guests.txt menu.txt
unknown:
  guests.txt
  menu.txt
```

When specifying a directory, Bazaar will show the status of all the files and subdirectories within that directory. This is especially useful when there are a lot of changes in the project and you want to see the status of only a subset of all the files. You can specify multiple files and directories at the same time.

Using Bazaar Explorer

In Bazaar Explorer, you can see the status of the project's files and directories at a glance in the **Status** view, as we saw earlier after initializing our sample branch:



In the left panel, we see a warning sign saying **Working tree differs from revision 0**. Since we haven't recorded any revisions yet, the repository is empty. In Bazaar, this state is "revision 0". Under the warning message, we see a list of unversioned files. This term is a synonym of "unknown" files used by the command-line interface; both mean that these files exist in the working tree but we have not told Bazaar that they should be part of the project.

In the right panel, we see a list of files similar to the ones in a common file explorer. The **Status** column shows the status of each file, at the moment all non-versioned, which is yet another synonym for "unknown".

Since the terms "unknown", "unversioned", and "non-versioned", all mean the same thing in Bazaar, we will refer to all of these as simply "unknown" throughout this book, sticking to the language of the command-line interface.

If you click on a file listed in the left panel, Bazaar Explorer will open it in the default text editor.

At this point, we can observe some advantages of Bazaar Explorer over the command line. The **Filter** box in the right panel is very helpful in showing only a subset of all the files—type something into the box and the view will show only the files whose name includes the pattern; for example, `txt`. You can also filter files by their status, using the combobox at the left of **Filter**. These features are especially useful in large projects with many files.

> The right panel of the **Status** view is not refreshed automatically when making changes to the working tree outside of Bazaar Explorer, even though the left panel is always up-to-date. When you suspect that the right panel might be out of sync, click on the large **Refresh** button in the toolbar.

Adding files to version control

Bazaar does not make assumptions about the files in your working tree, and treats them as "unknown" until you specify that they should be part of the project under version control.

Adding files to version control is a two-step process. First, you specify to Bazaar the files and directories you want to add. At that point, Bazaar acknowledges your intent, and when you check the status of the project, you will see the list of files marked for adding. To complete the operation and really add the files, you must record a new revision, so that the new files become part of the project's history. The second step is recording a new revision. This step completes the operation, and only after that files will be really added to version control, before that it's just a pending change.

Using the command line

You can mark files to be added to the project using the `bzr add` command. Without parameters, it will add all the unknown files. To add only some of the unknown files, you must specify them explicitly as parameters. For example:

```
$ bzr add guests.txt bills/
adding guests.txt
adding bills
adding bills/restaurant.txt
```

As a result, Bazaar prints the list of files it has marked for adding. When you specify a directory, Bazaar adds all the unknown files in that directory and all its subdirectories.

Let's confirm the status of the project:

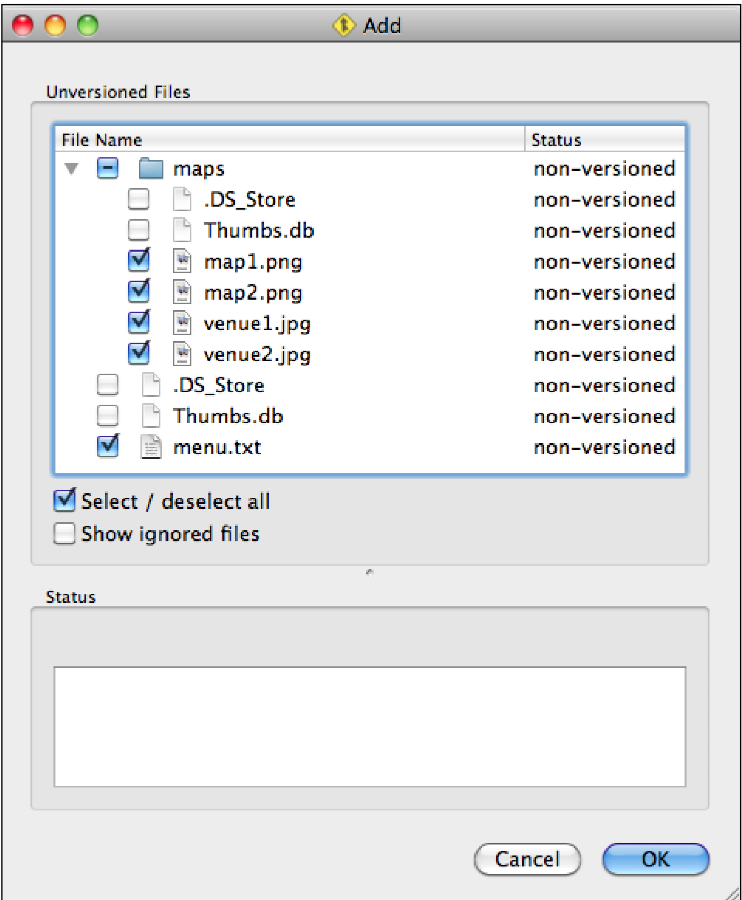
```
$ bzr status
added:
  bills/
  bills/restaurant.txt
  guests.txt
unknown:
  .DS_Store
  Thumbs.db
  maps/
  menu.txt
```

Now, some of the files are marked as unknown, and the others as added. The files are not really added yet until we record a new revision.

When specifying a directory, Bazaar will add all files and subdirectories within that directory. You can specify multiple files and directories at the same time.

Using Bazaar Explorer

In Bazaar Explorer, you can mark files to be added to the project by clicking on the large **Add** button in the toolbar or on the small plus icon in the left panel of the **Status** view. This opens up the **Add** view, with all unknown files automatically selected by default:

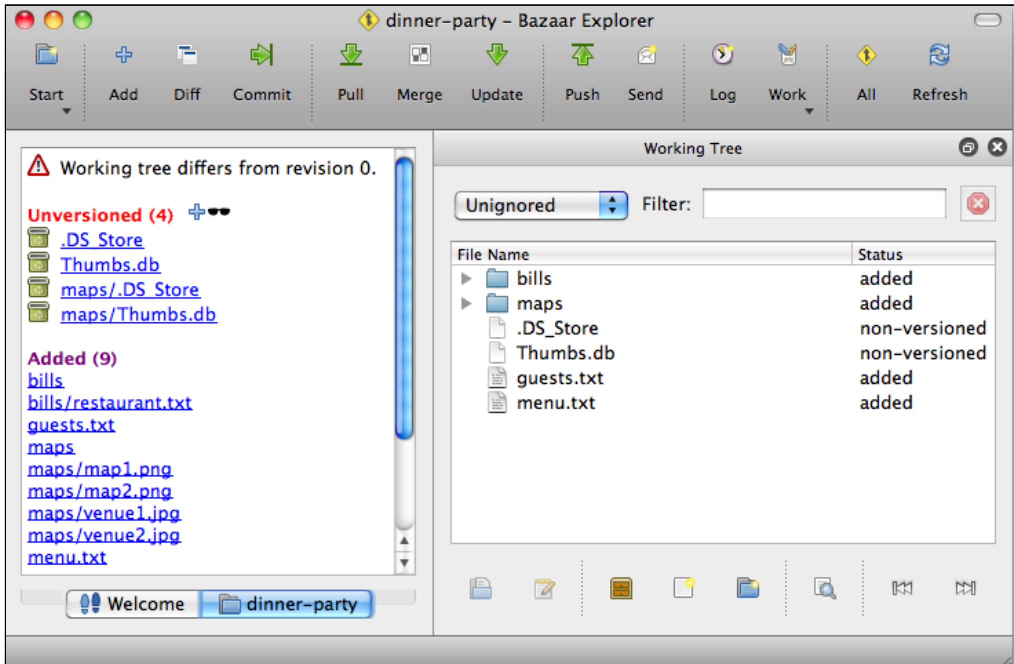


In this example, we have de-selected all the `.DS_Store` and `Thumbs.db` files. These are hidden files, automatically generated in Mac OS X and Windows systems to cache meta information about the directory contents, to speed up browsing with a file explorer. These files are pointless to add to version control since they are automatically generated and do not have any useful content to track.

After you click on **OK**, the **Status** box will show the `bzr` command that was executed and its output, as follows:

```
Run command: bzr add --no-recurse menu.txt maps/map1.png maps/map2.png
maps/venue1.jpg maps/venue2.jpg
adding menu.txt
adding maps
adding maps/map1.png
adding maps/map2.png
adding maps/venue1.jpg
adding maps/venue2.jpg
```

Click on **Close** to dismiss the **Add** view and return to the **Status** view. The left panel is updated to show only one unknown file left and 7 marked to be added. Similarly, the **Status** column in the right panel is also updated:



Another way to add files is to right-click on them in the right panel and select **Add**. If you select a directory, all the unknown files in all its subdirectories will be marked to be added.

Recording a new revision

In the previous step, we marked the files to be added, but to really add them to version control so that we can start tracking changes to them, we must record a new revision, called a "commit" operation.

Once you commit a change in version control, it will become part of the history of the project. Although you can undo anything later, you cannot remove something once it is added. For this reason, you should always double-check what you are adding, in order to avoid polluting the history with garbage, such as the `.DS_Store` or `Thumbs.db` files.

Before committing a new revision, it is important to double-check what exactly will be committed, either using the `status` command or the left panel of the **Status** view in Bazaar Explorer.

You can commit all the pending changes at once, or you may want to include only a subset of them. It is a good idea to commit logically-related changes together, separating unrelated changes to multiple commits, and thus distinct revisions.

The commit operations allow us to enter a message that will be recorded together with the selected changes to the project's files. A good message explains the changes of the revision as briefly as possible. The commit message can be crucial later, when looking through the history to find a specific revision.

Using the command line

You can commit the changes using the `bzr commit` command. Without parameters, it will commit all the pending changes. To commit changes to only some of the files, you must specify them explicitly as parameters. If you specify a directory, all the changes in that directory will be included in the commit. For example:

```
$ bzr commit bills
```

This will open up a text editor with the summary of changes that will be committed:

```
-------------- This line and the following will be ignored --------------
added:
  bills/
  bills/restaurant.txt
```

To complete the commit, enter a brief message at the top, describing the modifications in the new revision; for example, `added bills`, then save and exit the editor:

```
Committing to: /sandbox/dinner-party/
added bills
added bills/restaurant.txt
Committed revision 1.
```

You can cancel the commit by exiting the editor without saving. In this case, Bazaar will give you the option to go ahead with the commit or cancel it as follows:

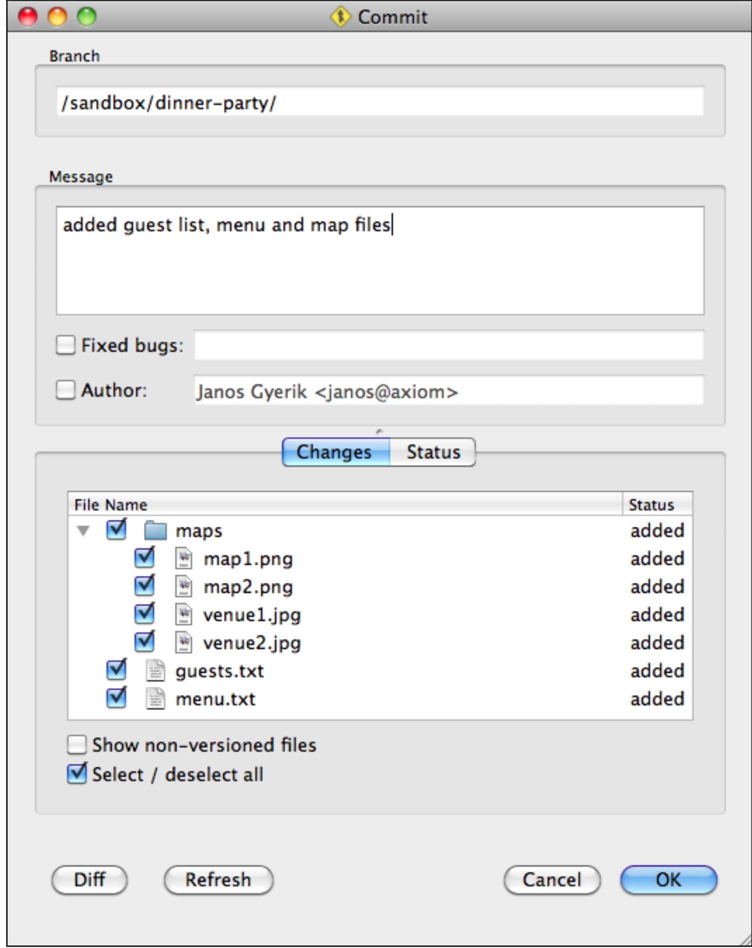
```
Committing to: /sandbox/dinner-party/
added bills
added bills/restaurant.txt
Commit message was not edited, use anyway? ([y]es, [n]o):
```

Press *n* here to cancel the commit. You should always write a commit message.

As the output of the commit command, Bazaar prints the URL of the branch we are committing to, the summary of changes, and the revision number. Revisions in a branch are numbered as a sequence of integers starting from 1.

Using Bazaar Explorer

Since we have already committed some of the changes, the warning message in the left panel of the **Status** view now shows that we have differences from revision 1 instead of revision 0. To commit the remaining changes, click on the large **Commit** button in the toolbar. This opens the **Commit** view:



The **Branch** box at the top shows the filesystem path of the Bazaar branch.

The **Message** box is for entering a brief summary for the new revision.

The lower part of the dialog box shows a list of files that will be committed. Using the checkboxes, you can deselect files you don't want to commit at this point.

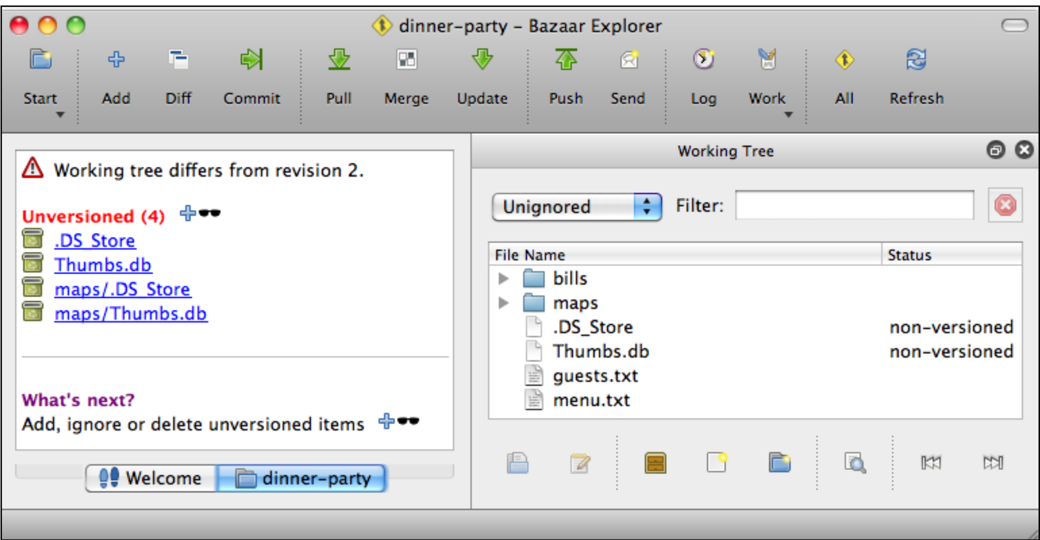
If you double-click on any of the files, Bazaar will open the difference viewer tool to show the changes in that file.

If you click on the **Diff** button, Bazaar will open the difference viewer tool to show all the changes in all the files to be committed.

After entering the commit message, click on **OK** to perform the commit and record a new revision. The **Status** box will show the `bzr` command that was executed and its output. For example:

```
Run command: bzr commit -m "added guest list, menu and map files" maps
guests.txt menu.txt maps/map1.png maps/map2.png maps/venue1.jpg maps/v...
Committing to: /sandbox/dinner-party/
added maps
added maps/map1.png
added maps/map2.png
added maps/venue1.jpg
added maps/venue2.jpg
added menu.txt
added guests.txt
Committed revision 2.
```

Click on **Close** to dismiss the **Commit** view and return to the **Status** view:



As expected, the revision number has been incremented to 2, and now the only outstanding changes are the unknown `.DS_Store` and `Thumbs.db` files.

Ignoring files

Some files are better left out of version control. For example, hidden files generated by your operating system, such as `.DS_Store` and `Thumbs.db`, or build products in software development projects, such as `*.o` files in C programming, or `.class` files in Java programming, and so on. Since these files are automatically generated, it's pointless to add them to version control. Adding these files can also cause problems in collaboration when the files are generated slightly differently depending upon the environment of each collaborator.

You can tell Bazaar to ignore specific files or filename patterns, so that they stop showing up in the output of the status command and in Bazaar Explorer. Bazaar stores the ignore definitions inside a hidden file named `.bzrignore`. Normally, this file is good to have under version control, so that all collaborators use the same ignore rules.

Bazaar ignores certain files by default. You can see these with the following:

```
$ bzr ignore --default-rules
*.a
*.o
*.py[co]
*.so
*.sw[nop]
*~
.#*
[#]*#
__pycache__
bzr-orphans
```

As you can see, patterns may use wildcards such as `*.o` to match all files in the project ending with `.o`. The pattern `*.py[co]` matches all files ending with `.pyc` or `.pyo`.

You can specify exceptions using lines starting with `!`, which will take precedence over regular patterns. For example, using these rules, all files ending with `.class` will be ignored by Bazaar except `special.class`:

```
*.class
!special.class
```

Using the command line

You can tell Bazaar to ignore files or patterns using the `bzr ignore` command, for example:

```
$ bzr ignore .DS_Store
```

This adds the specified files or patterns to `.bzrignore`, and if `.bzrignore` is not yet in version control, Bazaar automatically marks it to be added, as if you ran `bzr add` on it. You can confirm this with the `status` command:

```
$ bzr status
added:
  .bzrignore
unknown:
  Thumbs.db
  maps/Thumbs.db
```

Notice that all the `.DS_Store` files are now gone from the unknown group, since the rule we have added to `.bzrignore` is already in effect. At the same time, `.bzrignore` is now marked to be added. Let's confirm the content of `.bzrignore`:

```
$ more .bzrignore
.DS_Store
```

There is only one single line, with the parameter we specified on the command line.

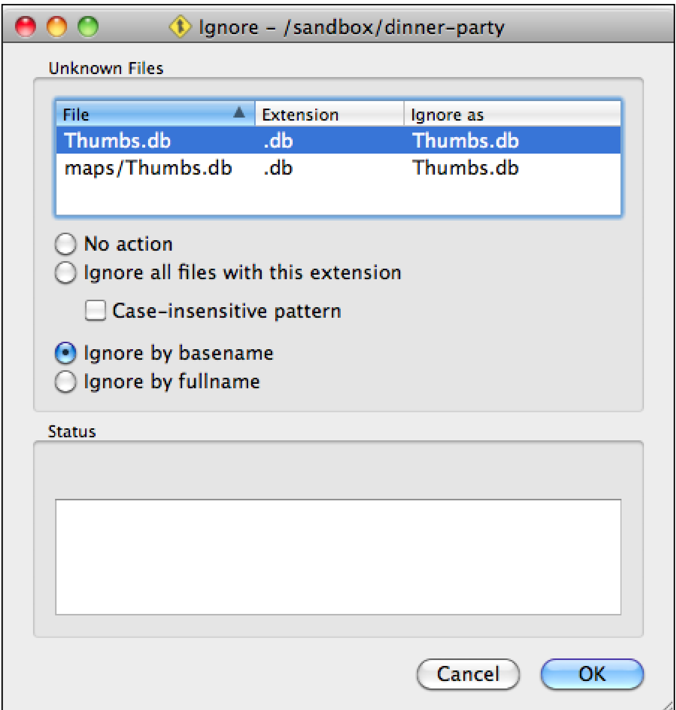
To check at anytime the files that are inside the working tree, but ignored by the defined rules, you can use the `bzr ignored` command:

```
$ bzr ignored
.DS_Store                                 .DS_Store
maps/.DS_Store                            .DS_Store
```

The output shows the files that are ignored on the left, and the rules that cause them to be ignored on the right.

Using Bazaar Explorer

In Bazaar Explorer, you can create ignore rules for unknown files using the **Ignore** view. When there are unknown files in the working tree, they are listed in the left panel of the **Status** view, with a sunglasses icon next to them. Click on this icon to open the **Ignore** view:



All the currently unknown files are listed at the top. Click on a file to choose an ignore rule. By default, the **No action** option is selected for each file. As you change the option, the **Ignore as** column is updated to show the rule that would be added to the `.bzrignore` file.

In the case of the `Thumbs.db` file, the **Ignore by basename** option is most suitable. The pattern in this case will be `Thumbs.db`, and as a result all **Thumbs.db** files in all the subdirectories of the project will be ignored.

In contrast, if you select **Ignore by fullname**, the pattern is `./Thumbs.db`, which matches only the `Thumbs.db` file at the top-level directory of the project; other `Thumbs.db` files in subdirectories will not be ignored.

If there had been more unknown files, you could select each of them one by one to choose an appropriate action.

After you click on **OK**, the **Status** box will show the `bzr` command that was executed and its output, as follows:

```
Run command: bzr ignore Thumbs.db
```

Click on **Close** to dismiss the window and return to the **Status** view.

Editing ignore patterns is quite limited in Bazaar Explorer. After choosing an ignore action for an unknown file, you cannot return to edit the setting later. If you want to edit ignore patterns later, the only way is to edit the `.bzrignore` file in a text editor.

Checkpoint

Let's record a new revision at this point, by adding the `.bzrignore` file in version control. Use the command line or Bazaar Explorer as you prefer. Enter a suitable message; for example, `added ignore patterns`.

Deleting files

Files that have been added under version control can be removed safely, because even if you remove them accidently, they can be restored from the repository later, if necessary.

When you tell Bazaar to remove files, they will be removed from the working tree immediately, but a new revision will not be recorded until you commit this change.

Another way to remove files from the working tree is to simply delete them from the filesystem. In this case, Bazaar will notice that some files are missing and assume that you probably want to delete them. However, it is better to always explicitly tell Bazaar to remove files, because that way it will take backups if necessary. For example, if you have made changes to a file but haven't committed it, then it's not safe to delete it as the changes will be lost. If you use Bazaar commands to remove such files, Bazaar will keep the backup files as a precaution, so that you can restore them later if needed.

Backup files are copies of the original with a postfix like `.~1~`; for example, the first backup of `guests.txt` will be named `guests.txt.~1~`, the second `guests.txt.~2~`, and so on. If and when you want to clean them up, it is up to you to decide whether you do or don't need them anymore and remove them from the working tree accordingly. These files are not under version control, and ignored by global ignore rules, and as such, they are listed in the output of the `bzr ignored` command.

Using the command line

You can tell Bazaar to delete files or directories from the project using the `bzr remove` command. For example:

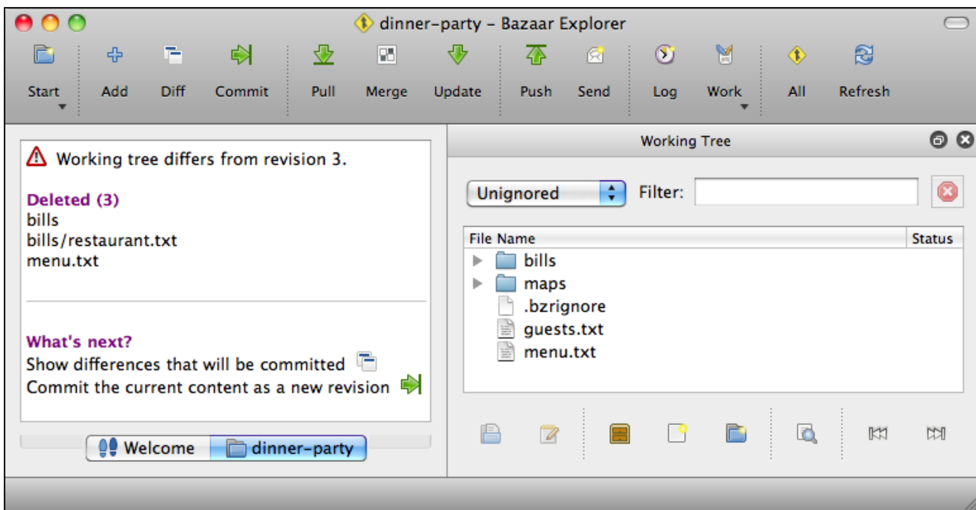
```
$ bzr remove bills/ menu.txt
deleted menu.txt
deleted bills/restaurant.txt
deleted bills
```

The specified files are immediately removed from the working tree, and by checking the status, we can confirm that the files have been scheduled for removal in the next revision:

```
$ bzr status
removed:
  bills/
  bills/restaurant.txt
  menu.txt
```

Using Bazaar Explorer

To delete files using Bazaar Explorer, right-click on the files in the **Status** view and select **Remove**. Files that have been removed are listed in the left panel:



If you remove files from the working tree without telling Bazaar about the action, Bazaar Explorer will show them with the status as `missing` instead of removing them from the **Status** view.

Undoing changes

Changes you make in the working tree are not permanent until you commit to the repository as a new revision. You can restore the content of the working tree to the state of the last revision at any time. This is called a **revert operation**.

The revert operation can be used not only to undo uncommitted changes, but also to restore the entire working tree or only a selected set of files to any past revision.

In the previous section, we deleted some files but have not committed the changes. This is a good opportunity to demonstrate how to revert changes before they are committed.

Using the command line

You can revert changes by using the `bzr revert` command. Without parameters, it will revert all the changes in the entire working tree. To revert only some of the changes, specify the files whose changes you want to undo as parameters. For example:

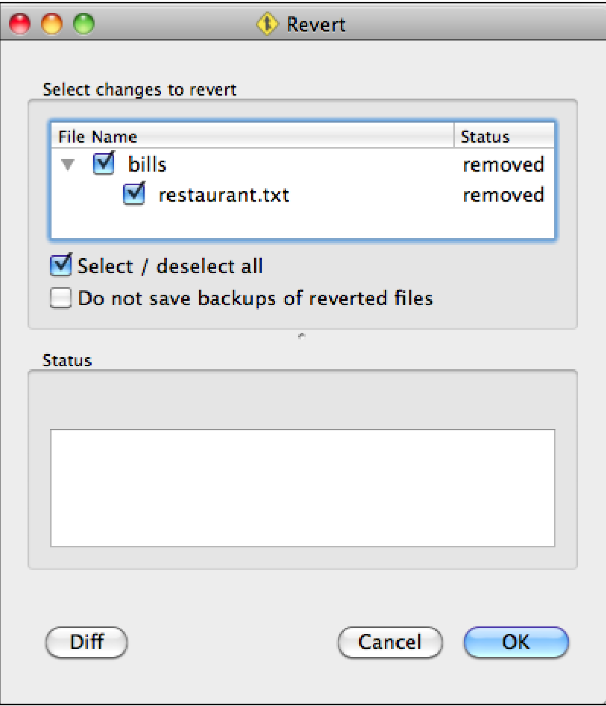
```
$ bzr revert menu.txt
+N  menu.txt
```

As a result, the `menu.txt` file is now back in the working tree, in the same state as it was as of the last revision. The `+N` in front of the name of the file in the output of the command indicates the action performed to restore the file; in this case, `add new file`.

When specifying a directory, Bazaar will revert the changes in all the files and subdirectories within that directory. This is especially useful when you want to revert only some of the changes selectively, rather than the entire working tree. You can specify multiple files and directories at the same time.

Using Bazaar Explorer

To revert changes using Bazaar Explorer, click on the large **Work** icon in the toolbar and select the **Revert Working Tree...** option. This opens up the **Revert** view:



By default, no file is selected, and for good reason. The revert operation can be considered unsafe, as the changes you revert may be lost. Anything that was once committed to version control is safe, because you can always return to a past revision. But if you revert a pending change that was never recorded in a revision, it will be lost.

Select the files to revert and click on **OK**. The **Status** box will show the `bzr` command that was executed along with its output. For example:

```
Run command: bzr revert bills bills/restaurant.txt
+N bills/
+N bills/restaurant.txt
```

Click on **Close** to dismiss the **Revert** view, and return to the **Status** view.

Editing files

You can edit files in a working tree in the same way as you always do. You don't need `bzr` commands or Bazaar Explorer, as you can use your editors or development tools and make changes to the project files. Bazaar will automatically detect when the content of some files has changed since the last revision, and will show it in the status command or Bazaar Explorer.

Let's go ahead and make some changes:

- In `guests.txt`, the names are in alphabetical order, except `Jason`. Let's fix that by moving the line up right after `James`.
- Also, in `guests.txt`, `Franck` should have been `Frank`. Let's fix that too.
- In `menu.txt`, let's add `Tacos`.

Using the command line

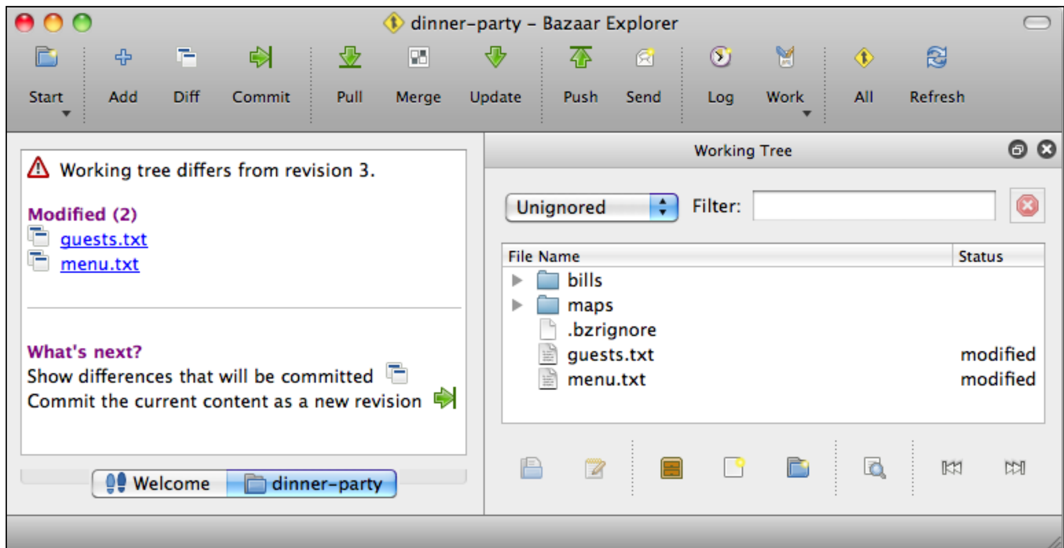
Having edited some files, let's check the status of the working tree:

```
$ bzr status
modified:
  guests.txt
  menu.txt
```

At this point, we can either commit or revert the changes, one by one, or all at once. In general, it is a good idea to commit changes often. Changes recorded in a revision are safe, as you can always return to them. Pending changes may get deleted by accident, and if they have never been committed in a past revision, they cannot be recovered.

Using Bazaar Explorer

In Bazaar Explorer, modified files are listed in the left panel of the **Status** view. In the right panel, the **Status** column should show the value `modified` for these files:



You can click on a file in the left panel to open it using the default text editor. Another way to edit files is by selecting them in the right panel and clicking on the **Edit** button in the lower toolbar.

Occasionally, the right panel may not be up-to-date when the working tree is edited outside of Bazaar Explorer. You can fix this by clicking on the large **Refresh** button in the toolbar.

Viewing differences in changed files

Viewing differences is one of the coolest features in a version control system. It is especially useful when working with large files, as it highlights the changed portion of files, so you don't overlook any changes accidently.

However, viewing the differences works only with plaintext files, such as `*.txt` files, program source code, and script files. These kind of files can be compared line by line. Binary files, such as Word, PowerPoint documents, images, or other binary data files cannot be compared on a line by line basis. Therefore, showing differences in these files is problematic.

To see how Bazaar shows differences to binary files, let's make changes to the image files in the `maps/` directory of our example project:

- Copy `maps/map1.png` to `maps/map2.png` (overwriting `maps/map2.png`)
- Copy `maps/venue1.jpg` to `maps/venue2.jpg` (overwriting `maps/venue2.jpg`)

Using the command line

You can view the differences using the `bzr diff` command. Without parameters, it will show the differences in the entire working tree. To view the differences only in some of the files, you must specify them explicitly as parameters. For example:

```
$ bzr diff guests.txt
=== modified file 'guests.txt'
--- guests.txt  2013-02-10 20:46:39 +0000
+++ guests.txt  2013-02-11 21:17:27 +0000
@@ -1,12 +1,12 @@
 Guest list
 ==========
Anna
-Franck
+Frank
 Gregory
 Jack
 James
+Jason
 Martin
 Mike
 Peter
 Pierre
-Jason
```

The difference is shown in unified diff format, the same as used by the patch tool in Unix and similar systems. For each file, there is a header section describing the change (in this example `modified file`), and the filename before and after the change (in this case, the same `guests.txt`). Deleted lines are prefixed with a "`-`"marker, added lines are prefixed with a "`+`" marker, and a few lines before and after the changes are shown for reference, called `context`. Modified lines are represented as deleted lines followed by an added line. Reading differences in this format can be challenging at first, but once you get used to it, it can be really useful.

If you have installed the **bzrtools** plugin (included by default in the Windows and Mac OS X installers), and if your terminal program supports colors, then a helpful alternative is the `bzr cdiff` command, which highlights the differences using colors. For example:

```
Default
$ bzr cdiff guests.txt
=== modified file 'guests.txt'
--- guests.txt  2013-04-20 15:45:45 +0000
+++ guests.txt  2013-04-23 06:33:28 +0000
@@ -1,12 +1,12 @@
 Guest list
 ==========
 Anna
-Frank
+Franck
 Gregory
 Jack
 James
-Jason
 Martin
 Mike
 Peter
 Pierre
+Jason

$
```

In case of binary files, for example the image files in the `maps/ directory` we changed, it is not possible and would not make much sense to see the differences line by line, as Bazaar only shows that files have changed. For example:

```
$ bzr diff maps/
=== modified file 'maps/map2.png'
Binary files maps/map2.png      2013-02-16 18:17:48 +0000 and maps/map2.
png     2013-02-17 03:07:58 +0000 differ
=== modified file 'maps/venue2.jpg'
Binary files maps/venue2.jpg    2013-02-16 18:17:48 +0000 and maps/
venue2.jpg  2013-02-17 03:09:01 +0000 differ
```

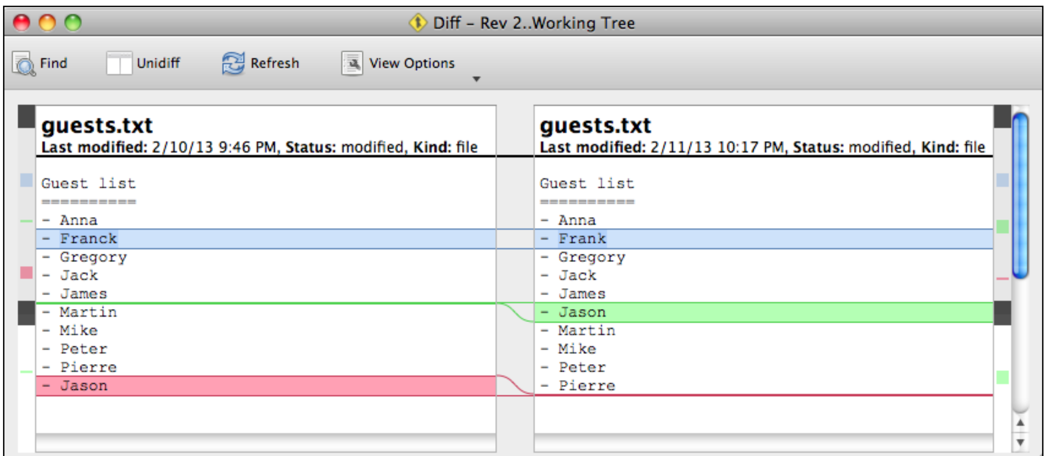
Notice that since we specified a directory as the parameter of `bzr diff`, Bazaar shows the changes in all files and subdirectories under that directory.

Using Bazaar Explorer

Bazaar Explorer has a built-in difference viewer tool that is much more user friendly than the command-line interface. There are multiple ways to launch the difference viewer in the **Status** view:

- In the left panel, click on the icon at the left of the modified files
- In the right panel, select one or more modified files, then right-click and select **Show differences**
- Click on the large **Diff** button in the toolbar to view all the changes in the entire working tree

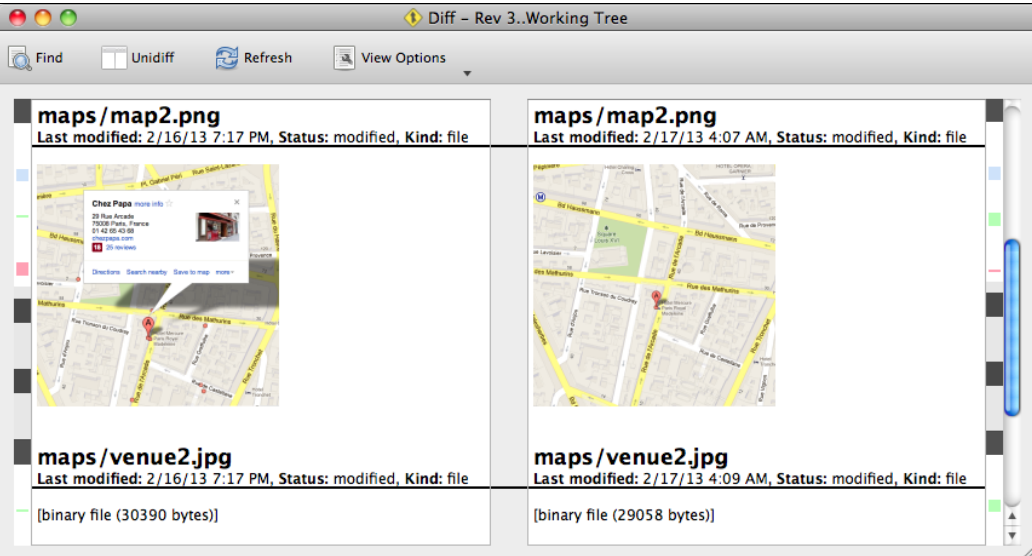
Whichever way you choose, the **Diff** view will open showing differences to one or more files. Let's choose the third option and click on the large **Diff** button in the toolbar:



Similar to the command-line tool, for each file there is a header section describing the change and the timestamp of the last modification. However, the added, deleted, and modified lines are easier to see in a side-by-side comparison:

- Modified lines are shown with a blue background, with a darker shade of blue highlighting the changed parts.
- Added lines are shown with a green background at the right, clearly indicating the position in the original file (at the left), where they were inserted. In case a file was added to the project, it will be shown entirely in green, as all the lines have been added compared to a "blank" state.

- Deleted lines are shown with a red background at the left, clearly indicating the position in the changed file (at the right) where they were deleted. In case a file was deleted from the project, it will be shown entirely in red, as all the lines have been deleted resulting in a "blank" state.
- In case of binary files, since the changes cannot be shown line by line, the tool only shows the fact that the files have changed, and the difference in their size.
- In case of image files in supported formats, the versions before and after the change are shown in the left and right panels for side-by-side comparison. This works with PNG files; it does not work with JPG, but might work with other formats too:



The buttons at the top have several options to adjust the view; for example, to show the entire files and not only their changed parts, ignored whitespaces, and other options. Sometimes, you may also want to drag the vertical separator between the left and right sides to adjust their width.

Checkpoint

Let's record a new revision at this point, committing the changes to `guests.txt`, `menu.txt`, and the images in the `maps/` directory. The changes to the image files are not meaningful, and normally we should not commit them, but go ahead with this purposefully bad commit, so that we can demonstrate how to roll back this change later, and restore the images to their good version.

Renaming or moving files

When changing the names of files and directories, or moving them around within the project, you have to tell these actions to Bazaar explicitly, so that it can track them. In Bazaar terms, both renaming and moving are the same kind of action, called **rename operation**.

> Handling renames and moves is one of the unique features of Bazaar as compared to other version control systems, as argued by *Mark Shuttleworth* in his article at `http://www.markshuttleworth.com/archives/123`.

If you rename files without using a Bazaar operation, for example, by drag-and-drop in your filesystem explorer, then Bazaar will not know what happened and it will interpret the change in the working tree as if "a file has disappeared and now there is a new unknown file". To fix this, you can either manually undo the rename and redo it using Bazaar, or use a Bazaar command (see below) to tell Bazaar about an already performed rename.

Using the command line

You can rename or move files using the `bzr mv` command. For example:

```
$ bzr mv maps/ images
maps => images
```

The directory is instantly renamed in the working space, and using the `status` command, we can confirm that Bazaar is aware of this change:

```
$ bzr status
renamed:
  maps/ => images/
```

If we rename a file or directory without telling Bazaar explicitly, it will not be aware of the change. For example:

```
$ mv menu.txt test.txt
$ bzr status
removed:
  menu.txt
renamed:
  maps/ => images/
unknown:
  test.txt
```

We have renamed `menu.txt` to `test.txt` using shell commands instead of a Bazaar operation. Comparing the current state of the working tree with the last revision, Bazaar interprets it as though the `menu.txt` file has disappeared and now there is a new unknown file `test.txt`.

One way to correct this is to rename `test.txt` back to `menu.txt`, and then rename it properly using `bzr mv`.

Another way is to use `bzr mv` with the `--after` flag, which tells Bazaar that the file itself has been renamed already, and we just want to register the Bazaar operation, as follows:

```
$ bzr mv menu.txt test.txt --after
menu.txt => test.txt
$ bzr status
renamed:
  maps/ => images/
  menu.txt => test.txt
```

Our earlier mistake is now fixed, and the rename operation is correctly registered.

To commit or revert specific rename operations (as opposed to committing or reverting all changes in the working tree), you can specify either the original name or the new name of the renamed files or directories. For example, in this case to revert the renaming of `menu.txt`:

```
$ bzr revert test.txt
R   snacks.txt => menu.txt
```

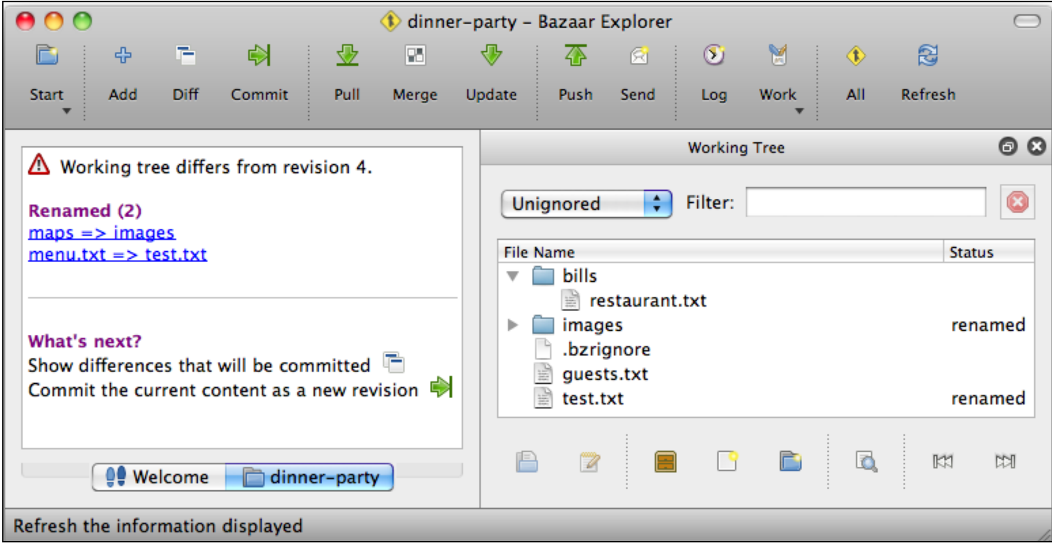
If there have been multiple renames without telling Bazaar, it can be troublesome to correct them one by one. In this case, instead of using `--after`, another option is to use the `--auto` flag, as follows:

```
$ mv menu.txt test.txt
$ bzr mv --auto
menu.txt => test.txt
```

That is, we again renamed `menu.txt` without telling Bazaar about it, and fixed it by using the `--auto` flag. However, this may not work every time, especially if you have made changes to the renamed files. As the documentation (`bzr mv --help`) says, Bazaar only "guesses" renames; it cannot guarantee 100 percent correctness.

Using Bazaar Explorer

To rename a file or directory in Bazaar Explorer, right-click on it and select **Rename** to edit the name. To move a file or directory to a different subdirectory, simply drag-and-drop. Renames are shown in the left panel:



Checkpoint

Let's record a new revision, but selectively. Renaming `maps/` to `images/` makes sense, as the image files in that directory are not all maps, but include photos. On the other hand, the renaming of `menu.txt` to `test.txt` is completely meaningless and was for our experimenting only, so let's revert that.

Viewing the revision history

Viewing and searching the revision history is very useful to retrace our steps, and critical to find specific revisions that you might want to restore.

Using the command line

You can view the list of past revisions using the `bzr log` command. Without parameters, it shows all the revisions in the branch, or if you specify a list of files or directories, then it will show only the revisions that affected those files. For example:

```
$ bzr log bills/
------------------------------------------------------------
revno: 1
committer: Janos Gyerik <janos@axiom>
branch nick: dinner-party
timestamp: Sat 2013-02-16 19:06:44 +0100
message:
  added bills
```

By default, Bazaar uses the long format, which displays quite detailed information about each revision:

- Revision number
- Name and email address of the committer
- Timestamp
- Commit message

The short format is a bit more compact with less details, namely the revision number, name of the committer, date, and message:

```
$ bzr log bills/ --short
    1 Janos Gyerik      2013-02-16
      added bills
```

The line format is even more brief, with a single line per revision:

```
$ bzr log --line
5: Janos Gyerik 2013-02-17 renamed maps/ to images/
4: Janos Gyerik 2013-02-17 correction in text files and map images
3: Janos Gyerik 2013-02-16 added ignore patterns
2: Janos Gyerik 2013-02-16 added guest list, menu and map files
1: Janos Gyerik 2013-02-16 added bills
```

In addition to these formatting options, the command has a number of other useful options, such as:

- `-v`: This is a verbose output, which shows the files that were changed in each revision, except when using the single line format (with `--line`).
- `-r ARG` **or** `--revision=ARG`: This shows the log of specified revisions. One way to specify revisions is by using the revision number. See the *Specifying revisions* section in this chapter for more details.

www.allitebooks.com

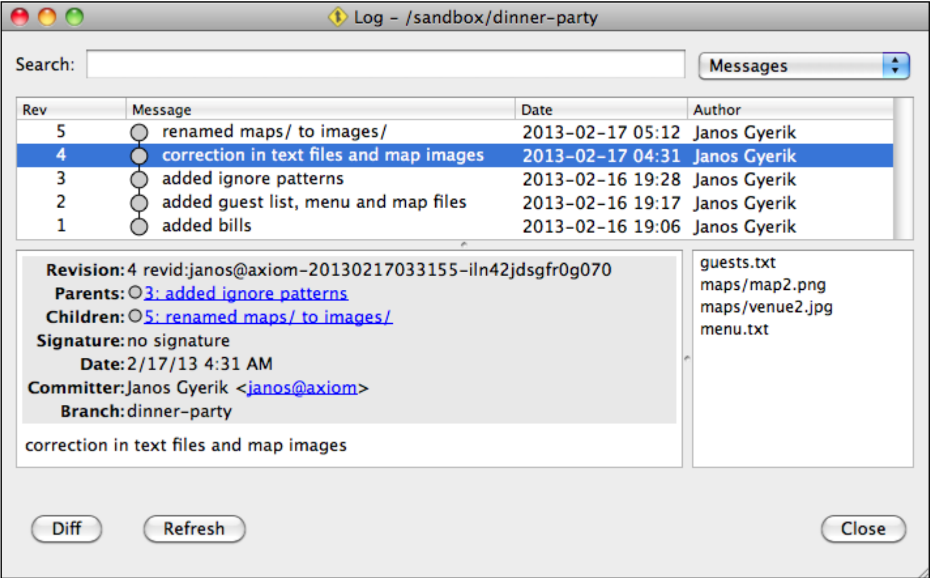
- `-l N` **or** `--limit N`: This shows at most N revisions, which is useful when you have thousands of revisions but are only interested in the recent ones.
- `-m ARG` **or** `--match=ARG`: This shows only those revisions whose properties (for example, message or author) match ARG (case-insensitive).
- `--match-message=ARG`: This shows only those revisions whose log message matches ARG (case-insensitive).
- `--match-author=ARG`: This shows only those revisions whose author matches ARG (case-insensitive).

You can combine all these options. For example:

```
$ bzr log --short -v --match-author janos --match-message added --limit 1
--revision 2..3
    3 Janos Gyerik      2013-02-16
      added ignore patterns
      A  .bzrignore
```

Using Bazaar Explorer

Viewing the history of changes is typically easier in Bazaar Explorer, as you can adjust filters and see the result immediately, and the different viewers can be accessed from it directly. To open the **Log** view, click on the large **Log** button in the toolbar:



The view opens showing all the revisions in the branch, with the basic meta information in the columns—revision number, message (as much as it fits within one line), date, and author.

If you click on a revision, the box at the bottom-left shows more details about the revision, including the full text of the commit message, and more. The box at the bottom-right shows the list of files that were affected in the selected revision. Double-click on a file to open the **Diff** view, to see the changes to that file in the selected revision.

To filter the list of revisions, enter keywords and search patterns in the **Search** box at the top and adjust the combo box at the right, depending on what you want to filter by; for example, by message text or author name.

Click on the **Diff** button at the bottom-left to open the **Diff** view, to see all the changes in all the files in the selected revision.

Restoring files from a past revision

We have seen earlier that the revert operation can restore files in the working tree to their state as of the last revision, effectively undoing their pending changes.

The same operation, with additional parameters, can also be used to restore files to their state of any past revision, not only the latest. Note that when used this way, the affected files in the working tree will be in a changed state, as by definition their content will have changed compared to the last revision recorded in the branch. As such, with usual and pending changes like these, you will have the option to either commit and record a new revision or revert (without the `revision` parameter) to restore to the state of the last revision.

Using the command line

You can revert a set of files to their state as of any past revision by using the `bzr revert` command and specifying the revision using the `-r` or `--revision` option. Without parameters, it will restore the entire working tree to the specified past state. To restore only a set of files, specify the files you want to restore. For example:

```
$ bzr revert -r2 guests.txt
 M  guests.txt
$ bzr status
modified:
  guests.txt
```

As a result, the file is now in a changed state, and if you look at its content, you will see that the modifications we did earlier have been reversed.

If the earlier version of this file was really what we wanted in the project, we can commit this change now, so that the latest version of the project is as it should be. Before committing, we can also further modify the file manually, if necessary. Otherwise, if you don't want to commit this, then you can revert the change as usual to restore the file back to its state as of the last revision.

In this example, we have used a revision number to revert to. This is one way of specifying revisions. See the *Specifying revisions* section later in this chapter for more details.

The right revision number to revert to, of course, depends on your project and its revision history. Use the `bzr log` command or the **Log** view in Bazaar Explorer to find the right revision.

Using Bazaar Explorer

To restore a file to a previous version, open the **Log** dialog box, find the right revision, perhaps using filters, right-click on the file in the box in the bottom-right corner, and select **Revert to this revision**.

If you want to restore multiple files or entire subdirectories to a past revision, it might be easier to use the command line. In this case, Bazaar Explorer can still be useful in helping to find the right revision number to revert to.

Putting it all together

The previous sections should get you started with using the various basic operations, but there are a couple of things not tied to any one specific operation that might be worth clarifying here. So, you understand Bazaar better and become more comfortable with it.

Making different kinds of changes

In the previous sections, we did only one type of change at a time—added, deleted, edited, and renamed files. You can, of course, make many different kinds of changes at once and commit them all together. While doing so, it helps to have a solid understanding of the output of status and revert commands, so we will review them here.

Understanding the output the revert command

To start with a clean state, first revert all the pending changes in your example project using `bzr revert` with no parameters. Next, let's revert to revision 2 and try to interpret the output:

```
$ bzr revert -r2
-D  .bzrignore
 M  guests.txt
R   images/ => maps/
 M  maps/map2.png
 M  maps/venue2.jpg
 M  menu.txt
```

For each file, the output of `revert` indicates the change Bazaar did to restore its state:

- `-` means a delete operation, and `D` following it means that the file was removed from the working tree
- `M` means the file was modified
- `R` means the file was renamed

If you revert again without any parameters, you will see the reversal of these operations:

```
$ bzr revert
+N  .bzrignore
 M  guests.txt
R   maps/ => images/
 M  images/map2.png
 M  images/venue2.jpg
 M  menu.txt
```

No surprises for modifications and renames, as their reverse is the same kind of Bazaar operation in both directions. The only difference is `.bzrignore`, `"+"` means an add operation, and `N` following it means that the file was added in the working tree.

Understanding the output of the status command

Let's revert again to revision 2 and see the output of the `status` command:

```
$ bzr status
removed:
  .bzrignore
renamed:
  images/ => maps/
modified:
  guests.txt
  maps/map2.png
  maps/venue2.jpg
  menu.txt
unknown:
  .DS_Store
  Thumbs.db
```

The output is more verbose and trivial to interpret, with the different kinds of changes divided into separate clearly identified groups.

It is always good to check the status right after a `revert` operation, because the output of `revert` does not always tell the full story. In this example, as a consequence of the deleted `.bzrignore` file, the `.DS_Store` and `Thumbs.db` files are no longer ignored, and now listed as unknown.

Let's revert again without parameters to return to the state of the last revision. After that, if you run `bzr status` again, the output will be empty, as there should be no pending changes.

Understanding the backup files created by Bazaar

When a Bazaar operation cannot be performed safely, Bazaar creates backup files to protect you from losing data. For example, when you revert a file that has been changed, the changes would be lost. Let's demonstrate this by appending a line to `menu.txt` and then reverting it:

```
$ echo hello >> menu.txt
$ bzr revert menu.txt
 M  menu.txt
```

If you now check the working tree, a new file named `menu.txt.~1~` will be created, and if you look inside, it will contain the line we appended to it. The number in the extension `~1~` is incremented when another backup of the same file is needed. For example, if you repeat the preceding commands, append a line and revert, then a new file named `menu.txt.~2~` will be created, and so on. This way Bazaar tries to protect you from losing data.

These backup files are ignored by Bazaar commands—thanks to a global ignore rule—so you don't see them in the output of status or in Bazaar Explorer. You can find them all using the `bzr ignored` command:

```
$ bzr ignored
.DS_Store                                              .DS_Store
Thumbs.db                                              Thumbs.db
menu.txt.~1~                                           *~
```

In any case, these backup files are normally visible in your filesystem. If you are sure you don't need them, then you can delete them in the same way you normally delete files, without using Bazaar commands.

Understanding the .bzr directory

In our example project, there is a single `.bzr` directory in the top-level directory of the project. All of Bazaar's data is stored there, and normally you do not need to look inside or understand the contents of this directory, except certain advanced features, which will be explained in the later chapters.

As long as this directory is intact, no matter what happens to the files and directories in the working tree, you can always restore its state to the latest or any past revision. Conversely, if you delete this directory, it will delete all of Bazaar's data, and the working tree will become a regular directory in your filesystem.

How often to commit?

There is no one-size-fits-all rule, but consider this—any changes that are not committed in a revision are not recoverable if the files are lost. For this reason, it is good to not keep pending changes around for too long, and commit often. Also, logically-related changes are usually committed together, and it is normal to include all kinds of changes at the same time, such as modifications, renames, and deletions.

Beyond the basics

In the previous section, we focused on the essential operations and most common use cases. In this section, we will go a step further and give you additional tips on using the command-line interface, explain more options of the basic commands, and introduce a few more useful new commands.

Mastering the command line

The command-line interface of Bazaar has an excellent built-in help system, and all commands behave in a predictable, consistent way. Here, we highlight a few simple tips that should greatly improve your experience with the command-line interface.

Common flags

A few flags are supported by all Bazaar commands:

- `-h`, `--help`: This shows the help message. Also, `bzr somecmd -h` and `bzr help somecmd` are equivalent.
- `-v`, `--verbose`: This shows the verbose output and displays more information than usual. Sometimes specifying the flag multiple times results in increased verbosity.
- `-q`, `--quiet`: This displays only errors and warnings.
- `--usage`: This shows usage messages and options.

Common behavior in all the commands

Flags and command-line parameters can appear in any order. For example, the following are equivalent:

```
$ bzr log --line -r1 file.txt
$ bzr log file.txt --line -r1
$ bzr log -r1 file.txt --line
```

Putting a space between a flag and its parameter is optional. For example, the following are equivalent:

```
$ bzr log -r1
$ bzr log -r 1
```

When using the long version of flags, for instance `--revision` instead of `-r`, the equal sign is optional. For example, the following are equivalent:

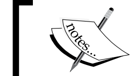
```
$ bzr log --revision=1
$ bzr log --revision 1
```

Using shorter aliases of commands

Many commands have shorter (or possibly more intuitive) aliases that can be used equivalently, for example:

- `bzr st` and `bzr stat` are the same as `bzr status`
- `bzr ci` and `bzr checkin` are the same as `bzr commit`
- `bzr rm` and `bzr del` are the same as `bzr remove`
- `bzr di` and `bzr dif` are the same as `bzr diff`
- `bzr move` and `bzr rename` are the same as `bzr mv`

The list of built-in aliases of each command is usually near the end of the help and usage messages.



In a later chapter, we will show how to define your own custom aliases.

Quick reference card

The Bazaar documentation includes very helpful quick references:

- `http://doc.bazaar.canonical.com/bzr.dev/en/_static/en/bzr-en-quick-reference.png`
- `http://doc.bazaar.canonical.com/bzr.dev/en/quick-reference/index.html`

Using tags

With tags, you can give revisions a meaningful name, which can be especially useful to identify past milestones, or revisions that you frequently make references to for some reason. Once you assign a tag to a revision, you can refer to that revision using `-rtag:somename` in the various `bzr log` and `bzr diff` commands.

- `bzr tag v2.6`: This assigns the tag "v2.6" to the current revision
- `bzr tag v2.6 -r117`: This assigns the tag "v2.6" to revision 117
- `bzr tag v2.6 -r119 --force`: This reassigns the tag "v2.6" to revision 119
- `bzr tag v2.6 --delete`: This deletes the tag "v2.6"
- `bzr tags`: This shows all the tags in the current branch

Tags are stored in a branch, and are propagated in the various branch operations such as merge, push, and pull, which will be explained in the next chapter.

Tags must be unique within a branch. If you try to assign the same tag to a different revision, `bzr` will return an error. In this case, you can either delete the tag and recreate with a different revision, or use the `--force` flag, as in the preceding example.

In general, it is not a good practice to delete or reassign tags, especially when working together with others. As such, it is best to use unique tag names that will never need to be reassigned; generic names such as `stable` or `testing` should be avoided.

Specifying revisions

Many Bazaar commands accept a `revision` parameter, which can be specified using the `-r` and `--revision` flags.

In the most simple cases, revisions can be specified by their numbers, as shown in the output of the log command or in Bazaar Explorer, but there are many other interesting formats that are useful to know, such as using dates, tags, and other special symbols.

You can find the complete documentation of revision specification formats in `bzr help revisionspec`, where we highlight only a few interesting examples.

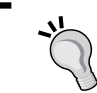
Specifying a single revision

In addition to the revision number, you can specify revisions by date, tags, and other special symbols. For example:

- `bzr log -rdate:yesterday`: This selects the first revision since yesterday. You can also use `today` or `tomorrow` similarly.
- `bzr log -rdate:2013-02-17`: This selects the first revision since 2013-02-17.
- `bzr log -rdate:2013-02-17,04:01:12`: This selects the first revision since 2013-02-17, 4 AM 1 minute 12 seconds.
- `bzr log -rtag:v2.6`: This selects the revision named by the tag "v2.6".
- `bzr log -rbefore:date:today`: This selects the first revision before the specified date.
- `bzr log -rbefore:3`: This selects the first revision before the specified revision number.

- `bzr log -rlast:1`: This selects the last revision.
- `bzr log -rlast:2`: This selects the second to last revision.
- `bzr log -r-1`: This selects the last revision.
- `bzr log -r-2`: This selects the second to last revision.

Actually, in most cases the `date` and `tag` keywords can be omitted; Bazaar can figure out that you are referring to dates and tags. This is especially useful when using the `before:` keyword, thus allowing a shorter expression.



The latest revision is often called the `HEAD` or the `tip` of the branch.

Specifying a range of revisions

Depending upon the command, a range of revisions may make sense, and can be specified using the `-r` and `--revision` flags in the format `N..M`. For example:

```
$ bzr log -rbefore:today..last:1
```

The revisions N and M can be any valid single revision specifier, and both N and M may be omitted. For example these are all valid revision ranges:

- `bzr log -r2..4`: This selects revisions 2 to 4
- `bzr log -r2..`: This selects revision 2 and all the revisions thereafter
- `bzr log -r..4`: This selects all the revisions up to and including 4
- `bzr log -r..`: This selects all the revisions

However, be warned that depending upon the Bazaar operation, ranges may be interpreted differently.

In case of the `log` command, the range is a sequence of log messages, and the range is considered closed. That is, `range 2..4` includes revisions 2, 3, and 4. Also, the end revision must be higher than the start revision.

In contrast, in case of the `diff` command, the range is a change between revisions, and is considered open-ended, excluding the beginning of the range. That is, `range 2..4` includes the changes done in revisions 3 and 4 but not in 2. The range can also be reversed, with the start revision higher than the end revision. In this case, the direction of changes is also reversed. For example, `range 4..2` includes the changes done in revisions 3 and 2 but not in 4, as if changing the project's state from revision 4 to revision 2.

Viewing differences between any two revisions

We have already seen how to view the differences of pending changes in the working tree that are not yet committed to the repository. Another very common need is to view the differences between any two revisions, or similarly, a past revision and the working tree.

In all the comparison operations, you can specify a list of files and directories to see only the subset of all changes involving those files. If you do not specify any file parameters, Bazaar will display all the changes between the two compared revisions.

At the time of this writing, the **Diff** view of Bazaar Explorer does not have a user interface to choose the revisions to compare. A workaround is to launch the **Diff** view from the command line using the `bzr qdiff` command. For all examples in this chapter, you can simply replace all occurrences of `bzr diff` with `bzr qdiff` to view the output in Bazaar Explorer's more friendly format, rather than in a terminal window.

Viewing differences between any revision and the working tree

To compare a past revision and the current state of the working tree, run the `diff` command and specify the past revision. For example:

```
$ bzr diff -r3 menu.txt
=== modified file 'menu.txt'
--- menu.txt    2013-02-16 18:17:48 +0000
+++ menu.txt    2013-02-17 23:16:51 +0000
@@ -15,3 +15,4 @@
 - Beef burrito
 - Mixed burrito
 - Onion soup
+- Tacos
```

An important detail that is easy to overlook is that the `diff` command shows the changes that were recorded after the specified revision; it does not include the changes in the revision itself. Take a look at the following example:

```
$ bzr log --line menu.txt
4: Janos Gyerik 2013-02-17 correction in text files and map images
2: Janos Gyerik 2013-02-16 added guest list, menu and map files
$ bzr diff -r4 menu.txt
$
```

That is, in our sample project, `menu.txt` was last modified in revision 4, but these changes will not be included in the `diff` command output if we specify revision 4; for that, we will have to specify revision 3.

Another important point is that when using the `diff` command this way, it compares the past revision not with the latest revision, but with the current state of the working tree. In other words, the pending changes in the working tree will be part of the output.

Specifying the last revision is equivalent to not specifying a revision at all, comparing the last revision with the current state of the working tree.

Viewing differences between any two revisions

To view the differences between two revisions, you must specify the revisions as a range using the `-r` or `--revision` flags, in the format `START..END`. The `diff` command will show the changes it would take to go from revision `START` to revision `END`.

It is important to remember that the changes in revision `START` itself will not be included in the output, because it does not fall within the definition of how the `diff` command works. To include changes in that revision, you must specify a previous revision, for example `START-1`, or by using the `before` keyword like this: `before:START`.

The order of the two revisions in the range is significant. If the range is `N..M`, the `diff` command will show the changes going from `N` to `M` (not including the changes in revision `N` itself), and if you reverse the end points of the range, then the `diff` command will show the changes going from `M` to `N` (not including the changes in revision `M` itself).

Viewing differences going from one revision to the next

This is a special case of comparing any two revisions, using a range of two revisions, where the end revision is equal to `start+1`. A convenient shortcut for this case is the `-c` or `--change` flag, that specify the revision whose changes we're interested in. For example, you can see the changes in revision 4 as follows:

```
$ bzr diff -c4 menu.txt
```

This is equivalent to:

```
$ bzr diff -r3..4 menu.txt
```

This is also the same as viewing the `diff` output of selected files in selected revisions in the **Log** view of Bazaar Explorer.

Cloning your project

Keeping backups is always a good idea. Using version control for your project gives you a lot of safety already—any accidental changes can be restored from a past revision. However, if your hard disk crashes the `.bzr` directory, your entire repository will be lost with it too.

The `bzr push` command is normally used when working with local or remote branches, which will be explained in later chapters. Incidentally, it is also an excellent and very simple way to take a backup. For example:

```
$ bzr push /media/backups/dev/dinner-party --create-prefix
Created new branch.
```

This command creates a perfect clone of the working tree as a new branch at the specified location. The path can be any suitable location where you would like to put the clone, ideally a different hard disk. The `--create-prefix` flag is useful if the parent directory of the specified path does not exist, otherwise the flag is optional and will do nothing.

Once you created the clone, you can re-run `bzr push` again at any time to copy the new revisions to it. You don't need to specify the path again, Bazaar remembers your original push target.

Summary

In this chapter, you have learned the core concepts and basic commands of Bazaar, including everything you might need when working solo on a single branch on your PC. In particular, now you know how to:

- Put any directory under version control
- Add, delete, and modify files
- Check the status of the working tree and view differences
- Revert or commit changes
- View the revision history
- Restore files from a past revision

In the next chapter, we will learn how to work with branches, which will open up a whole new range of possibilities to experiment freely, without letting ideas fade away unimplemented. Branches do not mean complexity, and being able to use them effectively can lead to more stable projects and improved productivity.

3

Using Branches

This chapter will explain how to work with multiple branches. We will continue to build on the simplistic setup of the previous chapter—working solo on a private project, which exists only on your computer. We will add to that a new angle, using multiple branches instead of just one.

In a solo project, using multiple branches opens many interesting new possibilities. This can greatly improve your productivity and change the way you work. In larger projects, with many collaborators, using branches is absolutely essential, as it is the primary way of combining efforts.

The following topics will be covered in this chapter:

- What is a branch?
- What you can do with branches
- Why use more than one branch
- Understanding core terms
- Using a shared repository
- Basic branching and merging
- Using the `branch` command
- Viewing basic branch information
- Comparing branches
- Merging branches
- Mirroring branches

What is a branch?

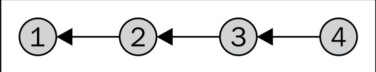
In Bazaar, a `branch` is an ordered set of revisions up to a specific latest revision, commonly called the **tip**. In the most simple cases, revisions follow one another in a timely order. In such cases, there is only one branch and the history is linear.

However, working in a linear manner can be limiting. Branches make it possible to work on different tasks, or different implementations of the same task at the same time. This effectively makes the history non-linear, bringing many practical benefits.

If you think of a project as a story, a branch is like an alternative ending. At any time, you can create a branch based on any of the revisions in the history, and start working in a different direction.

A single branch with a linear history

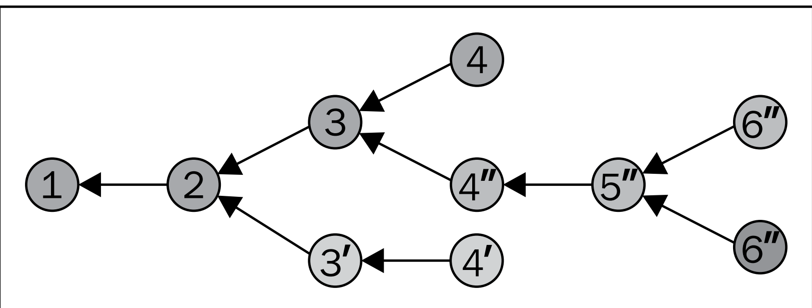
When working on a single branch, revisions simply follow one another in a timely order, resulting in a linear history:



This is a simple, well understood way of working, but it can be limiting sometimes. It is natural to have multiple versions of a project existing simultaneously, for example stable and unstable, with different changes going into both in parallel. However, this is not possible using a linear history.

Multiple diverged branches

When working on multiple branches, the revision history diverges in different directions:



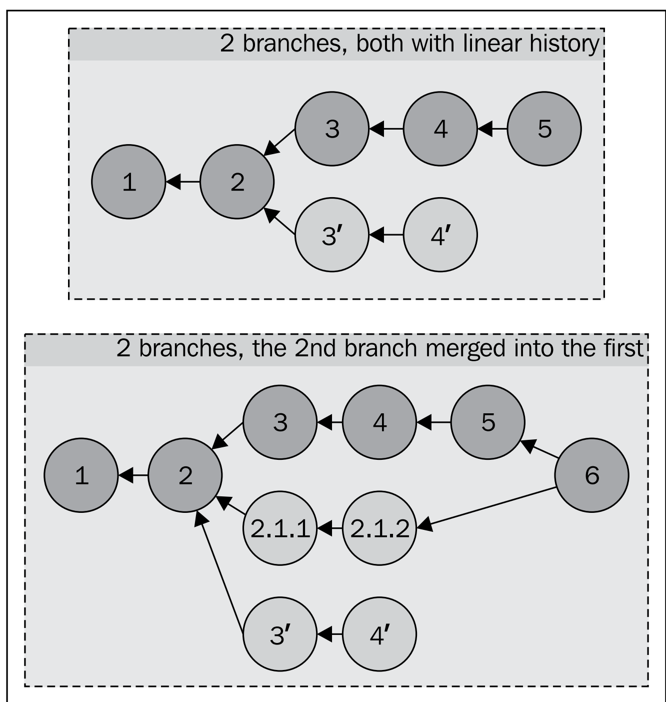
After revision 2 a new branch was started (adding revisions 3' and 4'), then after revision 3 of the first branch, yet another branch was started (adding revisions 4'', 5'',...), and so on. At this point, the project has 4 branches, and therefore 4 possible latest revisions: 4, 4', 6'' and 6'''.

The rule of numbering revisions is still the same—when recording a new revision in a branch, increment the last revision number by one. Revision numbers are unique per branch, but they are not unique globally in the project. As such, at this point it is not clear what is the latest version of the project.

Using multiple active branches in parallel for different purposes is very common, for example, to separate the stable and unstable versions of a project, or to work on multiple features and bug fixes at the same time, but isolated from each other.

Branches with non-linear history

When merging a branch into another, the historical ordering of the merged revisions is preserved, resulting in a non-linear history:

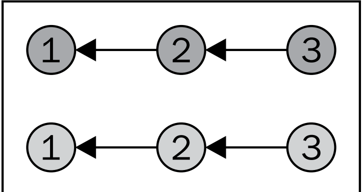


The project diverged after revision 2—while work continued in the first branch, a different set of changes started in another branch. After revision 5, the first branch merged from the second branch, recording this change as revision 6. As a result, the history of the first branch is no longer linear, as now it includes the unique revisions of the other branch with their historical ordering preserved.

To keep revision numbers unique within the first branch, the merged revisions were renamed using a dotted notation instead of simply an integer—revisions 3' and 4' in the second branch were renamed to `2.1.1` and `2.1.2`, respectively, in the first branch. This numbering logic will be explained in detail later in this chapter.

Unrelated branches

If two branches have no common ancestors, that is, no common base revision that they diverged from, then they are considered as **unrelated branches**:

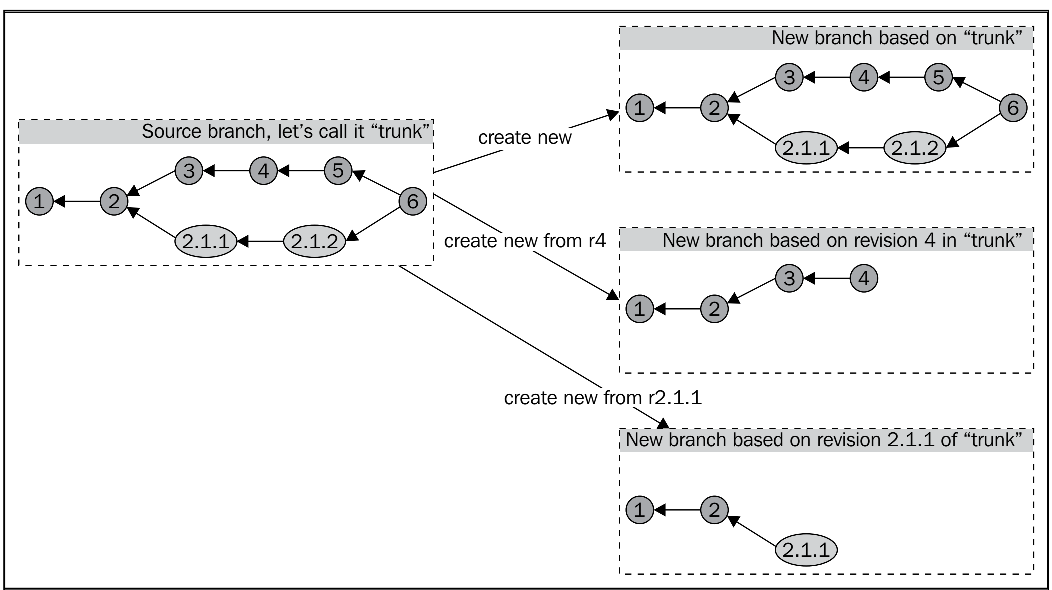


What can you do with branches?

Being able to create branches easily is great, but that in itself is not the greatest benefit. The real power lies in what you can do with multiple branches, how branches can interact with each other, and most importantly, how you can combine them to bring all the work done in the various branches into the main line of development of the project.

Creating branches

You can create a new branch based on any revision of an existing branch, often the latest revision. The new branch inherits the revision history up until the selected revision. The new branch is completely independent of the original branch; you can commit new revisions in it and create further branches based on it:



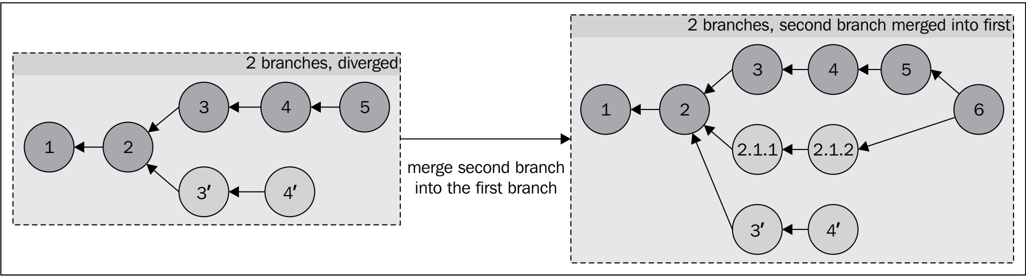
Comparing branches

Just like you can compare any two revisions within the same branch, you can also compare any two revisions across any two branches.

For a higher level of comparison (without the detailed differences), it can sometimes be useful to list the revisions (along with their basic information) that exist only in one branch but not in the other.

Merging branches

You can merge one or more branches into another branch. As a result, the merged revisions become part of the target branch, with their historical ordering correctly preserved as they were committed in their original branches:

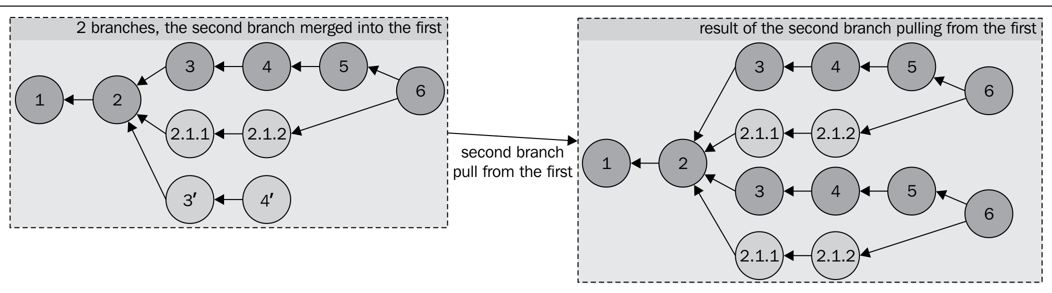


Mirroring branches

It is often useful to create mirror branches that are exact replicas of the original branch. Mirroring is used extensively to replicate local branches at another location, in order to make them accessible by other team members, or as a way of creating backups.

Mirroring can work in two directions—a **push** operation creates, converts, or updates another branch to be a mirror of the current branch, while a **pull** operation converts or updates the current branch to be a mirror of another branch. Both operations copy missing revisions and update the history of the target branch to be identical to the source branch.

Mirroring works only between branches that have not diverged. It is only meaningful when the source branch has all the revisions of the target branch. In our example, the second branch can pull from the first branch, or the first branch can push to the second branch, and the result will be the same:



Why use more than one branch?

In small, solo projects, branches can be useful in many ways, such as the following:

- Separating the development of unrelated features that can be implemented independently
- Switching between tasks
- Experimenting with different approaches to solve some problem

In large projects with many collaborators, using multiple branches is inevitable in order to maintain multiple versions of the project in parallel.

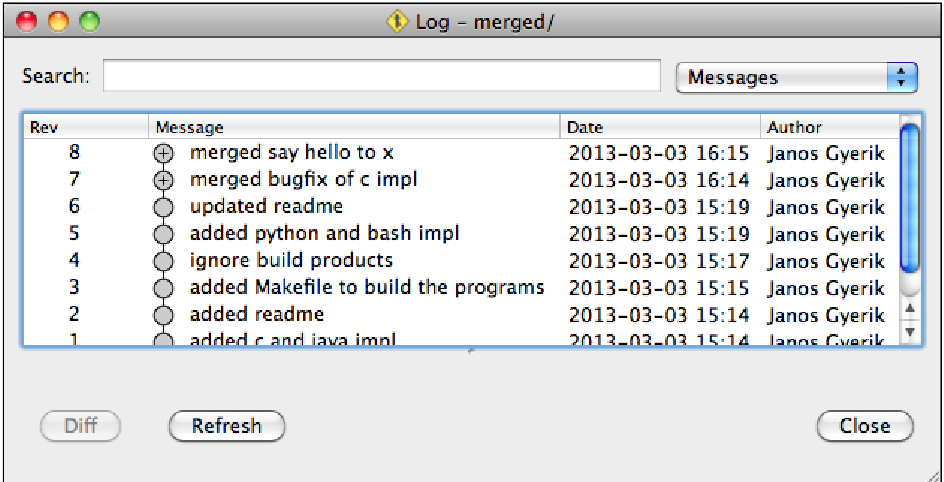
Separating the development of new features

By using a dedicated branch to implement some new features or a set of related changes within the same topic, the changes can be cleanly isolated from other work in the project. This effectively eliminates side effects and instabilities caused by other unrelated changes in the project while working on the new feature.

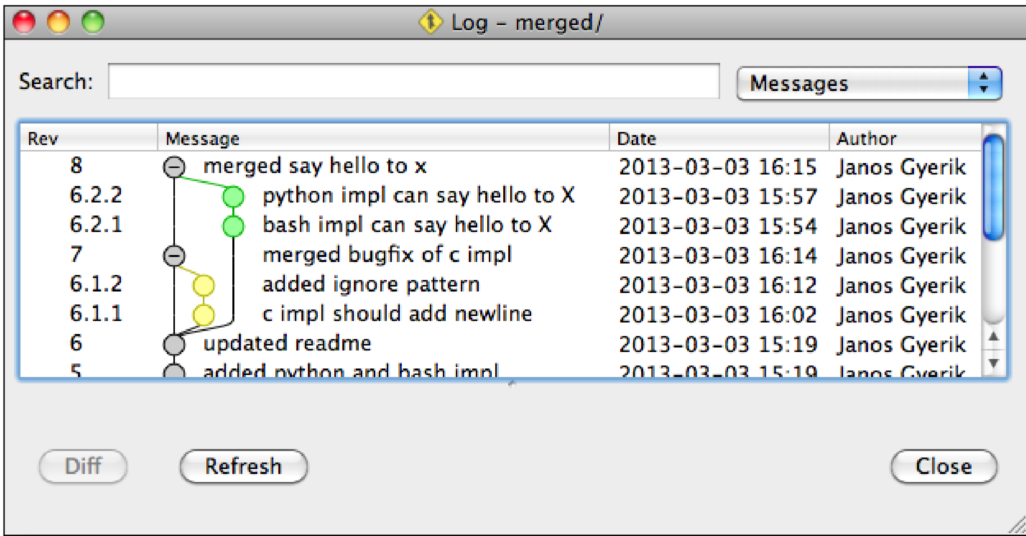
Such dedicated branches are often called **feature branches** or **topic branches**. These terms are equivalent; for simplicity, we will use the term feature branches throughout this book.

Isolating the development of features in this way also makes the project history cleaner—when the feature branch is merged into the main branch, the revisions in the branch are grouped together under a single revision, hiding the low-level details by default.

For example, merged branches are shown collapsed in the history:



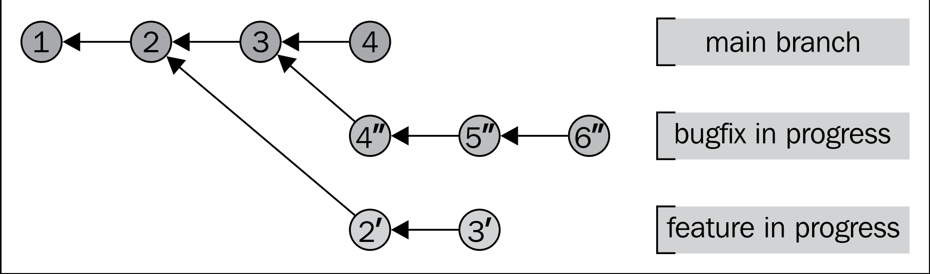
The following screenshot shows merged branches expanded to see the detailed steps:



In this way, you can get a higher level view of the history with only the larger steps, and can drill down into more details as needed. This also allows rolling back a feature by reverting a single revision that performed the merge.

Switching between tasks

When you work on different features or topics in separate dedicated branches, you can easily switch between tasks when needed. For example, if you are in the middle of developing a new feature but suddenly something more important comes up, then you can suspend your work in progress and switch to the urgent task. Once the urgent task is completed, you can return to your previous task and continue where you left off from:

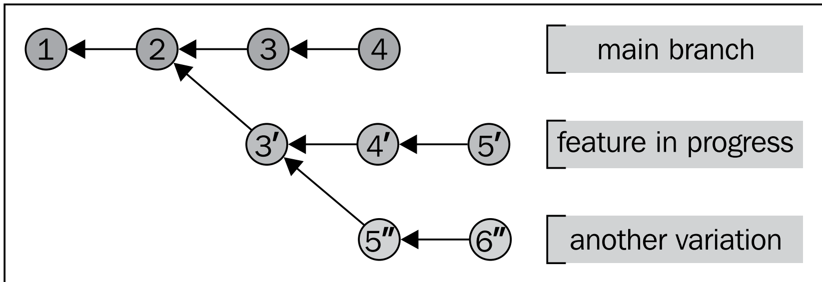


You can switch back-and-forth between multiple tasks in this way, develop them gradually through multiple commits, and when a feature is completed, you can merge it into the main branch of the project.

Experimenting with different approaches

In the same way that you can work on multiple features using multiple dedicated branches, you can also try different implementations of the same feature and split them into multiple branches.

Another typical situation is when in the middle of working on a feature, you realize that your last few commits may be going in the wrong direction and a different approach might be better. When this happens, you don't have to throw away what you've done so far; it is much better to create a new branch based on the last revision you were still happy with, and try a different approach in that new branch:



If at a later stage you realize that your first approach was not so bad after all, you can still return to it easily at any time.

Maintaining multiple versions

In projects with ongoing development and stable milestone releases, using multiple branches is inevitable in order to maintain multiple versions in parallel. In addition to the main branch with ongoing development, typically at least one more branch is needed for the occasional maintenance work and bug fixes on the last stable release.

For example, if a critical bug is discovered that needs to be fixed and released urgently, then using the main development branch is usually not an option, as it typically contains many changes that have not been thoroughly tested yet. One good solution in such situations is to keep a separate branch for the last stable release, ready to receive bug fixes and to be rereleased at a minimal risk of side effects.

In sufficiently large projects, multiple branches are essential for maintaining the stable lifecycle of the project and to manage its releases.

Understanding core terms and concepts

There are a handful of commonly used terms and concepts when working with branches.

trunk, master, and mainline

trunk, master, and mainline are common names for the main branch in a project, where most of the new development happens. These terms are equivalent; for simplicity, we will use the term **trunk** throughout this book.

Technically speaking, the trunk is no different from any other branch. What makes it special is the agreement among collaborators in the project to use this branch as the "official" version, and the basis for creating new branches for new development.

The trunk is typically not necessarily well-tested nor stable, though there are exceptions, and in general this depends on the working style adopted by the collaborators of the project.

The tip of a branch

The latest revision in a branch is often referred to as the **tip**, or the **HEAD**. These terms are equivalent; for simplicity, we will use the term tip throughout this book.

Source and target branches

In all branch operations, the direction of the operation is always significant, especially when the operation modifies one of the branches.

We will use the term **source branch** to refer to the branch where the operation starts from, or for a branch that is used in a read-only manner to copy data from. We will use the term **target branch** to refer to the branch that is the target of the operation, typically modified as a result of the operation.

For example:

- When creating branchB based on branchA, we will call branchA the source and branchB the target.
- When merging branchX into branchY, we will call branchX the source and branchY the target.
- When pushing branchP onto branchQ, we will call branchP the source and branchQ the target.

Parent branch

When branchB is created based on another branch, branchA, then branchA is called the **parent branch** of branchB. In other words, the source branch that branchB was created from is its parent branch. Bazaar records this relationship in branchB, and it can be useful when you want to reference the parent branch in certain branch operations.

Diverged branches and the base revision

Two branches that have a common history only up to a certain revision, but a different history after that point are **diverged branches**.

The last common revision in two related but diverged branches is called the **base revision**. Identifying the base revision is important when merging branches, in order to find the unique revisions in each branch to be merged.

The whole point of having multiple branches is, of course, to diverge—make different kinds of changes, go in different directions. However, keep in mind that if the further two branches diverge from each other, it may be increasingly difficult to merge them back together later. This will be explained in more detail, later in this chapter.

Storing branch data

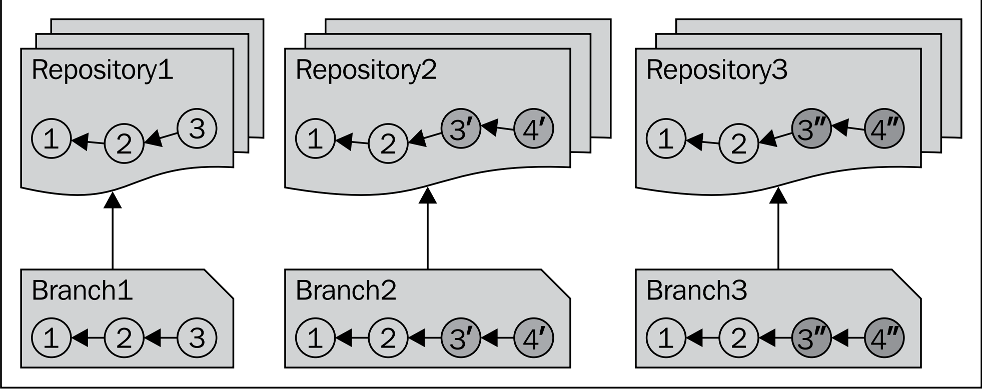
Although branches typically evoke the image of a directed graph of revisions, in terms of physical storage, the data of a branch in Bazaar is surprisingly simplistic—it consists of the internal revision ID of the last revision (branch tip), along with a few additional lightweight properties.

Since the metadata stored in each revision includes references to parent revisions, Bazaar can reconstruct the graph of the entire revision history by starting from the branch tip and following the parent relationships all the way until the initial revision.

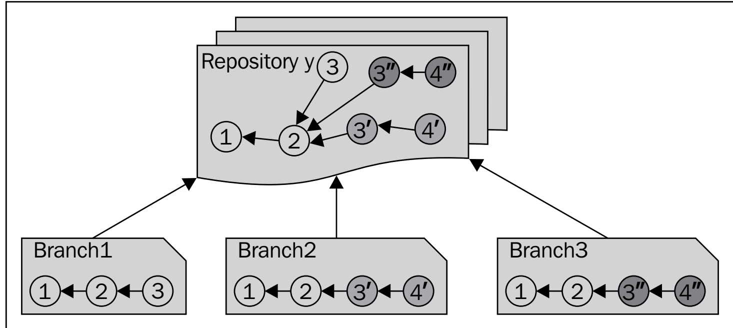
Using a shared repository

The default configuration of a Bazaar project is the standalone tree. A standalone tree includes a repository, a branch and a working tree.

When working with multiple branches, creating multiple standalone trees is inefficient, because the revision data is duplicated in multiple repositories:



A more efficient solution is the shared repository configuration, where branches don't have their dedicated repository but share a single common repository. The shared repository stores all the revisions of all the branches, eliminating unnecessary duplication, and speeding up branch operations:



A shared repository with its branches is organized as follows in the local filesystem:

```
/path/to/shared/repo/
|-- .bzr
|   |-- repository      # repository data
|-- some_branch
|   |-- ...             # project files in the working tree
|   |-- .bzr            # hidden .bzr directory
|       |-- checkout    # working tree data
|       |-- branch      # branch data
|-- another_branch
|   |-- ...             # project files in the working tree
|   |-- .bzr            # hidden .bzr directory
|       |-- checkout    # working tree data
|       |-- branch      # branch data
|-- yet_another_branch
|   |-- ...
```

The shared repository is inside the `.bzr` directory at the top level.

The subdirectories correspond to branches, optionally with a working tree. In this setup, the branches don't have their own repository; they all use the common shared repository.

In practice, most branches differ only in a few revisions. Therefore, using a shared repository usually makes a huge difference both in terms of storage and speed. The larger the project, the greater the benefits of a shared repository.

In general, it is almost always better to use a shared repository right from the start instead of a standalone tree, even if you don't expect to work with branches or collaborate with others in the near future.

Using the command line

You can create a shared repository using the `bzr init-repository` command with the target location as a parameter. For example:

```
$ bzr init-repository /tmp/shared-repo
Shared repository with trees (format: 2a)
Location:
  shared repository: /tmp/shared-repo
```

This creates a new directory with only a hidden `.bzr` directory inside. This directory stores only a repository without branch data or working tree data. Working tree and branch commands will not work at the root of a shared repository. For example:

```
$ bzr status
bzr: ERROR: No WorkingTree exists for "/tmp/shared-repo/.bzr/checkout/".
$ bzr log
bzr: ERROR: Not a branch: "/tmp/shared-repo/.bzr/branch/": location is a
repository.
```

The shared repository directory is a container for branches. You can create as many branches inside as needed. The branches will store their revisions inside the shared repository instead of their own `.bzr` directory. This way, all the branches can reuse the common revisions.

Using Bazaar Explorer

In Bazaar Explorer, you can create a shared repository using the **Initialize** view, the same view you used for creating branches. You can open this view in several ways:

- From the menu, select **Bazaar** | **Start** | **Initialize**
- From the **Welcome** view, in the **Start a new project** tab, select **Initialize**
- Using shortcuts such as *Ctrl* + *N* (Windows, Linux), *Cmd* + *N* (Mac OS X)

In the **Location** textbox, you can either type the path to the target directory, or click on the **Browse** button and navigate to it. In the **Workspace Model** box, select either **Shared repository** or **Feature branches**.

With the **Shared repository** option, Bazaar will simply create an empty shared repository, just like the **bzr init-repository** command. With the **Feature branches** option, Bazaar will create a shared repository and a new branch inside it named `trunk`. If the target directory contains files, use **Move existing files**, if any, to the working tree location to have Bazaar move those files into the new branch.

Basic branching and merging

To give you an idea of how branching and merging works, we will walk through a simple example that uses multiple branches. Our example is a simple project containing "hello world" programs that simply print **Hello World!** on the console, implemented in different languages.

Given a stable branch called `trunk`, we will create a branch to work on a new feature, and before the feature is finished, we will create another branch to fix an urgent bug. Finally, we will merge the `bugfix` branch back to the trunk.

In the example steps, we will focus on using the branch operations with basic parameters, the content of the files, and the changes made are not important. In the later sections, we will explain all branch operations in more detail.

To use branches efficiently, let us first create a shared repository using the command line or Bazaar Explorer, as explained in the previous section. The examples will assume the path `/sandbox/hello` (on Linux or Mac OS X) or `C:\sandbox\hello` (on Windows), but you may choose any other path of your choice.

Getting the example project

The example project is available publicly on Launchpad (`https://code.launchpad.net/~bzrbook/bzrbook-examples/hello-start`). This URL points to a Bazaar branch. Using the **branch operation**, we will create a local branch that is a perfect replica of the remote branch, with all its revisions and complete history copied locally.

The new local branch will be completely independent of the original remote branch. We will use it as the basis of many other branch operation examples throughout the chapter.

Using the command line

You can create a branch based on another branch by using the `bzr branch` command and by specifying the URL of the source branch. Optionally, you can specify the target directory, where you want to create the new branch. Let's try that to download the example branch into `/sandbox/hello/trunk`:

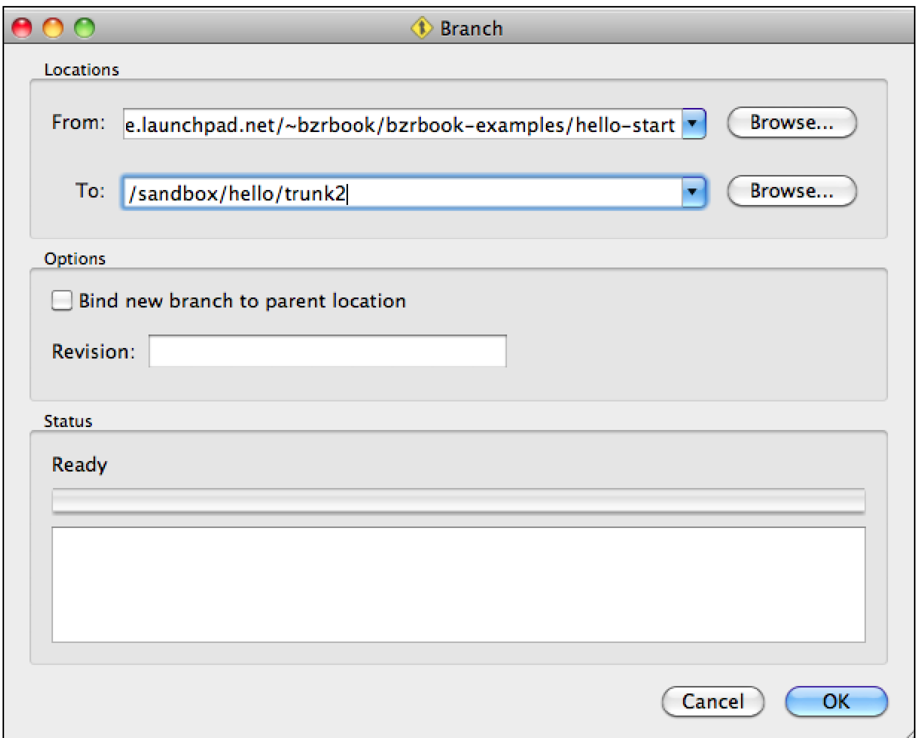
```
$ mkdir /sandbox/hello
$ cd /sandbox/hello
$ bzr branch https://code.launchpad.net/~bzrbook/bzrbook-examples/hello-start trunk
Branched 6 revisions.
```

Using Bazaar Explorer

You can create a branch based on another branch using the **Branch** view. You can open this view in several ways:

- Click on the large **Start** button in the toolbar and select **Branch...**
- From the menu, select **Bazaar** | **Start** | **Initialize**

In the **From:** textbox, enter the URL of the source branch. In the **To:** textbox, you can either type the path of the directory where you want to create the branch, or click on the **Browse** button and navigate to it. In the **Revision** textbox, you can specify a revision to download the history only up to that revision. In this example, we want to download all the revisions, so simply leave it empty:



After you click on **OK**, the **Status** box will show the `bzr` command that was executed along with its output. For example:

```
Run command: bzr branch https://code.launchpad.net/~bzrbook/bzrbook-
examples/hello-start /sandbox/hello/trunk2 --use-existing-dir
Branched 6 revisions.
```

Creating a feature branch

At this point, the "Hello World!" implementations are very primitive. Executing any of them simply prints the message **Hello World!** on the screen, and that's it. For example:

```
$ ./hello.py
Hello World!
```

Let's say we want to improve the programs to accept an input parameter, and use it to make them more personal. For example, if you run `./hello.py Jack`, it should print **Hello Jack!** instead of **Hello World!**

In order to separate the new development from the main branch, let's create a new branch for it. Actually, you already know how to create a feature branch; the method is the same as the one we used to download the sample branch. The only difference is that the source branch will be a local branch, which is the trunk, instead of the original remote branch.

Using the command line

Let's use the same method we used earlier to get the remote branch, but this time branching from the local trunk, and let's call the new branch `say-hello-to-x`:

```
$ cd /sandbox/hello
$ bzr branch trunk say-hello-to-x
Branched 6 revisions.
```

Using Bazaar Explorer

To create a new branch from the trunk, open the **Branch** view in the same way as you did earlier, using the menu or the large **Start** button in the toolbar. When coming from the **Status** view, the `From:` textbox is prefilled with the path of the current branch. In the `To:` textbox, enter the path or browse to the location to place the new branch. For example, `/sandbox/hello/say-hello-to-x2`.

Working on a branch

The goal of our branch is to implement a new feature—change the programs to use a parameter. Let's start with `hello.sh`; rewrite it as follows:

```
#!/bin/sh
#
if test "$1"; then
    echo "Hello $1!"
else
    echo 'Hello World!'
fi
```

This change seems noteworthy enough to include a memo in the `README` file. For example, let's append the following to the end of the file:

```
Variation: "say hello to X"
---------------------------
- Bash
```

So far so good; let's commit the change using Bazaar Explorer or the command line:

```
$ bzr commit -m 'bash impl can say hello to X'
Committing to: /sandbox/hello/say-hello-to-x/
modified README.md
modified hello.sh
Committed revision 7.
```

Starting another branch

We are not yet done with the `say-hello-to-x` branch; we still have to change the implementations of the other languages too. However, let's suppose that somebody using our project has discovered a bug in the C implementation—the program does not print a line ending character, making the output strange. Let's suppose we have to finish this quickly, or else our good reputation is at stake here.

Instead of fixing the problem in the current branch or trunk, let's use a new dedicated branch. For one thing, the change clearly does not belong to the `say-hello-to-x` branch. And to avoid our intermediate commits in the fixing process from affecting the trunk, using a new branch is really the cleanest solution.

Let's create the new bugfix branch based on the trunk, using Bazaar Explorer or the command line. For example:

```
$ cd /sandbox/hello
$ bzr branch trunk fix-c
Branched 6 revisions.
$ cd fix-c
```

The fix is easy enough. We can rewrite `hello.c` as follows:

```
#include "stdio.h"

int main() {
    printf("Hello World!\n");
}
```

So far so good, let's commit the change using Bazaar Explorer or the command line:

```
$ bzr commit -m 'c impl should add newline'
Committing to: /sandbox/hello/fix-c/
modified hello.c
Committed revision 7.
```

On a second thought, our current implementation is not great. The first line is in a somewhat old-fashioned writing style; it would be better to change it as follows:

```
#include <stdio.h>
```

Let's commit this too, using Bazaar Explorer or the command line:

```
$ bzr commit -m 'use more modern include-style'
Committing to: /sandbox/hello/fix-c/
modified hello.c
Committed revision 8.
```

Merging the bugfix branch

Now that the work on the bugfix branch is completed, let's merge it in the trunk.

Using the command line

You can merge from another branch using the `bzr merge` command and specifying the URL of the source branch to merge from. Let's change to the directory of the trunk and merge from the bugfix branch:

```
$ cd /sandbox/hello/trunk
$ bzr merge ../fix-c/
 M  hello.c
All changes applied successfully.
```

At this point, the changes in the bugfix branch have been applied to the working tree, leaving it in a changed state without recording a new revision, as we can confirm using the `status` command:

```
$ bzr status
modified:
  hello.c
pending merge tips: (use -v to see all merge revisions)
  Janos Gyerik 2013-03-03 use more modern include-style
```

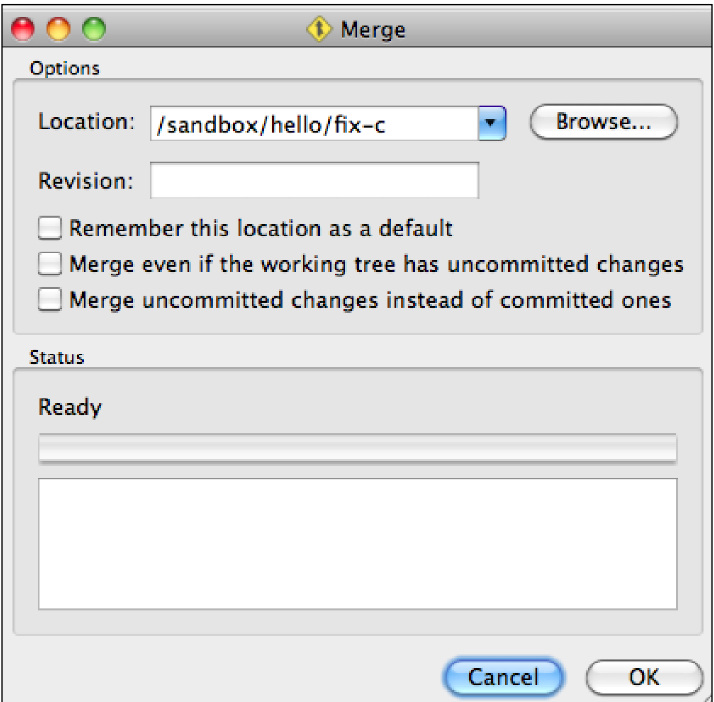
Notice that in addition to showing that `hello.c` has been modified, the `status` command also indicates that we are in the middle of merging. By default, Bazaar shows the log message of the last revision in the branch that is being merged.
Use the `-v` flag to see all merged revisions.

After running the `merge` command, you should always verify the changes that resulted by the merge and that everything in the project is still working well. If everything looks in order, you can commit the merge. If there are problems, you can use the `revert` command to undo the merge. We will revert now to try merging using Bazaar Explorer.

Using Bazaar Explorer

To merge the bugfix branch into the trunk, you need to open the trunk in Bazaar Explorer. You can do this by using any of the **Open...**, or **Open location...**, or **Open Recent** options in the **File** menu, or by using the *Ctrl* + *O* keyboard shortcut in Windows and Linux systems or *Cmd* + *O* in Mac OS X.

Having the trunk opened in the **Status** view, click on the large **Merge** button in the toolbar to open the **Merge** view.



By default, the **Location:** input box is prefilled with the location of the parent branch; in our example, it is the remote branch we started from. Change that to the path of the `fix-c` branch by directly entering the path or by using the **Browse** button.

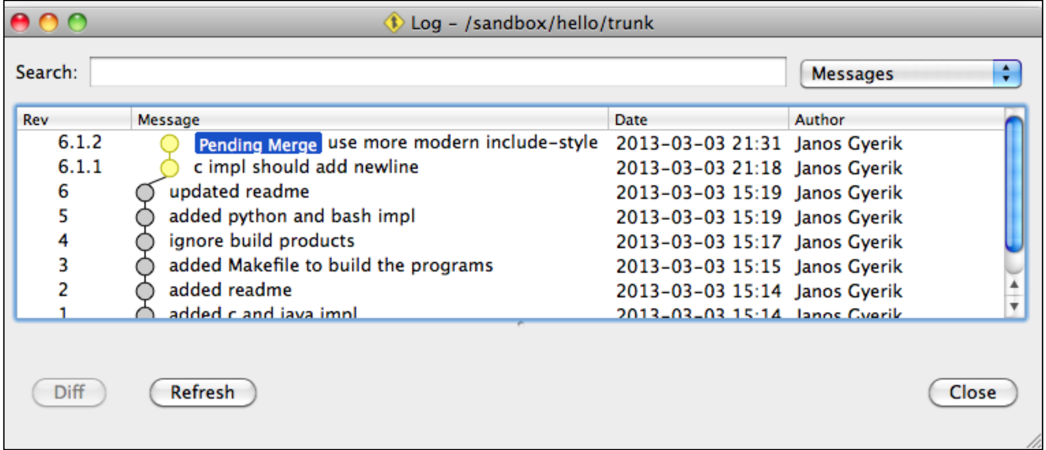
As the **Revision:** input box suggests, we can specify a revision to merge all the changes up to a certain revision only. But in our current example, we want to merge everything.

After you click on **Ok**, the **Status** box will show the `bzr` command that was executed along with its output. For example:

```
Run command: bzr merge /sandbox/hello/fix-c/
 M  hello.c
All changes applied successfully.
```

Click on **Close** to dismiss the **Merge** view and return to the **Status** view of the trunk. The left panel now shows the **Pending Merge** tip—the log message of the last revision in the branch that is being merged.

Until you commit, the merge is in a pending state. Open the **Log** view to see all the revisions that will be merged:



To finish the merge, commit the changes, ideally with a message that summarizes the meaning of the changes in the merged branch. For example, `bugfix of the C implementation`.

Viewing merged revisions in the log

When viewing the revision history in the logs, the revisions of the merged branches are hidden by default. In this way, you get an overview of the larger steps in the project, with the option to drill down into more details and see all the merged revisions.

Using the command line

Let's look at the recent four revisions in the trunk:

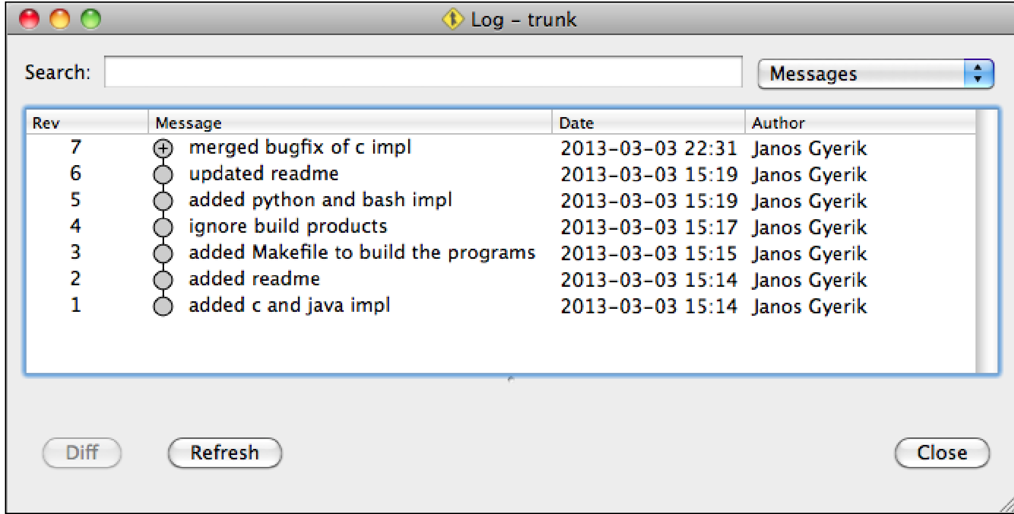
```
$ bzr log --line -l4 /sandbox/hello/trunk
7: Janos Gyerik 2013-03-03 [merge] bugfix of c impl
6: Janos Gyerik 2013-03-03 updated readme
5: Janos Gyerik 2013-03-03 added python and bash impl
4: Janos Gyerik 2013-03-03 ignore build products
```

Notice `[merge]` in front of the log message of the last commit, where we merged the bugfix branch. This was added by Bazaar to indicate that the revision merged other branches. Use the `-n0` or `--include-merged` flag to see the merged revisions too:

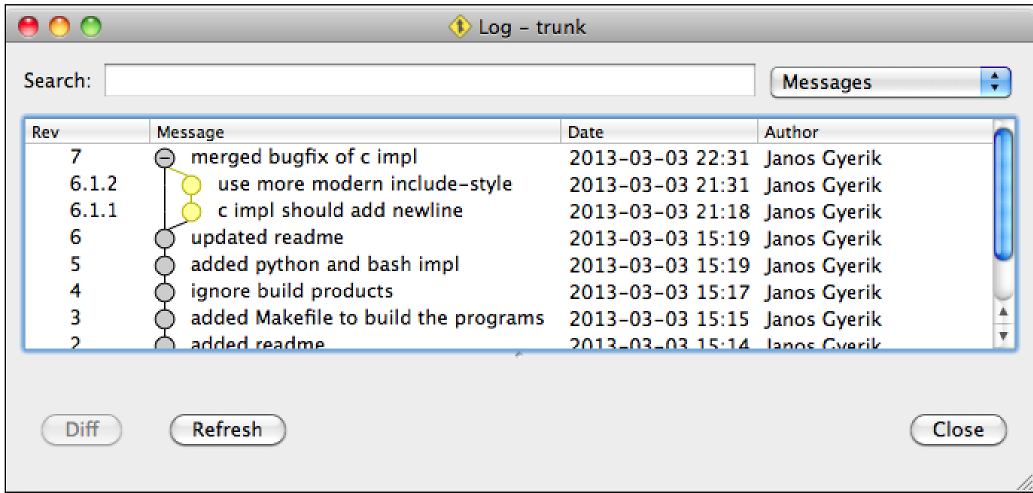
```
$ bzr log --line -l4 /sandbox/hello/trunk -n0
7: Janos Gyerik 2013-03-03 [merge] bugfix of c impl
  6.1.2: Janos Gyerik 2013-03-03 use more modern include-style
  6.1.1: Janos Gyerik 2013-03-03 c impl should add newline
6: Janos Gyerik 2013-03-03 updated readme
```

Using Bazaar Explorer

In the **Log** view, the merged revisions are collapsed by default, showing only the revision that committed the merge:



To expand the revision and view the merged revisions, click on the **+** symbol:



Using the branch command

The purpose of the `branch` command is to create a new branch based on another branch. Since Bazaar stores branches in separate directories, new branches are created in a new directory.

When used without additional parameters, the command creates a perfect replica of the source branch, copying its complete revision history. As each branch exists in its own directory in the filesystem, this is a good way of separating working environments right before starting to work on a feature branch as in the basic example earlier.

When creating the new directory for the new branch, by default Bazaar also creates a working tree. That is, the directory of the branch is populated with the project's files as of the latest or the specified revision. Note, however, that ignored files and pending changes in the source branch will not be copied, as they are not under version control.

Creating branches based on an older revision

It is often useful to create a branch that contains only the revisions up until the specified revision, instead of cloning all the revisions in the source branch. For example, when you realize that your recent changes have destabilized the project and you would like to try a different approach based on the last stable point.

Using the command line

To create a branch based on an older revision of another branch, specify the revision number with the `-r` or `--revision` flags. For example:

```
$ cd /sandbox/hello
$ bzr branch -r3 trunk try-different
Branched 3 revisions.
```

The second parameter is the name of the directory in which we create the new branch.

When the source branch is in a different parent directory or if it is a remote branch as in our first example earlier, then the second parameter becomes optional. In this case, Bazaar will use the last path segment of the URL as the name for the new branch as a sensible default. For example, the command `bzr branch http://example.com/branches/test1` without a second parameter will create the new branch in a subdirectory called `test1`, unless of course such a subdirectory already exists locally.

Using Bazaar Explorer

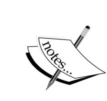
To create a branch based on an older revision of another branch using Bazaar Explorer, open the **Branch** view using the large **Start** button in the toolbar, then inside the **Options** box enter the revision number in the **Revision:** textbox.

Viewing basic branch information

The `bzr info` command prints the basic information about a specified location or the current working directory, such as the type of configuration, the parent branch, and others. For example:

```
$ bzr info /sandbox/hello
Shared repository with trees (format: 2a)
Location:
  shared repository: /sandbox/hello
```

That is, the location is a shared repository, which is configured with the `trees` option. This means that the new branches will be created with a working tree.



In the later chapters, we will see examples where it is useful to create branches without a working tree, and will learn how to reconfigure branches and shared repositories to use this option.

In case of the `fix-c` branch, we get the following:

```
$ bzr info /sandbox/hello/fix-c/
Repository tree (format: 2a)
Location:
  shared repository: /sandbox/hello
  repository branch: /sandbox/hello/fix-c
Related branches:
  parent branch: /sandbox/hello/trunk
```

That is, the location is a branch, which is in the `Repository tree` configuration. This means that it is a part of a shared repository, and it has a working tree. The output also tells the location of the shared repository and the parent branch.

You can view even more details about a branch using the `-v` or `--verbose` flags.

Comparing branches

When working with multiple branches, it is important to be able to compare them. You can see the differences between branches with different levels of detail. You can get a quick summary of the missing revisions between two branches, or view the differences between any two revisions in the same way as you compare revisions within the same branch.

The command line is more powerful, and it gives you full control over the input parameters and the level of detail in the output. On the other hand, Bazaar Explorer is somewhat limited in terms of input options, but it has a much easier to use interface.

Using the command line

When using the commands that compare branches, the basis of the comparison is the current branch by default, and the other branch to be compared must be specified as a parameter. The ordering of branches in the comparison is relevant, as comparisons depend on the perspective.

Viewing missing revisions between branches

"Missing revisions" are revisions that exist in one branch but not in the other. You can view the list of missing revisions by using the `bzr missing` command. For example:

```
$ cd /sandbox/hello/fix-c/
$ bzr missing ../say-hello-to-x/ --line
You have 2 extra revisions:
8: Janos Gyerik 2013-03-03 use more modern include-style
7: Janos Gyerik 2013-03-03 c impl should add newline
You are missing 2 revisions:
8: Janos Gyerik 2013-03-03 python impl can say hello to X
7: Janos Gyerik 2013-03-03 bash impl can say hello to X
```

The command shows the missing revisions relative to the current branch. In this example, we are in the `fix-c` branch directory and compare it to the `say-hello-to-x` branch. Thus we have two extra revisions and two missing revisions.

The missing revisions are shown using the same format as the `log` command. For brevity, we used the `--line` flag in this example, but all the other log format options work too.

To see the extra or missing revisions in only one of the branches, use the `--mine-only` or `--theirs-only` flags. For example:

```
$ bzr missing ../say-hello-to-x/ --line --mine-only
You have 2 extra revisions:
8: Janos Gyerik 2013-03-03 use more modern include-style
7: Janos Gyerik 2013-03-03 c impl should add newline
$ bzr missing ../say-hello-to-x/ --line –theirs-only
You are missing 2 revisions:
8: Janos Gyerik 2013-03-03 python impl can say hello to X
7: Janos Gyerik 2013-03-03 bash impl can say hello to X
```

Instead of `--mine-only`, you can use the shorter aliases, namely `--mine` or `--this`. Instead of `--theirs-only`, you can use the shorter aliases, namely `--theirs` or `--other`.

Viewing the differences between branches

You can view the differences between two branches by using the `bzr diff` command in the same way as you compared the revisions within the same branch. However, you must specify the other branch by using either the `--new` or `--old` flags, depending upon which branch should be the basis for comparison.

Without the additional parameters, the command shows the differences in the entire project. To view the differences only to a set of files, specify the files or subdirectories as parameters. For example, to see the changes going from the `fix-c` branch to the `say-hello-to-x` branch affecting only the `README` file, as follows:

```
$ bzr diff --new ../say-hello-to-x/ README.md
=== modified file 'README.md'
--- README.md   2013-03-03 14:19:48 +0000
+++ README.md   2013-03-07 20:53:05 +0000
@@ -9,3 +9,9 @@
 - C
 - Java
 - Python
+
+
+Variation: "say hello to X"
+--------------------------
+- Bash
+- Python
```

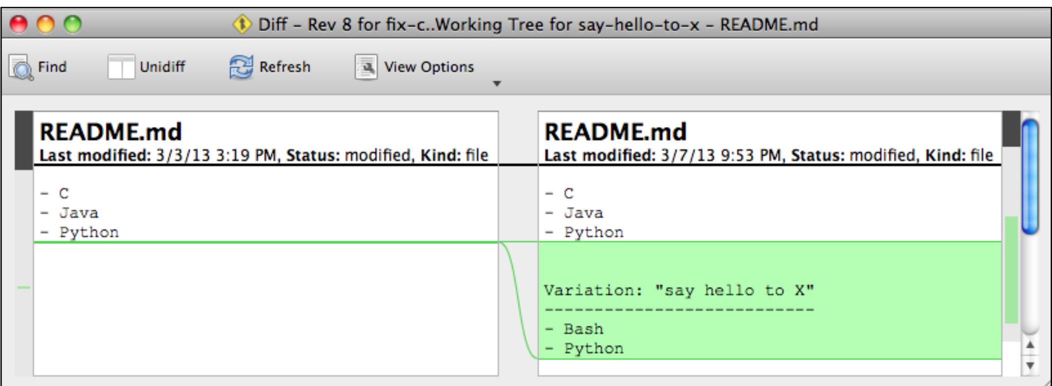
That is, the `say-hello-to-x` branch added some lines in the file. Sometimes, it can be useful to view the differences in the other direction; that is, to see the changes going from the `say-hello-to-x` branch to the `fix-c` branch:

```
$ bzr diff --old ../say-hello-to-x/ README.md
=== modified file 'README.md'
--- README.md   2013-03-03 14:57:13 +0000
+++ README.md   2013-03-07 20:55:52 +0000
@@ -9,9 +9,3 @@
```

```
  - C
  - Java
  - Python
-
-
-Variation: "say hello to X"
----------------------------
-- Bash
-- Python
```

We simply replaced `--new` with `--old` to reverse the direction of the differences.

Remember that you can always replace the `diff` command with `cdiff` to highlight the differences using colors, or `qdiff` to view the differences in Bazaar Explorer instead. For example:



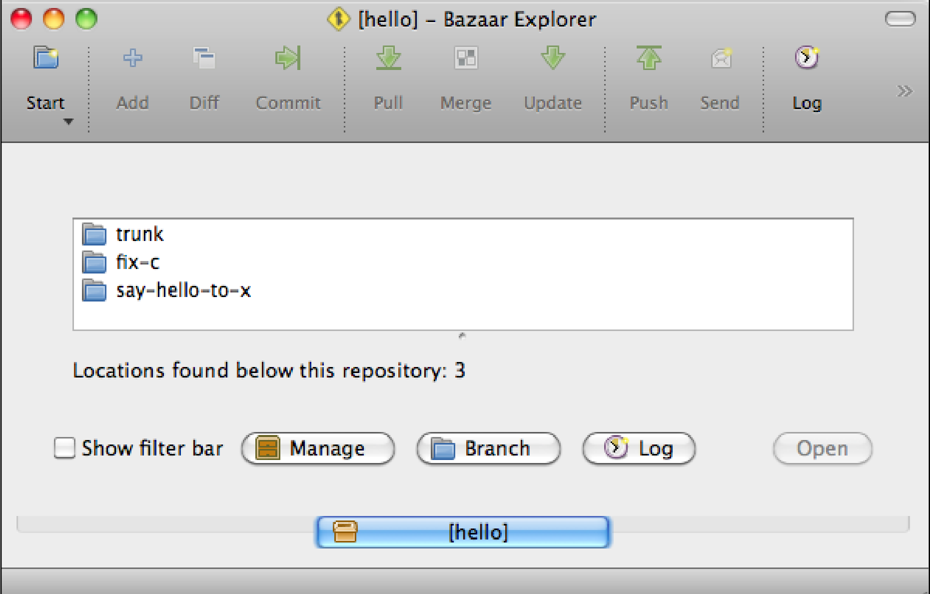
The `cdiff` command is provided by the `bzrtools` plugin, and it requires a terminal application that supports colors. The `qdiff` command is provided by the `qbzr` plugin.

Note that the `revision` parameter is always applied to the current branch. One way to compare against a specific revision in another branch is to do it in two steps—first, create a temporary branch based on the other branch at the desired revision, and then compare the current branch against the temporary branch.

Using Bazaar Explorer

A very useful interface in Bazaar Explorer for viewing and managing multiple branches is the **Repository** view. To open this view, use the **Open...** option in the **File** menu or the *Ctrl* + *O* keyboard shortcut in Windows and Linux or *Cmd* + *O* in Mac OS X, then navigate to the shared repository location, and click on **Choose**.

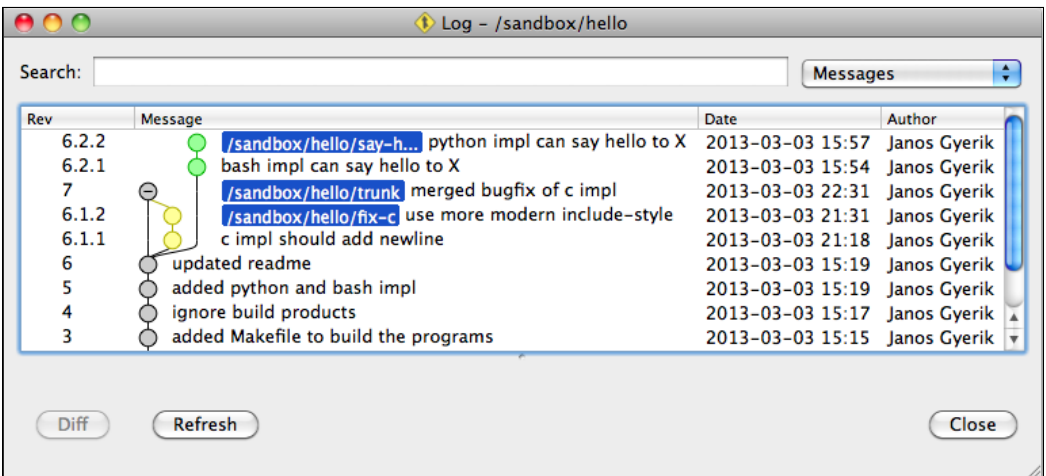
When the view opens, it simply shows a list of the branches in the shared repository. For example:



Let's explore the different functions that are accessible from this main view.

Viewing the tree of branches

A very nice function in Bazaar Explorer is to show multiple branches at the same time in the **Log** view. You can select multiple branches in the list by pressing the *Ctrl* key and clicking in Windows and Linux, or pressing the *Cmd* key and clicking in Mac OS X. To view all the branches, simply select nothing (by clicking anywhere within the empty area) and click on the **Log** button:



In this example, all the branches are shown in the tree, with the branch tips highlighted with the location of the branch. This is a great way for viewing the revisions in the different branches at a glance.

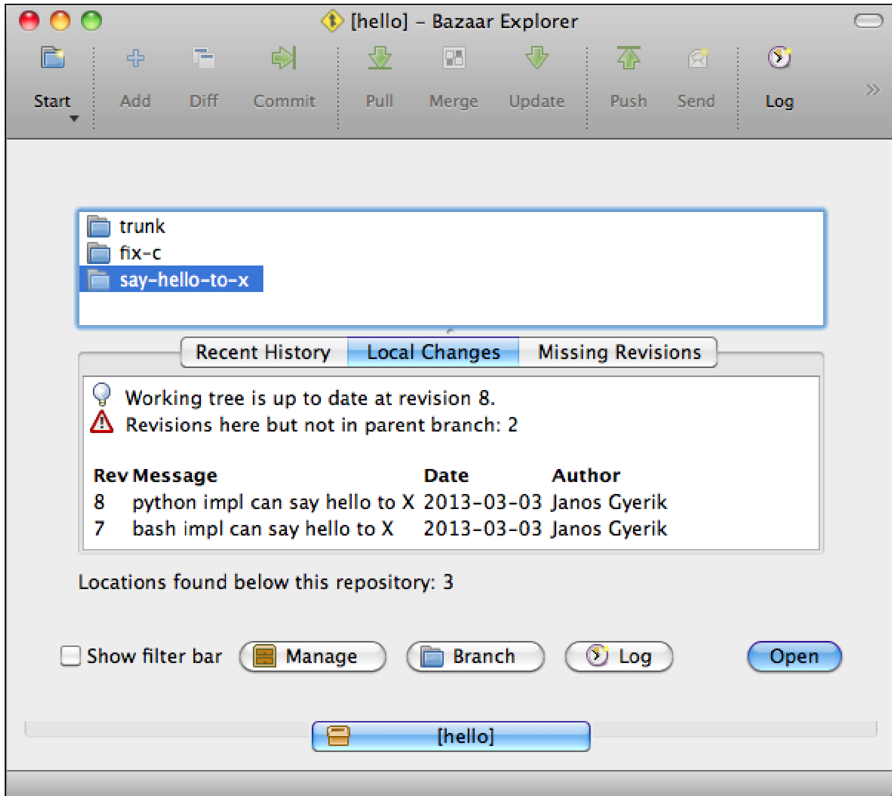
The other functions of the **Log** view work as usual, such as filtering and viewing differences in each revision. However, this also means that you can only see the differences in a single revision, and at the time of this writing, there is no user interface for comparing two arbitrary revisions.

Viewing missing revisions between branches

When you click on a branch in the **Repository** view, the **details** panel under the list of branches shows more details about the branch using three tabs:

- **Recent History**: This shows the last five commits
- **Local Changes**: This shows the commits that are in the branch but not in the parent branch
- **Missing Revisions**: This shows the revisions that are in the parent branch but not in the selected branch

For example, the `say-hello-to-x` branch has two revisions that are not yet in the parent:



Viewing the differences between branches

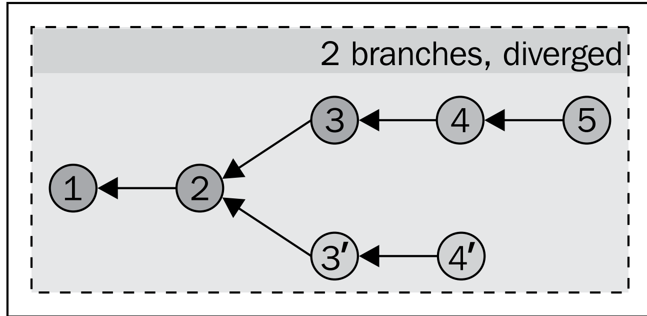
At the time of this writing, the **Repository** view doesn't have an option to parameterize the **Diff** view and select revisions. As a workaround, you can use the command line to select the revisions in the same way as you do with the `bzr diff` command, but instead of `bzr diff`, run `bzr qdiff`.

Merging branches

Merging branches is a complex operation in any version control system. The `merge` operation in Bazaar combines the changes of another branch into the current branch, and copies all the revision metadata including the historical ordering information as well.

Performing a three-way merge

The default merge algorithm used by Bazaar is the so-called three-way merge. The first step in this algorithm is identifying the base revision of the two branches; that is, the point after which the branches have diverged. For example, given the following two branches, the base revision is `2`, which is also the common ancestor of the two branches:



The general logic of the three-way merge algorithm is to take the base revision as the starting state, then automatically apply the non-conflicting changes that took place in the two branches. For example, if a file was edited only by the first branch or the second branch but not by both, the changed file will be included in the result regardless of which branch it was edited in.

In case of text files that were edited by both the branches, the same logic can be applied by evaluating the changed blocks within the file. For example, if we take the original line `Lorem ipsum` at revision 2 and the same line at the latest revisions in the two branches, then the three-way merge algorithm works out the correct action as follows:

Line in base revision	**Line in the first branch**	**Line in the second branch**	**Result**	**Comment**
`Lorem ipsum`	`Lorem ipsum`	`Lorem ipsum`	`Lorem ipsum`	No change
`Lorem ipsum`	`Dolor sit amet`	`Lorem ipsum`	`Dolor sit amet`	Use line from first
`Lorem ipsum`	`Lorem ipsum`	`Ante ipsum`	`Ante ipsum`	Use line from second
`Lorem ipsum`	`Est ipsum`	`Est ipsum`	`Est ipsum`	Same change in both
`Lorem ipsum`	`Dolor sit amet`	`Ante ipsum`	`?`	Conflict

That is, if none of the branches changed the line, then the line is kept unchanged. If only one branch changed the line, the algorithm simply takes the changed line, regardless of the branch that changed it. If both the branches changed the line the same way, use the changed line.

However, if both the branches changed the same line in a different way, then the algorithm cannot decide what is the right thing to do. So, it marks the file as conflicted, which must be resolved manually by editing the file and correcting it.

The same is true for other kinds of changes; for example, the renaming of a file. If only one of the branches renamed a file, then the file will be renamed in the result. If both the branches renamed the file differently, then the `merge` operation will mark that as a conflict that must be resolved manually by choosing the correct name.

Completing the merge

As a result of the merge, the changes in the unique revisions of the other branch are applied to the working tree, leaving it in a changed state. You can use the usual Bazaar commands to view what has changed, such as `bzr status`, `bzr diff`, `bzr log`, and their equivalents in Bazaar Explorer.

As nothing is committed yet, you have at least two options:

- Commit the changes to record a new revision
- Abort the merge and restore the working tree

Committing the merge

If the merge operation was successful and there were no conflicts, you should review the changes carefully. Even though there were no conflicts during the merge operation, the changes in the other branch might have important implications for the current branch, warranting adjustments. For this reason, it is important to understand the changes introduced by the other branch.

After you have reviewed the changes and tested the new state of the project, you can commit the changes as a new revision in the same way as always. In addition to the changes, Bazaar will also record all the meta information about the merged revisions in the other branch.

As a result, the current branch will have all the information of both the branches combined, and you will be able to reconstruct both the original branches as they were before the merge operation, if necessary in the future.

Aborting the merge

If the merge operation was not successful, for example if it resulted in too many conflicts, or you are simply not happy with the new state of the project, then you can abort the merge by reverting the changes. As always, you can revert the changes selectively by specifying files and directories, or revert all the changes at once.

Reverting all the changes will completely abort the merge and restore the working tree to the state of the last revision in the current branch.

If you revert the files selectively, keep in mind that during a merge operation Bazaar keeps a **pending merge marker** with the meta information of the merged revisions. If you commit in such a state, Bazaar will still consider that as a merge. You can revert the pending merge marker using the `--forget-merges` flag.

Resolving conflicts

When two branches have made contradicting changes, it is impossible to determine automatically which change should take precedence, or to combine the changes in a meaningful way. In such situations, Bazaar marks the involved files as **conflicted**, and you must resolve them manually.

It is easy to think of contradictory changes. For example:

- BranchA and BranchB changed the same line of the same file differently
- BranchA and BranchB renamed the same file differently
- BranchA changed something in a file, but BranchB deleted that file
- BranchA added a file to a directory, but the directory was deleted in BranchB
- BranchA added a new file, but BranchB also added a file with the same name but with different content

These are just a few examples of contradictory changes that can happen in two branches and cannot be resolved automatically. You must investigate why such contradicting changes took place in the branches, and decide the right resolution that is appropriate and logical for the project.

When Bazaar detects a conflict, it marks the files involved in the change as conflicted. You can view the list of conflicts by using the `bzr conflicts` command. As long as there are unresolved conflicts in the working tree, Bazaar will not let you commit a new revision.

Bazaar categorizes conflicts into conflict types, as documented in `bzr help conflict-types`. Regardless of the conflict type, resolving a conflict involves two main steps:

1. Make the necessary changes in the working tree to fix its state, typically by combining the changes of the two branches in a meaningful way, or sometimes by taking the changes of only one branch and ignoring the other.
2. Tell Bazaar that the conflict is resolved in order to clear the conflict marker.

Let's demonstrate how this works by using two common conflict types as examples—text conflicts and content conflicts.

Resolving text conflicts

When both the branches changed the same line of the same text file in a different way, it is called a **text conflict**. Let's create such a scenario using the following steps:

1. Create two new branches, namely `text1` and `text2`, based on the trunk.
2. In `text1`, edit the `hello.c` file, change `Hello World` to `Hi World`, and commit the change.
3. In `text2`, edit the `hello.c` file, change `Hello World` to `HELLO World`, and commit the change.
4. In `text2`, merge from the `text1` branch.

Using the command line as follows:

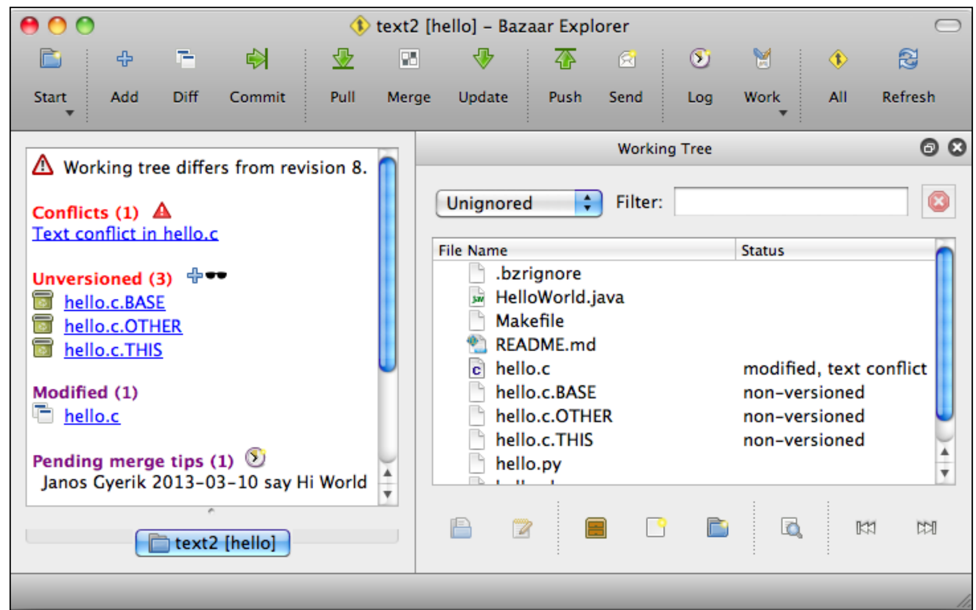
```
$ bzr branch trunk/ text1
Branched 7 revisions.
$ bzr branch trunk/ text2
Branched 7 revisions.
$ cd text1
$ vim hello.c  # change Hello World to Hi World
$ bzr commit -m 'say Hi World'
Committing to: /sandbox/hello/text1/
modified hello.c
Committed revision 8.
$ cd ../text2
$ vim hello.c  # change Hello World to HELLO World
$ bzr commit -m 'say HELLO World'
Committing to: /sandbox/hello/text2/
```

```
modified hello.c
Committed revision 8.
$ bzr merge ../text1
 M  hello.c
Text conflict in hello.c
1 conflicts encountered.
```

If we take a look at the status of the working tree, it will look similar to the following:

```
$ bzr status
modified:
  hello.c
unknown:
  hello.c.BASE
  hello.c.OTHER
  hello.c.THIS
conflicts:
  Text conflict in hello.c
pending merge tips: (use -v to see all merge revisions)
  Janos Gyerik 2013-03-10 say Hi World
```

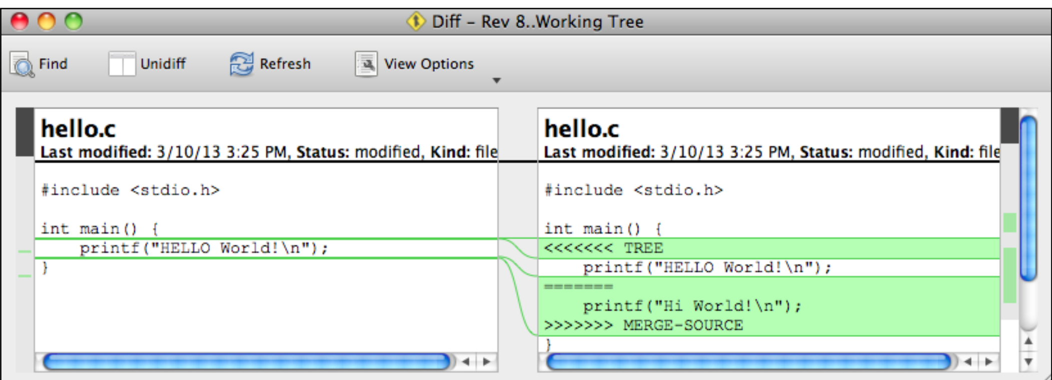
If we use Bazaar Explorer to see the output, it will look as follows:



In addition to modifications in the `hello.c` file, Bazaar adds three new files with `.BASE`, `.OTHER`, and `.THIS` suffixes, and reports a text conflict. These three new files are different versions of `hello.c` that help us sort out the conflict:

- `hello.c.BASE`: This is a copy of the file as of the base revision
- `hello.c.OTHER`: This is a copy of the file as of the revision in the other branch
- `hello.c.THIS`: This is a copy of the file as of the revision in the current branch

Let's look at the modifications in `hello.c` using Bazaar Explorer:

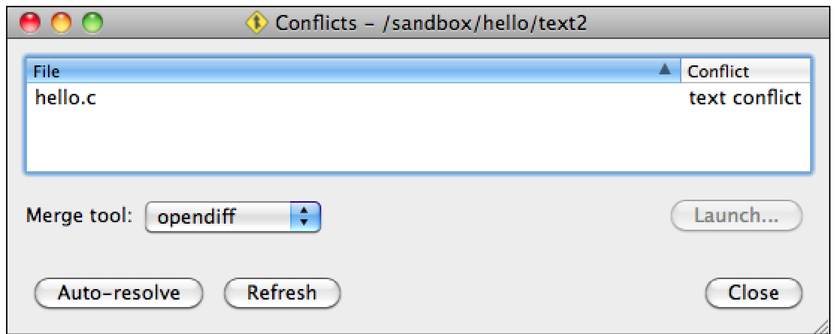


The markers "`<<<<<<<`" and "`>>>>>>>`" are called **herringbone markers**, and have been added by Bazaar to help compare the conflicting changes in two branches. The lines between `<<<<<<< TREE` and `=======` are as they were in the current branch (`text2`), and the lines between `=======` and `>>>>>>> MERGE-SOURCE` are as they were in the merge source (the other branch, `text1`). That is, the affected area in the file is changed following this general format:

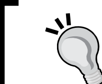
```
<<<<<<< TREE
ZERO OR MORE LINES AS THEY EXIST
IN THE REVISION IN THE CURRENT BRANCH
=======
ZERO OR MORE LINES AS THEY EXIST
IN THE REVISION IN THE OTHER BRANCH (THE MERGE SOURSE)
>>>>>>> MERGE-SOURCE
```

One solution is to edit the file, remove the herringbone markers, and make the line look like the way it should, which depends upon the situation and cannot be decided automatically.

A graphical merge tool can help in reviewing and resolving the conflicts. To launch the graphical merge tool, you first need to open the **Conflicts** view in Bazaar Explorer. You can either click on the small red triangle next to the list of conflicts, or click on the large **Work** button in the toolbar and select **Resolve Conflicts...**, or with the `bzr qresolve` command:



The **Conflicts** view shows the list of conflicts. Click on a conflict to select it, use the **Merge tool:** combo box to choose a merge tool and click on **Launch...** to start it. Using the merge tool you can review the conflicting blocks inside the selected file. Depending upon the tool of your choice, you may be able to choose the changes from one of the branches, or freely edit the file using the `.THIS`, `.OTHER`, and `.BASE` versions as references; the available options and features depend on the tool.



If you made a mistake while editing the working tree to resolve the conflicts, you can always restart from the beginning by reverting and restarting the merge.

After correcting the file manually or with the help of a merge tool, you must tell Bazaar that the conflict is resolved using the `bzr resolve` command. For example:

```
$ bzr resolve hello.c
1 conflict resolved, 0 remaining
```

With this step, Bazaar clears the conflict marker, and it also removes the temporary `.THIS`, `.OTHER`, and `.BASE` files it created earlier.

Instead of resolving the conflicting changes one by one, sometimes the solution is to take the changes of one branch and ignore the conflicting changes of the other. The `bzr resolve` command provides convenient shortcuts for such situations by using the `--take-this` and `--take-other` flags:

- `--take-this`: This resolves the conflict by using the version of the current branch, ignoring the conflicting changes by the other branch
- `--take-other`: This resolves the conflict by using the version of the other branch, ignoring the conflicting changes by the current branch

Resolving content conflicts

When both the branches changed the same file in a conflicting way that is not a text conflict, it is called a **content conflict**. This can happen, for example, if a binary file was changed by both branches, or if a file was changed by one branch and deleted or renamed by the other.

Let's create such a scenario using the following steps:

1. Create a new branch named `content1` based on the trunk.
2. Delete the `hello.c` file and commit the change.
3. Merge the `text1` branch we created in the previous example, which modified the `hello.c` file.

Using the command line:

```
$ bzr branch trunk/ content1
Branched 7 revisions.
$ cd content1/
$ bzr rm hello.c
deleted hello.c
$ bzr ci -m 'removed the c impl'
Committing to: /sandbox/hello/content1/
deleted hello.c
Committed revision 8.
$ bzr merge ../text1
+N  hello.c.OTHER
Contents conflict in hello.c
1 conflicts encountered.
```

If we take a look at the status of the working tree, it will looks similar to the following:

```
$ bzr status
added:
  hello.c.OTHER
unknown:
  hello.c.BASE
conflicts:
  Contents conflict in hello.c
pending merge tips: (use -v to see all merge revisions)
  Janos Gyerik 2013-03-10 say Hi World
```

In this case, Bazaar did not add `hello.c.THIS`, as it would not make sense since we deliberately deleted the file in this branch. The `hello.c.OTHER` file is marked to be added. Perhaps you must rename this file and add it back under version control, otherwise you might lose the work on it that was done by the other branch. Again, `hello.c.BASE` is also created, so that you can compare `hello.c.OTHER` with it, and see what the other branch has changed in it.

This is a good example where the correct resolution is probably simply taking the changes of one branch and ignoring the other:

- If it was a mistake to remove `hello.c` in our branch, then we can resolve the conflict by using `bzr resolve --take-other`, which will add `hello.c` back in the current branch, including the changes by the other branch.
- If `hello.c` is to be considered obsolete and the other branch should not have worked on it, then we can resolve the conflict by using `bzr resolve --take-this`, which will simply ignore the change by the other branch.

Redoing the merge

Before you even begin trying to resolve the conflicts, it is a good idea to retry the merge using a different algorithm. One way to do this is to abort the merge and repeat the same merge command but with an option to specify a different merge algorithm. For example, `--diff3`, `--lca`, `--weave`, or `--merge-type=ARG`.

An easier way is to use the `bzr remerge` command. This command accepts the same options to select a merge algorithm as `bzr merge`, but it has a great advantage that you can specify a subset of the files to run on. In this way, you can try different merge algorithms depending upon what is best for a given file or a set of files. By running only for a subset of files, remerge is more efficient than aborting and re-applying a merge on the entire working tree, as it only needs to work with the selected files.

The merge algorithm that produces the best result varies from case to case. The weave algorithm is known to produce better results when two branches frequently merge from each other.

Another useful option is `--reprocess`, which tries to do additional processing to reduce the size of the conflict regions. It makes the merge command run slower, but usually it is worth a try.

Resolving other types of conflicts

The documentation in `bzr help conflict-types` explains all the conflict types you may see in Bazaar and how they could happen, and it provides hints to resolving them. The general logic is always the same:

1. Check the status of the project, review the content of the files that are in conflict.
2. Use the additional temporary files created by Bazaar to compare the files in different states—base revision, last state in this branch, or state in the other branch.
3. Make the necessary changes in the working tree to fix the project.
4. Use the `--take-this` and --`take-other` shortcuts to resolve the conflicts by accepting the changes from one branch and ignoring from others, if this makes sense in the given situation.
5. Inform Bazaar about the conflicts that have been resolved.

If there are too many conflicts, you may want to re-do the merge using a different algorithm. At any time, you may abort the merge to restore the working tree to the last revision of the current branch and postpone the merge. Or, after resolving all conflicts, you may commit the merge to record a new revision.

Merging a subset of revisions

By default, the `merge` command tries to merge all the revisions of the specified source branch. However, in some cases, it can be useful to merge only a subset of the missing revisions.

Merging up to a specific revision

To merge only up to and including a particular revision of the source branch, specify that revision by using the `-r` or `--revision` options.

As an example, let's merge into the trunk from the `say-hello-to-x` branch at a particular revision. First, let's confirm the missing revisions in order to find a suitable revision:

```
$ cd /sandbox/hello/trunk
$ bzr missing ../say-hello-to-x/ --line --other
You are missing 2 revisions:
8: Janos Gyerik 2013-03-03 python impl can say hello to X
7: Janos Gyerik 2013-03-03 bash impl can say hello to X
```

Revisions `7` and `8` of the `say-hello-to-x` branch are missing in the trunk. Use the following command to merge only up to revision `7` and thus ignore revision `8`:

```
$ bzr merge ../say-hello-to-x/ -r7
 M  README.md
 M  hello.sh
All changes applied successfully.
```

Using the `status` command with the `--verbose` or `-v` flag, we can confirm that only revision `7` has been merged:

```
$ bzr status -v
modified:
  README.md
  hello.sh
pending merges:
  Janos Gyerik 2013-03-03 bash impl can say hello to X
```

Merging a range of revisions

If you specify a range of revisions by using `-r BASE..OTHER` or `--revision BASE..OTHER`, only the revisions through `BASE` to `OTHER` will be merged, excluding `BASE` but including `OTHER`. In the previous example, merging up to revision `7` is equivalent to specifying the range `6..7`.

If `BASE` is a revision that does not exist in the current branch, then the historical relationship between the current branch and the specified branch segment cannot be determined. In such a case, the merge will be treated as "cherry-picking".

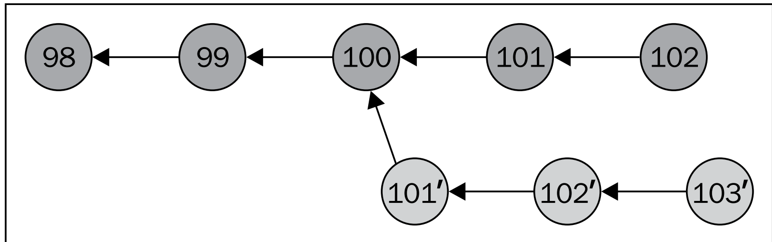
Cherry-picking

Cherry-picking is a merge operation, where the historical relationship between the target and the specified branches segment does not exist, or cannot be determined. This can happen, for example, when merging changes from an unrelated branch, or when merging from a related branch but specifying a range of revisions that does not include a revision that already exists in the destination branch.

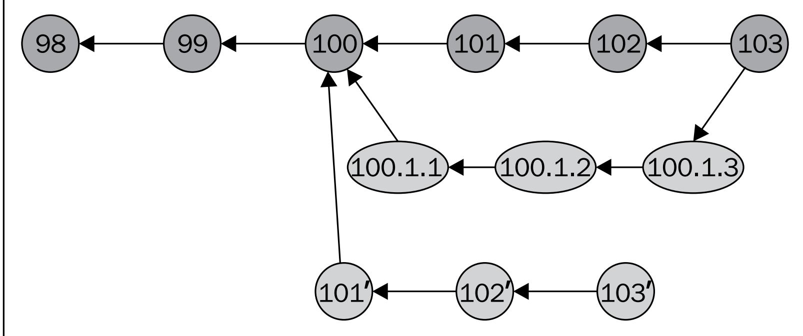
Bazaar does not track the revisions of a cherry-pick merge; in other words, the individual revisions of the specified range will not be preserved, and the changes will be applied as an independent change-set.

Understanding revision numbers

The integer revision numbers are unique per branch, but not unique globally in the project. For example, after creating a branch at revision 100, the next revision in the original branch will be 101, and the next revision in the other branch (with completely different content) will also be 101:



After a branch is merged into the current branch, its revision history is preserved, but to keep revision numbers unique in the current branch, the merged revisions are renumbered using a dotted notation:



The format of revisions in the dotted notation is `BASE.BRANCH.REV`:

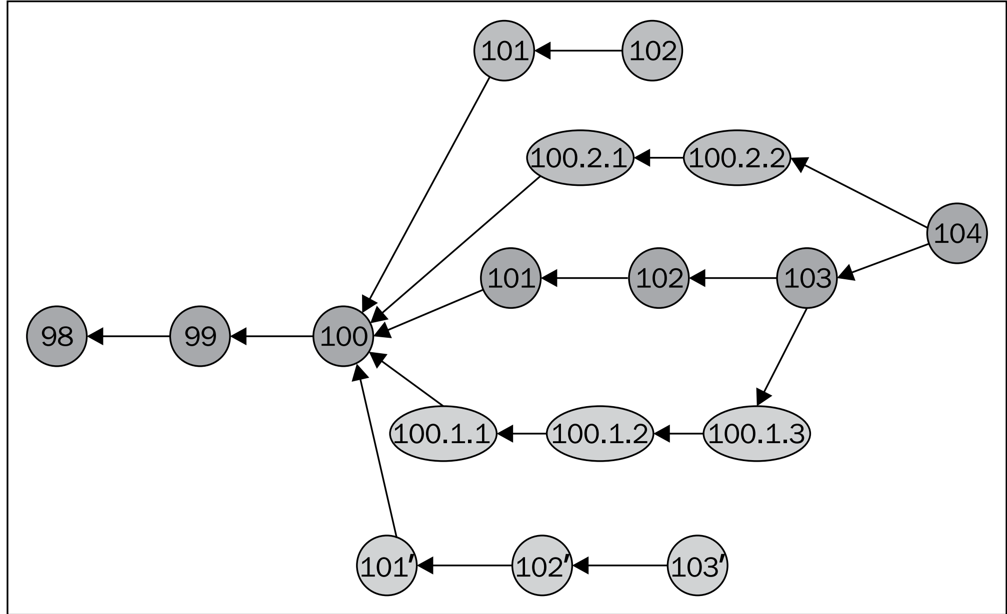
- `BASE`: This is the base revision where the branch started from.
- `BRANCH`: This is the sequence number of the branch, starting from 1, and incremented by 1 every time another branch is merged with the same base revision.
- `REV`: This is the sequence number starting from 1 and incremented by 1 for each merged revision.

In the preceding example, the original revision numbers of the second branch were renamed as follows:

- `101 → 100.1.1`
- `102 → 100.1.2`
- `103 → 100.1.3`

This is because the base revision of the merged branch is `100`, and since this is the first branch merged with this base revision, `BASE = 100`, `BRANCH = 1`, and the revision numbers are incremented as usual starting from 1.

If we were to merge another branch that also started from base revision 100, then the `BRANCH` number component in the renamed revisions will be `2`, as shown in the following example:



www.allitebooks.com

Note that the merge operation does not affect the source branch. As illustrated on the graph, even after the merge operation, the second branch still exists, completely unaffected. Since its unique revisions have been merged into the first branch, we can safely delete the second branch.

If needed, you can recreate the deleted second branch by branching from the first branch at revision `100.1.3`. The result will be a perfect replica of the second branch, with the revisions `100.1.1`, `100.1.2`, and `100.1.3` renamed back to `101`, `102`, and `103`, respectively.

Merging from multiple branches

The merge operation in Bazaar works only with two branches at a time. However, you can merge multiple branches by merging them one by one, as follows:

```
$ cd /path/to/target/branch
$ bzr merge /path/to/source1
$ bzr merge /path/to/source2 --force
$ bzr commit
```

Notice that when merging the second branch, you must use the `--force` flag, because by default Bazaar refuses to merge when there are uncommitted changes in the working tree. Since the first merge leaves the working tree in a changed state, the `--force` flag is necessary to push through the second merge.

It is usually better to merge branches one by one in separate commits, as in this way the history will be cleaner, with the merge commits appearing as distinct larger steps in the evolution in the project.

Mirroring branches

Mirroring branches can be useful in many ways, typically to transfer branches between computers, which is essential when collaborating with others. Another common use is to mirror a branch to an external disk or an archive server as a backup measure.

We have already seen that the `bzr branch` command can create a perfect replica of a branch. However, as the original branch evolves, the replica becomes outdated. Using one of the mirroring commands, it is possible to bring another branch up to date, in sync with the original branch.

Mirroring from another branch

The `bzr pull` command updates the current branch from another one that is some versions ahead, but not diverged. This is useful when the current branch is used as a mirror of another branch, and no revisions are added to it except when updating from the source branch.

We can simulate such a scenario by creating a branch based on an older revision of another branch. For example:

```
$ cd /sandbox/hello
$ bzr branch fix-c/ -r-2 sample-for-pull
Branched 7 revisions.
```

The `sample-for-pull` branch is now one revision behind the `fix-c` branch. We can bring it up-to-date by pulling from its parent branch:

```
$ cd sample-for-pull/
$ bzr pull
Using saved parent location: /sandbox/hello/fix-c/
 M  hello.c
All changes applied successfully.
Now on revision 8.
```

Since we did not specify the branch to pull from, Bazaar used the parent location as a sensible default. As a result, the missing revisions are copied into the current branch and the branch history is also updated accordingly, to be identical to the source branch.

The two branches are now identical, which we can confirm by using the `bzr missing` command:

```
$ bzr missing ../fix-c/
Branches are up to date.
```

When using `bzr pull` without parameters, Bazaar uses the remembered parent branch. To pull from a different branch and remember that location as the new parent branch, use the `--remember` flag. For example:

```
$ bzr pull ../trunk/ --remember
All changes applied successfully.
Now on revision 7.
```

We can confirm that the parent location has been changed by using the `bzr info` command:

```
$ bzr info
Repository tree (format: 2a)
Location:
  shared repository: /sandbox/hello
  repository branch: .

Related branches:
  parent branch: /sandbox/hello/trunk
```

Mirroring from the current branch

The `bzr push` command updates another branch based on the current branch to bring it up-to-date, if it has not diverged. This is useful when the other branch is used as a mirror of the current branch, and no revisions are added to it except when updating from the current branch.

We can simulate such a scenario by creating a branch based on an older revision of another branch. For example:

```
$ cd /sandbox/hello
$ bzr branch fix-c/ -r-2 push-sample
Branched 7 revisions.
```

The `push-sample` branch is now one revision behind the `fix-c` branch. We can bring it up-to-date by pushing to it from the `fix-c` branch:

```
$ cd fix-c/
$ bzr push ../push-sample/
All changes applied successfully.
Pushed up to revision 8.
```

As a result, the missing revisions are copied into the target branch and the branch history is also updated accordingly, to be identical to the source branch.

The two branches are now identical, which we can confirm by using the `bzr missing` command:

```
$ bzr missing ../push-sample/
Branches are up to date.
```

When pushing a branch to another location for the first time, Bazaar remembers to target location as the push branch. We can confirm that by using the `bzr info` command:

```
$ bzr info
Repository tree (format: 2a)
Location:
  shared repository: /sandbox/hello
  repository branch: .

Related branches:
    push branch: /sandbox/hello/push-sample
  parent branch: /sandbox/hello/trunk
```

If you want to push to the same location again, you can use `bzr push` without parameters.

To push to a different location and remember that location as the new push branch use the `--remember` flag. For example:

```
$ bzr push /tmp/push-test --remember
Created new branch.
```

We can confirm that the push location has been changed:

```
$ bzr info
Repository tree (format: 2a)
Location:
  shared repository: /sandbox/hello
  repository branch: .

Related branches:
    push branch: /tmp/push-test
  parent branch: /sandbox/hello/trunk
```

Summary

In this chapter, we have explained what branches are and gave a few practical examples where they can be useful. We have covered the core concepts and commands that should enable you to perform all the basic branch operations, such as creating branches, comparing, merging, and mirroring branches.

The next chapter will build on what you learned here, and show you how to combine the various branch operations to collaborate with others in a small team.

4

Using Bazaar in a Small Team

This chapter explains how to work together with others in a small team. The most natural way to achieve this in Bazaar is by branching and merging from each other.

In essence, this is not very different from working solo and using multiple branches. However, instead of all the branches existing on your computer, they are spread out across multiple computers. Therefore, the branch operations between collaborators must take place over the network.

We will show you a few simple ways of sharing branches with others over the network, as well as a few example workflows that you can use to combine the various branch operations and collaborator branches in an organized manner, for the evolution of the project.

We will cover the following topics in this chapter:

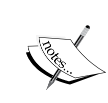
- Collaborating with others
- Sharing branches over the network
- Working with remote branches
- Implementing simple workflows

Collaborating with others

Working together with others is technically very similar to working on multiple branches by yourself—the same way as you would create a new branch based on another one on your computer to work on a new feature. A collaborator can create the new branch on his or her computer to do the same. When the feature is complete, you can merge from a collaborator's branch in the same way as you would merge from any of your own branches.

The most natural way to collaborate with others using Bazaar is by branching and merging from each other. In order for this to work, the branches involved in the operations between you and your collaborators must be accessible by Bazaar in some way:

- You must share a branch in order to let others create their own branches based on it, or to merge from it into their own branches
- Your collaborators must share their completed branches in order to let you or others merge from them

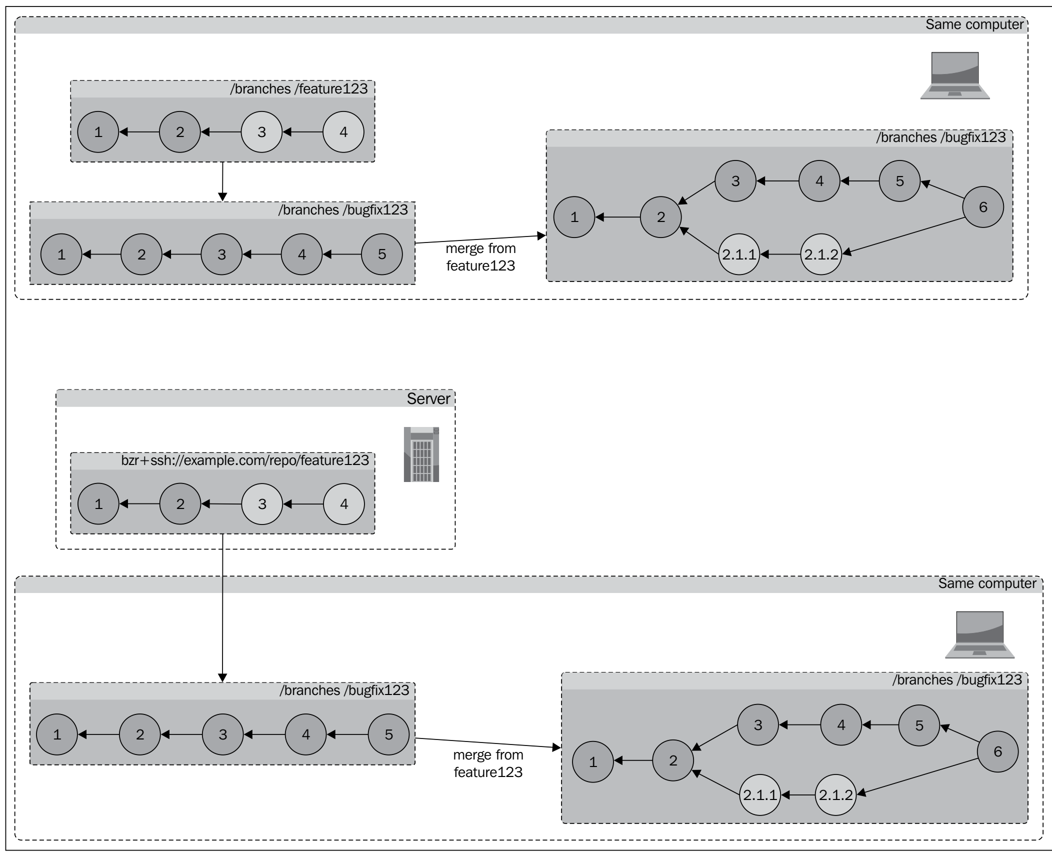


The same logic also applies when you are working solo and using different computers; for example, your desktop and a laptop. In that case too, you need to transfer branches between computers, and technically it is as if the other collaborator is you.

Working with remote branches

We refer to branches on your computer as **local branches**, and branches on other computers as **remote branches**.

All the branch operations in Bazaar work completely transparently and consistently between branches, whether they are local or remote. Even the syntax is the same; you can simply specify remote branches by their remote URLs instead of a local filesystem path as in the case of local branches.



As long as a remote branch is accessible by a remote URL and supported protocol, you can branch, merge, or mirror from it in exactly the same way you do with local branches.

We will show a few simple ways of making branches accessible to others, in order to let collaborators perform the various branch operations, such as branching, merging, or mirroring.

Implementing simple workflows

Another important aspect in collaboration is the manner in which you and your collaborators combine the various branch operations using the various branches in the project.

We will show a few example workflows that you can implement when working with a small number of peers. The workflows will involve a combination of various branch operations, such as branching, merging, and mirroring.

Sharing branches over the network

When you create new branches on your computer, they are normally private and only accessible by you. In order to work with others, you need to make your branches accessible to your collaborators, and likewise they also need to share their branches with you.

In this section, we will focus mainly on the technical details of making branches accessible to others as read-only remote branches. Setting up a full-blown Bazaar hosting server is beyond the scope of this chapter; we will only explain a few relatively simple ways of sharing with others, including any necessary server configuration.

Not all methods may apply to you and your network environment. Feel free to skip subsections and focus on only what is relevant in your particular case.

Specifying remote branches

In all the branch operations, you must specify the source branch to work with by its URL. In case of local branches, the URL is simply the local filesystem path of the branch, as we have seen in the previous chapters. In case of remote branches, the URL starts with a prefix depending upon the transport protocol used.

Bazaar supports many protocols to work with remote branches. The complete list is explained on the `urlspec` help page. For example:

```
$ bzr help urlspec
URL Identifiers


Supported URL prefixes:


  aftp://              Access using active FTP.
  bzr://               Fast access using the Bazaar smart server.
  bzr+ssh://           Fast access using the Bazaar smart server over SSH.
  file://              Access using the standard filesystem (default)
  ftp://               Access using passive FTP.
  http://              Read-only access of branches exported on the web.
  https://             Read-only access of branches exported on the web
using SSL
.
  sftp://              Access using SFTP (most SSH servers provide SFTP).


 ...
```

These are the basic protocols supported by Bazaar; additional protocols are provided by plugins. Although in general all the branch operations work transparently regardless of the protocol, access may be limited to read-only operations, and there may be inherent differences in performance depending upon the protocol.

Protocols that use the Bazaar smart server provide the fastest access. In case of these methods, Bazaar is installed on the server, and incoming branch operations are handled by the `bzr serve` command internally. These protocols are tuned for performance, and can support both read-only and write operations.

Other protocols are slower than the smart server, because they cannot use the `bzr serve` command. Thus the Bazaar client cannot receive assistance from the server side and it has to do more work and transfer more data. These protocols are often referred to as **dumb servers**.

The FTP and SFTP protocols support both read-only and write operations, while the HTTP and HTTPS protocols allow only the read-only access by default.

Write operations can be possible over HTTP and HTTPS by using the **WebDAV** plugin.

Using URL parameters

Depending upon the remote branch, you may need to specify the username, password, and port number as a part of the URL. The general format of a URL is as follows:

```
<protocol>://[user[:password]@]host[:port]/[path]
```

This format works with all the protocols. For example:

```
http://jack@example.com:8080/repos/myproject
bzr+ssh://jack@example.com:8022/repos/myproject
```

Using remote branches through a proxy

If access to the Internet must go through a proxy, you must set the URL of the proxy server in appropriate environment variables:

- `http_proxy`: This is used to access a remote branch via `http://`
- `https_proxy`: This is used to access a remote branch via `https://`
- `ftp_proxy`: This is used to access a remote branch via `ftp://` or `aftp://`

For example, if the proxy URL is `http://proxy:8080/proxy.js`, then you can set it as follows in Windows:

```
$ set http_proxy=http://proxy:8080/proxy.js
$ set https_proxy=http://proxy:8080/proxy.js
$ set ftp_proxy=http://proxy:8080/proxy.js
```

In GNU/Linux and Mac OS X:

```
$ export http_proxy=http://proxy:8080/proxy.js
$ export https_proxy=http://proxy:8080/proxy.js
$ export ftp_proxy=http://proxy:8080/proxy.js
```

Sharing branches using a distributed filesystem

If you and your collaborators have access to some kind of distributed filesystem, such as a network filesystem in GNU/Linux and Mac OS X, or a network share in Windows, then you can create remote branches without additional setup.

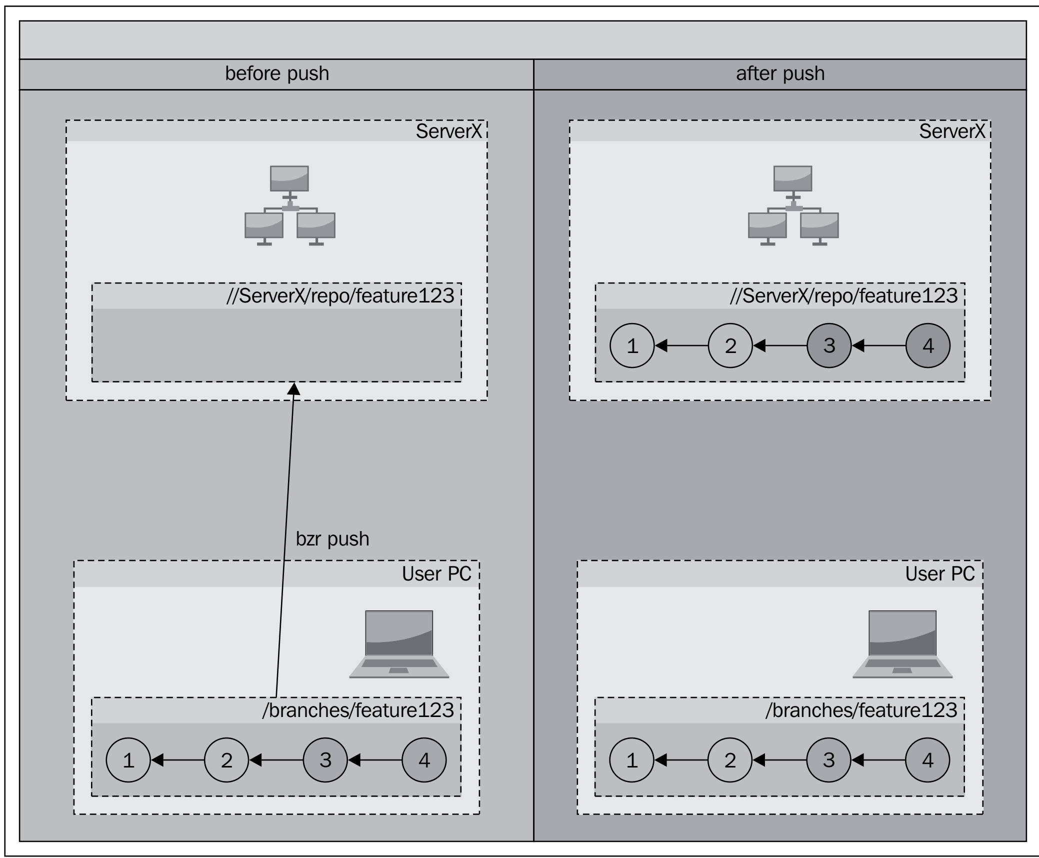
In GNU/Linux and Mac OS X, specify the network filesystem path to create the remote branch. For example:

```
$ bzr push /path/to/nfs/path/to/create --no-tree
```

In Windows, specify the UNC path on the network share. For example:

```
$ bzr push //ServerComputerName/ShareName/path/to/create --no-tree
```

In both the examples, we used the `--no-tree` flag to tell Bazaar to skip creating a working tree. Since the working tree is pointless and potentially confusing to exist in a mirror branch, this is a good measure to save disk space and speed up the `push` operation.



As long as collaborators have read access to the remote branches created in this way, they can work with these branches directly or create local mirror branches based on them. In fast local networks, the network overhead of accessing these branches may be negligible, thus creating local mirrors may be unnecessary.

Sharing branches over SSH

Using an SSH server to share branches with others is very easy to set up with minimal configuration. If you and your collaborators have access to an SSH server, there are several ways to access branches:

- Using Bazaar's smart server and individual SSH accounts
- Using Bazaar's smart server with a shared restricted SSH account
- Using individual SSH accounts with SFTP

Using Bazaar's smart server provides fast access to branch data. However, for this to work, Bazaar must be installed on the server, and the `bzr` command must be included in the execution path of the user account used when connecting to the SSH server. In this case, the incoming branch operations are handled by the `bzr serve` command internally, which is tuned for fast performance.

Using individual SSH accounts

If Bazaar is installed on the server, you can benefit from using the smart server by constructing the remote URL of the branch in the following format:

```
bzr+ssh://[user@]host/[path]
```

Here:

- `user` is the username of your SSH account
- `host` is the hostname of the SSH server
- `path` is the absolute path of the Bazaar branch in the server's filesystem

For example, if you have a user account named `jack` on the SSH server `example.com`, and there is a Bazaar branch at the path `/srv/bzr/projectx` on the server, then you can access the branch by the URL `bzr+ssh://jack@example.com/srv/bzr/projectx`, as follows:

```
$ bzr info bzr+ssh://jack@example.com/srv/bzr/projectx
```

Collaborators can use their own SSH account to access the branch by simply replacing the username in the URL. Keep in mind that standard filesystem permissions apply; collaborators can only access the branches if their user accounts have the appropriate filesystem permissions on the branch paths.

When referring to branches under your own home directory, you can replace the home directory part in the absolute path with `~ (tilde, the home directory indicator in UNIX systems)`. For example, the following commands are equivalent:

```
$ bzr info bzr+ssh://jack@example.com/home/jack/bzr/projectx
$ bzr info bzr+ssh://jack@example.com/~/bzr/projectx
```

Since, by default, other users don't have write permission to your home directory, it can be a suitable place to put your branches in order to provide strictly read-only access to others.

Using individual SSH accounts with SFTP

If you cannot or don't want to install Bazaar on the SSH server, another option is to use SFTP instead, if it is enabled on the server. You can construct the remote URL in the same way as in the previous section, but replace the `bzr+ssh://` prefix in the branch URL with `sftp://`.

The main difference between these two modes is that when using SFTP, the Bazaar smart server is not used at the server side, therefore performance is slower. Nonetheless, this can be a suitable option if installing Bazaar on the server is not possible.

Using a shared restricted SSH account

Instead of creating individual SSH accounts for each collaborator, an interesting alternative is to use a shared SSH account with command restrictions.

This setup requires that collaborators use the SSH public key authentication when connecting to the server, and that appropriate access permissions to the branches are configured in the `~/.ssh/authorized_keys` file of the shared SSH account.

Let's suppose that:

- There is a shared repository on the server in `/srv/bzr/projectx`
- You want to let `jack` create his branches in `/srv/bzr/projectx/jack`
- You want to let `mike` create his branches in `/srv/bzr/projectx/mike`
- The shared repository is owned by the user `bzruser`

To make this work, add the following two lines to the `~/.ssh/authorized_keys` file of `bzruser`:

```
command="bzr serve --inet --allow-writes --directory=/srv/bzr/p
rojectx/jack",no-agent-forwarding,no-port-forwarding,no-pty,no-us
er-rc,no-X11-forwarding PUBKEY_OF_JACK
```

```
command="bzr serve --inet --allow-writes --directory=/srv/bzr/pro
jectx/mike",no-agent-forwarding,no-port-forwarding,no-pty,no-user-r
c,no-X11-forwarding PUBKEY_OF_MIKE
```

Replace `PUBKEY_OF_JACK` and `PUBKEY_OF_MIKE` with the SSH public key of Jack and Mike, respectively. For example an SSH public key looks similar to the following:

```
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAo6+TOzByRt9EVUjpMBs5kRft9SSPa
mI3cRlvaX4DuMbRqjtfkRTO4tik+MAWaFeIHyO5EsdFBGp+XVH9BMqehXdjAQga4Wa2
oGX/w7bn+O+gdIoJE2wzMlGV2eXcaW2PKdDIqQpUn0n+xX68vjRaCiZmqGXWhVej3cVi9
dtIwIQMrcIF4T+4wONic09UjPXZKbjL2GmkzsR6SMQJBomr4TUcRgyaR5ija9R8AzvsSdN
eDKkVwf83lva3jruwEMute3aZFulM5JqvjFIFqooAlSjWjdniF8ZdweeN1c2Q2QH+eCl48h
Y2drUsdZ+oQH+xp8x6llkZiDWFE/RZLa3Glw== Joe
```

The `command` parameter restricts the login shell to the `bzr serve` command.
In this way, the users will not be able to do anything else on the server except run Bazaar commands. The `--directory` parameter further restricts Bazaar operations to the specified directory. To give only read-only access, simply drop the `--allow-writes` flag.

The other options on the line after `command` are to make the SSH session as restricted as possible, as a good measure for security.

When accessing branches in this setup, the `path` component in the branch URL must be relative to the directory specified in the `authorization` line. For example, Jack can access his branch in `/srv/bzr/projectx/jack/feature-123`, as follows:

```
$ bzr info bzr+ssh://bzruser@example.com/feature-123
```

The drawback of this setup is that you can only have one configuration line per SSH key. One way to work around this can be by adding another shared SSH account, configuring it to have read access to the shared repository, and creating the `~/.ssh/authorized_keys` file as follows:

```
command="bzr serve --inet --directory=/srv/bzr/projectx",no-agent-
forwarding,no-port-forwarding,no-pty,no-user-rc,no-X11-forwarding P
UBKEY_OF_JACK
```

```
command="bzr serve --inet --directory=/srv/bzr/projectx",no-agent-f
orwarding,no-port-forwarding,no-pty,no-user-rc,no-X11-forwarding PUBKE
Y_OF_MIKE
```

Again, replace `PUBKEY_OF_JACK` and `PUBKEY_OF_MIKE` with the SSH public key of `jack` and `mike`, respectively. Notice that we removed the `--allow-writes` flag and adjusted the `--directory` parameter to specify the shared repository rather than the per-user directories.

Using SSH host aliases

If you use the same SSH server frequently, it can be convenient to set up an alias in your `~/.ssh/config` file as follows (only in GNU/Linux and Mac OS X):

```
Host repo
Hostname example.com
User jack
```

In this way, you can omit the username and shorten the server name in the URL:

```
$ bzr info bzr+ssh://repo/~/myproject/mybranch
```

Using a different SSH client

To use a different SSH client instead of Bazaar's default, you can specify the path of another SSH client using the `BZR_SSH` variable. This can be especially useful in Windows, if you use **PuTTY** to store your SSH private keys. You can tell Bazaar to use PuTTY by setting `BZR_SSH`, as follows:

```
set BZR_SSH=c:\program files\putty\plink.exe
```

Sharing branches using bzr serve

You can use the Bazaar smart server directly to listen to incoming connections and serve branch data.

Use the `bzr serve` command to start the smart server. By default, it listens on port `4155` and serves the branch data from the current working directory in read-only mode. It has several command-line parameters and flags to change the default behavior. For example:

- `--directory DIR`: This specifies the base directory to serve the branch data from instead of the current working directory
- `--port PORT`: This specifies the port number to listen on instead of the default `4155`
- `--allow-writes`: This allows write operations instead of strictly read-only

Use the -h or --help flags to see the list of supported command-line parameters.

Branches served in this way can be accessed by URLs in the following format:

```
bzr://host/[path]
```

Here, host is the hostname of the server, and path is the relative path from the base directory as configured in the server process.

For example, if the server is example.com, and the smart server is configured to use the directory /srv/bzr/repo, and there is a Bazaar branch at the path /srv/bzr/repo/projectx/feature-123, then the branch can be accessed as follows:

```
$ bzr info bzr://example.com/projectx/feature-123
```

The advantage of this setup is that the smart server provides good performance. On the other hand, it completely lacks authentication.

Sharing branches using inetd

On GNU/Linux and UNIX systems, you can configure inetd to start the bzr serve command automatically as needed, by adding a line in the inetd.conf file as follows:

```
4155  stream  TCP  nowait  bzruser  /usr/bin/bzr /usr/bin/bzr serve
--inet --directory=/srv/bzr/repo
```

Here:

- 4155 is the port number where the Bazaar server should listen for incoming connection.
- bzruser is the user account the bzr serve process will run as.
- /usr/bin/bzr is the absolute path of the bzr command.
- /usr/bin/bzr serve --inet --directory=/srv/bzr/repo is the complete command to execute when starting the server. The --directory parameter is used to specify the base directory of Bazaar branches.

Once configured, this setup works exactly in the same way as explained in the previous section, and it has the same advantages and disadvantages.

Sharing branches over HTTP or HTTPS

If you have a website, and you have an SSH or SFTP access to the server hosting the website, then you can make your branches available to others in read-only mode by pushing them to somewhere visible on the website.

For example, if your website's files are in `/var/www/example.com`, you can push your Bazaar branches `to /var/www/example.com/bzr/projectx/` and let others access them via the URL `http://example.com/bzr/projectx`, as follows:

```
$ bzr info http://example.com/bzr/projectx
```

Although in this setup the Bazaar client cannot receive assistance from the server side, it can figure out the necessary information for completing requests by downloading from the appropriate `.bzr` directories. As a result, the performance is significantly slower compared to a smart server, and this kind of setup is referred to as dumb server.

Although normally the HTTP and HTTPS protocols allow only read-only access, write operations can be possible by using the WebDAV plugin.

Working with remote branches

Since all the branch operations work transparently and consistently between remote and local branches, you can work with remote branches directly. However, it is often more practical to work with remote branches in an indirect way, using mirror branches. For example:

- You can speed up branch operations by using local mirror branches instead of working with a remote branch directly
- If your collaborators don't have direct access to your local branches, you can provide remote mirror branches at a more accessible location.

In this section, we will explain various practical considerations when working with remote branches. In particular, it is important to become very comfortable with the various mirroring operations in order to work with remote branches with ease.

Working with remote branches directly

All the branch operations work exactly in the same way with remote branches as with local branches. For example:

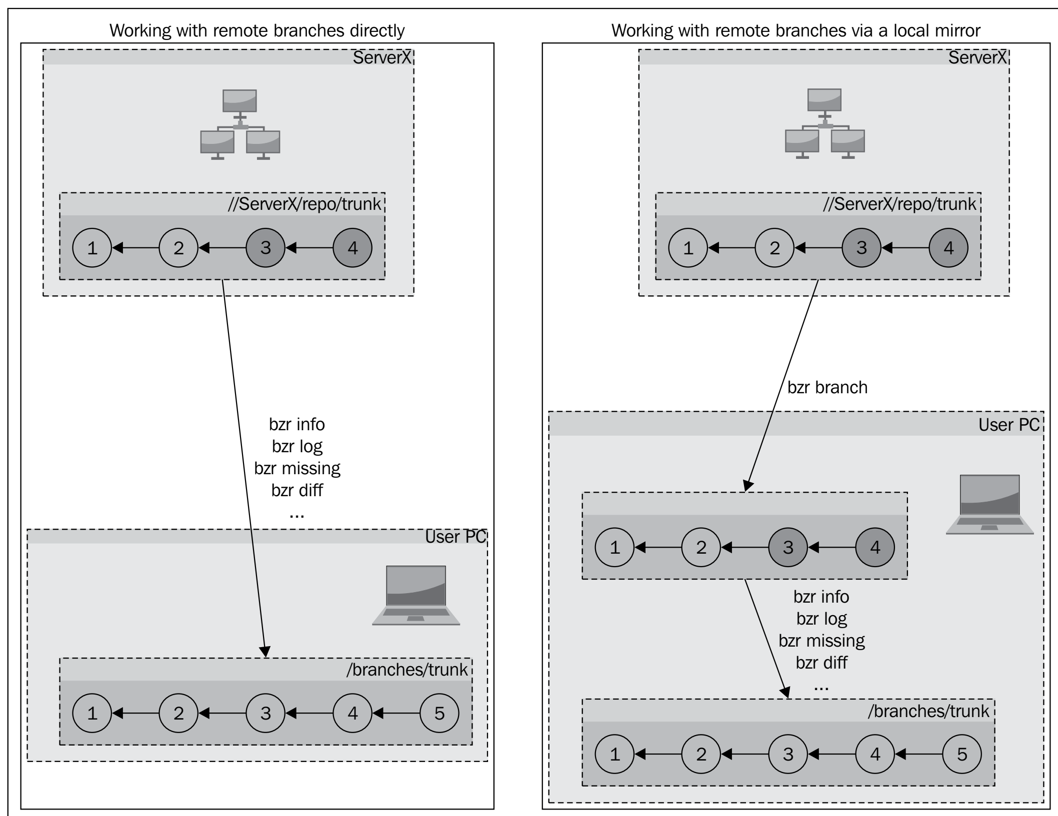
- `bzr info REMOTE_URL`: This prints basic information about the remote branch.
- `bzr log REMOTE_URL`: This shows the revision history of the remote branch.
- `bzr qlog REMOTE_URL`: This shows the revision history of the remote branch in Bazaar Explorer.
- `bzr branch REMOTE_URL [TO_LOCATION]`: This creates a new local branch as a perfect replica of the specified remote branch. The second parameter can be used to specify the path of the new local branch, otherwise the branch is created as a subdirectory in the current directory.
- `bzr missing REMOTE_URL`: This compares the current branch with the specified remote branch and prints the list of missing revisions in both.
- `bzr diff --new REMOTE_URL`: This compares the current branch with the specified remote branch and prints the differences going from the current branch to the remote branch. Use `--old REMOTE_URL` to show the differences in the other direction.
- `bzr merge REMOTE_URL`: This merges the specified remote branch into the current branch.
- `bzr remerge [FILE...]`: This performs a redo of the merge, possibly using a different merge algorithm on all files or selected files only.
- `bzr push REMOTE_URL`: This makes the remote branch a mirror of the current branch. It works only if the two branches have not diverged.
- `bzr pull REMOTE_URL`: This makes the current branch a mirror of the remote branch. It works only if the two branches have not diverged.

However, keep in mind that due to the overhead of transferring data over the network, all these operations will be slower as compared to when working with local branches.

Using local mirror branches

When working with remote branches, the transport over the network poses an overhead—all branch operations are slower because the data must travel over the network. In addition, when working with the same remote branches repeatedly, it can be tedious to re-enter their possibly long and complicated URLs.

In such situations, instead of working with a remote branch directly, it is more practical to create a local mirror branch and perform branch operations using the local mirror instead of the remote branch. Provided that the remote branch has not changed since the local mirror was created, the result of all branch operations will be equivalent regardless of using the remote branch directly, or its local mirror.



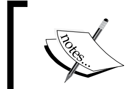
This is especially practical when merging from a remote branch. Before performing the merge, you might want to run a series of commands to inspect the branch; for example, using the `bzr log`, `bzr missing`, and `bzr diff` commands, or their equivalents in Bazaar Explorer. After performing the merge you might want to redo the merge with `bzr remerge` by using a different algorithm.

Running multiple commands directly on a remote branch while it hasn't changed is inefficient, as Bazaar would have to transfer data over the network repeatedly. A more efficient way is to first create a local mirror, and perform all the branch operations you may need on the local mirror instead of the remote branch.

Creating a local mirror

You can create a local mirror by simply creating a local branch based on a remote branch, by using the `bzr branch` command or its equivalent in Bazaar Explorer.

The local branch will be identical to the original remote branch, and the result of all branch operations using the local branch as the source branch will be identical to using the remote branch directly, but more efficient because the network overhead is completely eliminated.



> When creating a branch based on another branch, the source branch is called the **parent branch** of the new branch.

Keep in mind that the local branch is only a mirror as long as you use it that way. Technically speaking, a mirror branch is no different from any of your other local branches. If you make changes and commit new revisions to a mirror branch, it will have diverged from its parent. If you keep it as a pristine copy without committing any new revisions to it, then you will be able to bring it up-to-date with its parent later.

Using a shared repository

When working with multiple branches, it is always a good idea to use a shared repository, as this eliminates the unnecessary copying of revision data between branches, saving disk space and speeding up branch operations.

Using a shared repository is especially useful when creating new local branches based on remote branches, because Bazaar can re-use the revisions that already exist in the shared repository, thus reducing the amount of data transferred over the network, greatly mitigating the network overhead.

Updating a local mirror

A local mirror branch is not updated automatically when new revisions are added in its parent branch. This comes from the fact that a mirror branch is no different from any other branch—a local branch is only a mirror because you use it that way.

To bring a local branch up-to-date with its parent, use the `bzr pull` command, or in Bazaar Explorer the large **Pull** button in the toolbar.

This works only if the local branch has not diverged from its parent; that is, you have not committed any new revisions to it yourself. The `pull` operation only makes sense between branches that have not diverged, otherwise Bazaar will fail with an error, since the local branch is no longer a mirror of its parent.

As long as the local branch is not changed in another way, you can pull from its parent branch repeatedly, to download any new revisions that may have been added at the remote side. If the local branch is already up-to-date, pulling again will simply do nothing.

To distinguish local branches that you intend to use as read-only local mirrors, it is a good idea to create them inside a subdirectory within the shared repository. For example, named `mirrors`.

Using remote mirror branches

Branches that you create on your computer are normally not accessible by others. In order to collaborate with others, you need to make your branches available to them somehow.

Providing direct access to your local branches can be difficult or often impossible due to the network topology between your computer, and the computers of your collaborators. A common solution is to provide remote mirrors of your local branches at a location that is accessible by your peers. Since a remote mirror branch is identical to its parent branch, your collaborators can work with the mirror branches and get the same result of all branch operations as if they were accessing your local branches directly.

Creating a remote mirror

You can create a remote mirror branch from the current branch by using the `bzr push` command and specifying a remote URL that supports write operations, such as `bzr://`, `bzr+ssh://`, and `ftp://, aftp://`. For example:

```
$ bzr push bzr://example.com/path/to/create
$ bzr push bzr+ssh://jack@example.com/path/to/create
$ bzr push aftp://jack@example.com/path/to/create
```

Of course, in order to try these operations, you need to have access to a remote server that is configured appropriately. See the explanation earlier in the *Sharing branches* section.

Using a shared repository

When creating multiple remote branches on the same server, it is always a good idea to use a shared repository, as this eliminates the unnecessary copying of revision data between branches, therefore saving disk space and speeding up branch operations.

Using a shared repository is especially useful when creating new remote branches, because Bazaar can re-use the revisions that already exist in the remote shared repository, thus reducing the amount of data transferred over the network and greatly mitigating the network overhead.

Updating a remote mirror

A remote mirror branch is not updated automatically when new revisions are added in its parent branch. This comes from the fact that a mirror branch is no different from any other branch—a remote branch is only a mirror because you use it that way.

To bring a remote branch up-to-date with its local parent, use the `bzr push` command, or in Bazaar Explorer the large **Push** button in the toolbar.

This works only if the remote branch has not diverged from its parent. The `push` operation only makes sense between branches that have not diverged, otherwise Bazaar will fail with an error, since the remote branch is no longer a mirror of its parent.

As long as the remote branch is not changed in another way, you can push to it from the same local branch repeatedly, to upload any new revisions that you have added locally. If the remote branch is already up-to-date, pushing again will simply do nothing.

Using branches without a working tree

If you intend to use a branch as a mirror, then it makes sense to skip creating a working tree. By default, Bazaar creates a working tree in new branches, but a working tree is optional in general, and you can save disk space and speed up operations by not creating it.

Creating a local branch without a working tree

To create a branch without a working tree, use the `--no-tree` flag with the `bzr branch` command. For example:

```
$ bzr branch lp:~bzrbook/bzrbook-examples/hello-start --no-tree
Branched 6 revisions.
```

The directory of the new branch created in this way is not populated with files; you will find only the `.bzr` directory inside and none of the files of the project. All the branch operations will work with such a branch, except operations that require a working tree such as the `bzr add`, `bzr remove`, and `bzr commit` commands.

Creating or removing the working tree

You can use the `bzr reconfigure` command to create or remove the working tree from the directory of an existing branch.

To remove an existing working tree from a branch, use the `--branch` flag. For example:

```
$ bzr reconfigure --branch
```

To create a working tree in a branch that doesn't have one, use the `--tree flag`. For example:

```
$ bzr reconfigure --tree
```

Within a shared repository, branches without a working tree are called repository branches, and branches with a working tree are called a repository tree. Outside a shared repository, branches without a working tree are called standalone branches, and branches with a working tree are called a standalone tree:

	With working tree	**Without working tree**
Shared repository	Repository tree	Repository branch
Standalone	Standalone tree	Standalone branch

You can confirm the configuration of a branch by using the `bzr info` command. The first line of the output tells the name of the configuration. For example:

```
$ bzr init /tmp/branch --no-tree
Created a standalone branch (format: 2a)
$ bzr info /tmp/branch/
Standalone branch (format: 2a)
Location:
  branch root: /tmp/branch
$ bzr reconfigure --tree /tmp/branch/
$ bzr info /tmp/branch/
Standalone tree (format: 2a)
Location:
  branch root: /tmp/branch
```

Reconfiguring working trees in a shared repository

In the default setup of a shared repository, new branches are created with a working tree. You can change that behavior using the `--with-no-trees` and `--with-trees` flags.

To turn off working trees by default:

```
$ bzr reconfigure --with-no-trees
```

To turn on working trees by default:

```
$ bzr reconfigure --with-trees
```

The configuration with working trees enabled is called **shared repository with trees**, and without working trees is called simply **shared repository**. You can confirm the configuration of a shared repository using the `bzr info` command; the first line of the output tells you the name of the configuration. For example:

```
$ bzr init-repo /tmp/repo
Shared repository with trees (format: 2a)
Location:
  shared repository: /tmp/repo
$ bzr reconfigure --with-no-trees /tmp/repo/
$ bzr info /tmp/repo/
Shared repository (format: 2a)
Location:
  shared repository: /tmp/repo
```

The working trees setting affects the default behavior of the `bzr branch` and `bzr push` commands when used to create branches within the shared repository. Changing the setting has no effect on existing branches; you must use the `bzr reconfigure` command explicitly to create or remove working trees in the existing branches.

Creating remote branches without a working tree

When creating a remote branch, by default, Bazaar will try to create a working tree if the protocol supports it. Use the `--no-tree` flag to override this behavior and skip creating a working tree.

When creating remote branches for the purpose of mirroring, it is usually better to skip creating a working tree for two reasons:

- Save disk space and speed up push operations
- Remote mirror branches are meant to be used in branch operations only, thus having a working tree is pointless and may become confusing

Slicing and dicing branches

When working with multiple local and remote branches, it is important to be fully comfortable with mirroring branches locally or remotely. In order to master these operations, perhaps it helps to play around with them to see how they can be combined in different ways that result in perfectly equivalent branches.

Given two branches `branchA` and `branchB` that have a common base revision (common ancestor) but have diverged in different directions, let's define a few additional branches to work with:

- Let `branchA_B` be the result of merging from `branchB` to `branchA`
- Let `branchB_A` be the result of merging from `branchA` to `branchB`
- Let `branchA_old` be the result of branching from `branchA` at a past revision
- Let `branchB_old` be the result of branching from `branchB` at a past revision

Then the following statements are all true:

- The result of `bzr branch branchA_B -rlast:2` is identical to `branchA`
- The result of `bzr branch branchB_A -rlast:2` is identical to `branchB`
- The result of `bzr pull -d branchB branchA_B` is identical to `branchA_B`
- The result of `bzr pull -d branchA branchB_A` is identical to `branchB_A`
- The result of `bzr push -d branchA_B branchB` is identical to `branchA_B`
- The result of `bzr push -d branchB_A branchA` is identical to `branchB_A`
- There is a revision `REV` in `branchA_B` such that the result of bzr `branch -rREV branchA_B` is identical to `branchB`
- There is a revision `REV` in `branchB_A` such that the result of `bzr branch -rREV branchB_A` is identical to `branchA`
- The result of `bzr push -d branchA_B branchA_old` is identical to `branchA_B`
- The result of `bzr push -d branchB_A branchB_old` is identical to `branchB_A`

To verify the preceding statements, you can recreate the branches in the original assumptions, or download our sample branches using the following commands:

```
$ bzr init-repository /tmp/slicing-and-dicing
$ cd /tmp/slicing-and-dicing
```

```
$ bzr branch lp:~bzrbook/bzrbook-examples/branchA_B
$ bzr branch lp:~bzrbook/bzrbook-examples/branchB_A
$ bzr branch branchA -rlast:3 branchA_old
$ bzr branch branchB -rlast:3 branchB_old
```

Based on these branches you can verify all the preceding statements, make comparisons, and test your own assumptions. For example:

- `bzr missing`: This should print an empty output for equivalent branches
- `bzr diff`: This should print an empty output for equivalent branches
- `bzr merge`: This should print `Nothing to do.` if branches are equivalent
- `bzr push`: This confirms when it is possible to push and when not, and the result of the push operation
- `bzr pull`: This confirms when it is possible to pull and when not, and the result of the pull operation

Going through these examples should solidify your understanding of the branch, push and pull operations, and thereby enable you to slice and dice branches at ease.

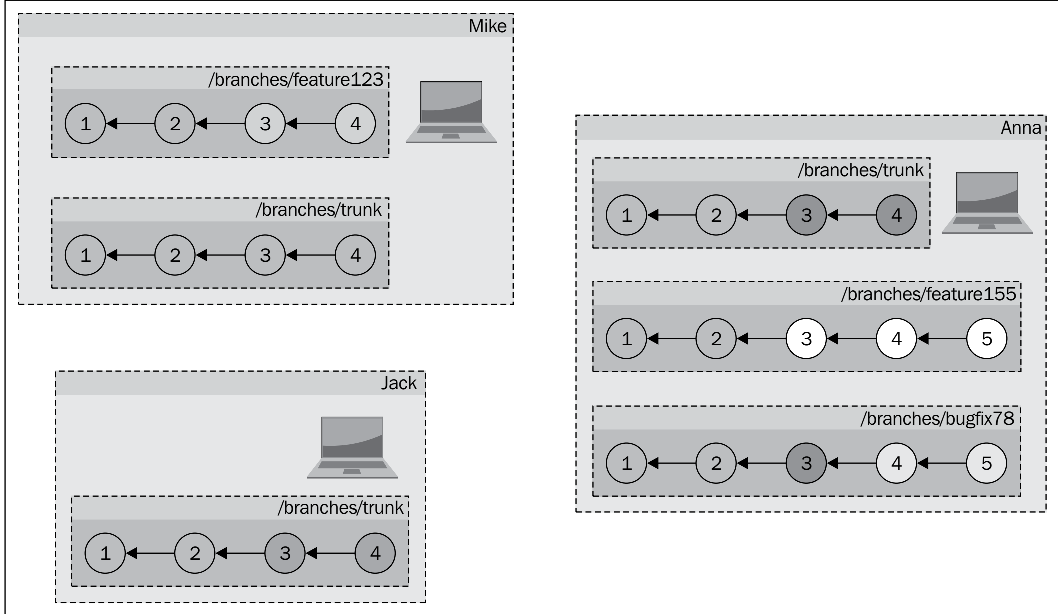
Implementing simple workflows

There are many ways in which you can combine the various branch operations on the various branches, created by all the collaborators in a project. We will introduce two simple workflows suitable for small teams, which you can implement directly, or use as an example to design your own workflows when collaborating with others.

For simplicity, we will assume direct access between collaborator branches. In practice, you may have to work with remote mirrors of your collaborators' branches and use local mirrors for practical reasons. However, such technical details should not affect the main principles of the workflows.

Using independent personal branches

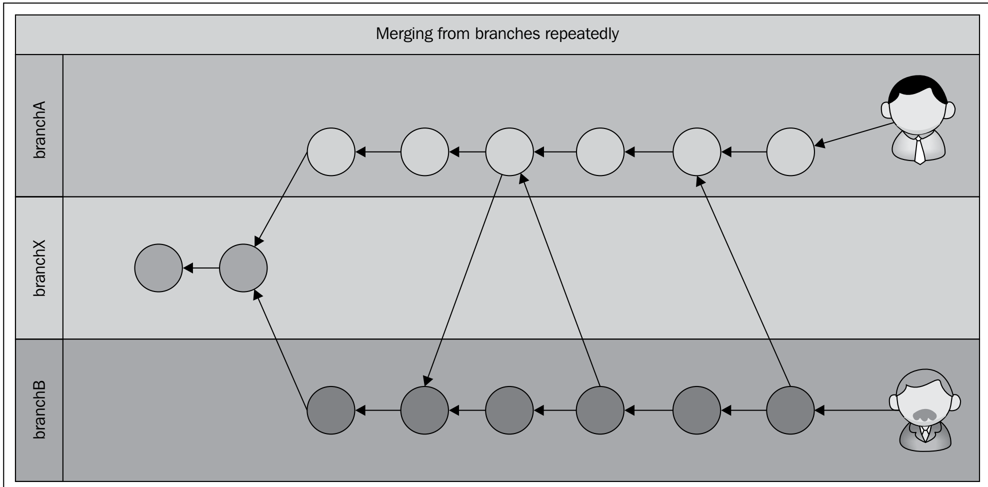
In this workflow, each collaborator has his or her own main personal branch that others can access in read-only mode—they can branch, merge, or pull from it, but cannot commit or push to it. Collaborators work on the project by committing their changes in their own branch, and by occasionally merging from the branches of others:



This workflow can be suitable in very small teams with only a few members, especially in the beginning of a new project. As long as the team members merge from each other frequently, their main branches will be quite similar. Over time and as the team grows, a branch may emerge as the official, if its owner merges relatively more often from other team members, thus making the branch the most complete and up-to-date point of reference.

Merging from branches repeatedly

In this workflow, team members inevitably merge from each other's main branch repeatedly. This means that not only are there repeated merges from `branchA` to `branchB`, but at the same time there are also repeated merges in the other direction, from `branchB` to `branchA`:



In this example, there are two users, Jack and Mike, each working on a single branch, and occasionally merging from each other. The timeline of their actions is as follows:

Step no.	Jack's branch	Mike's branch	Summary after the step
1	Branched from `branchX`	Branched from `branchX`	The two branches are identical to each other and their common parent
2	Added new revisions	Added new revisions	The two branches have diverged
3		Merged from Jack	Jack's revisions are copied into Mike's branch
4	Merged from Mike		Mike's revisions are copied into Jack's branch, plus Mike's merge commit

The somewhat tricky part is what happens to "merges of merges" in step 4. In this step, Mike's branch contains all the changes of the project, including Jack's revisions. That is, when Jack performs the merge to get Mike's changes, the source branch contains his own changes too.

Bazaar handles this correctly, because in step 3 the merge operation records not just the changes in the content of the project's files, but also the unique revision IDs of the revisions by Jack. In this way, when merging from Mike's branch into Jack's branch in Step 4, Bazaar recognizes that the changes involved in Jack's revisions should not be re-applied.

Handling criss-cross merges

When two branches merge the same changes and then merge from one another, it results in a so-called "criss-cross" in the branch history. This can cause problems with the three-way merge algorithm, which is the default method in Bazaar to handle merges.

The principle of the three-way merge algorithm is finding a common base revision, in order to determine whether the differences in the two branches are due to one side adding lines or another side removing lines. In case of a criss-cross, there is no good choice for a base—selecting a recent merge point could cause one side's changes to be silently discarded, while selecting older merge points could cause more than necessary conflicts to be emitted.

The **weave algorithm** is not affected by this problem, because instead of using a base revision to detect the cause of differences, it uses so-called **line-origin detection**.

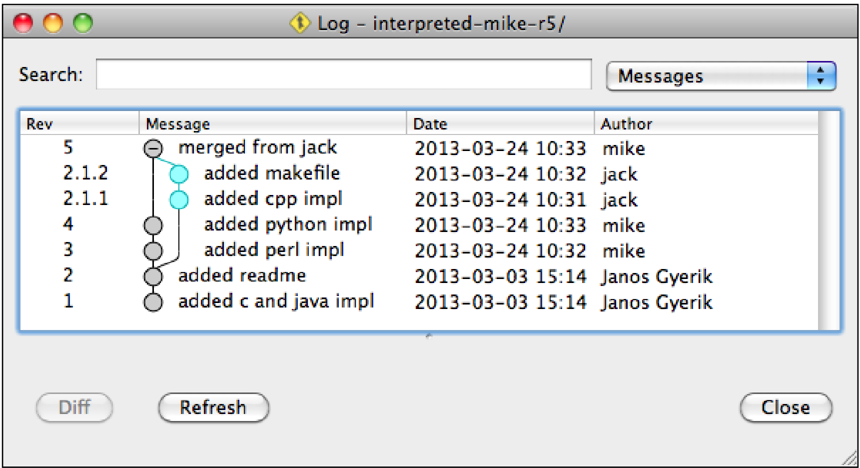
If you encounter too many conflicts with the three-way merge, it can be a good idea to redo the merge on selected files or the entire project using the `bzr remerge` command using the `--weave` flag.

For more details on how the weave algorithm works, refer to the following pages in the Bazaar documentation, and on Wikipedia:

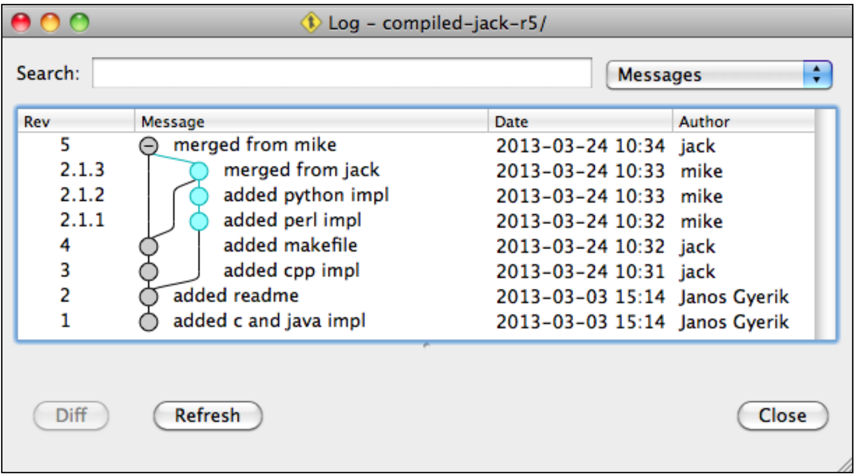
- `http://doc.bazaar.canonical.com/beta/en/user-reference/criss-cross-help.html`
- `http://en.wikipedia.org/wiki/Merge_(revision_control)#Weave_merge`

Viewing the history from different perspectives

It is important to keep in mind that Bazaar shows the revision history from the perspective of the current branch. In other words, each collaborator in this workflow will see the history differently. For example, Mike sees the history as follows:



While Jack sees the history as follows:



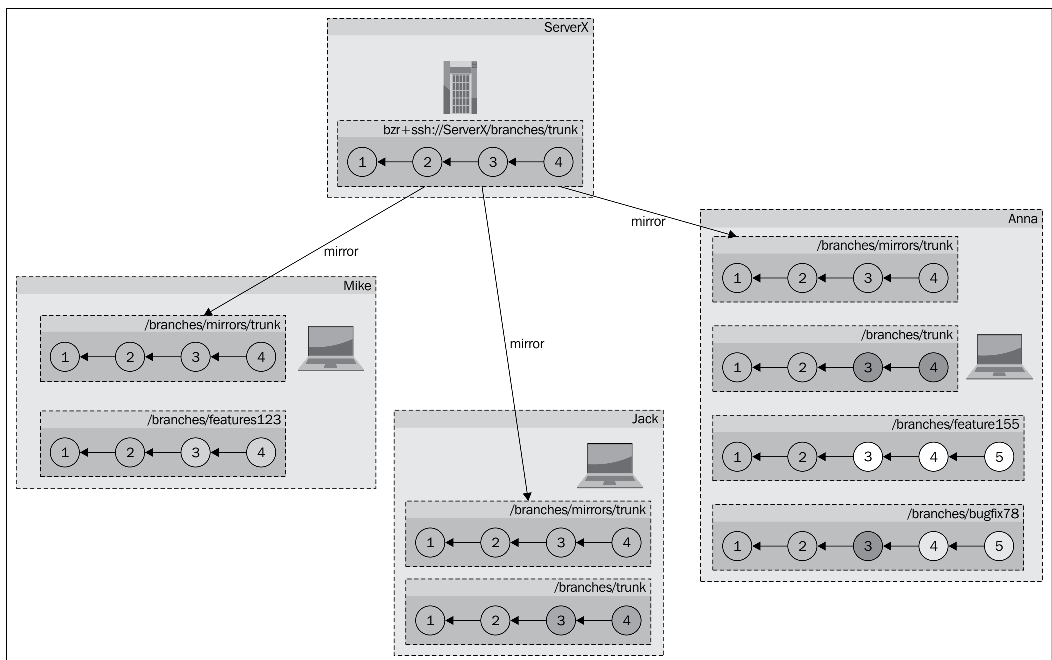
In this example, both the branches have exactly the same content in terms of project files, but their view of the history is different. This is because Bazaar uses increasing integer revision numbers for commits in the current branch, and renames merged revisions using a dotted notation.

Also note that at this point, Jack's branch contains all the revisions of Mike's branch, but the converse is not true—revision 5 in Jack's branch does not exist in Mike's branch. The last branch that merged the other branch, will always have one revision not in the other branch—the revision that committed the merge.

At this point, Mike can pull from Jack to make the two branches identical. As a result, both branches will have Mike's perspective of the revision history. It might be a good idea to pull from each other whenever possible instead of a merge, as that would reduce the number of criss-crosses in the branch history.

Using feature branches and a common trunk

In this workflow, collaborators do all their work on dedicated feature branches. When a feature branch is completed, its owner or another team member merges it into the common trunk. For simplicity, we will assume that the common trunk is available at a central location, and all collaborators have write access to it by push operations.



This workflow can be suitable in small- to medium-sized teams, because it is not complicated to implement, and the common trunk helps in keeping the project organized.

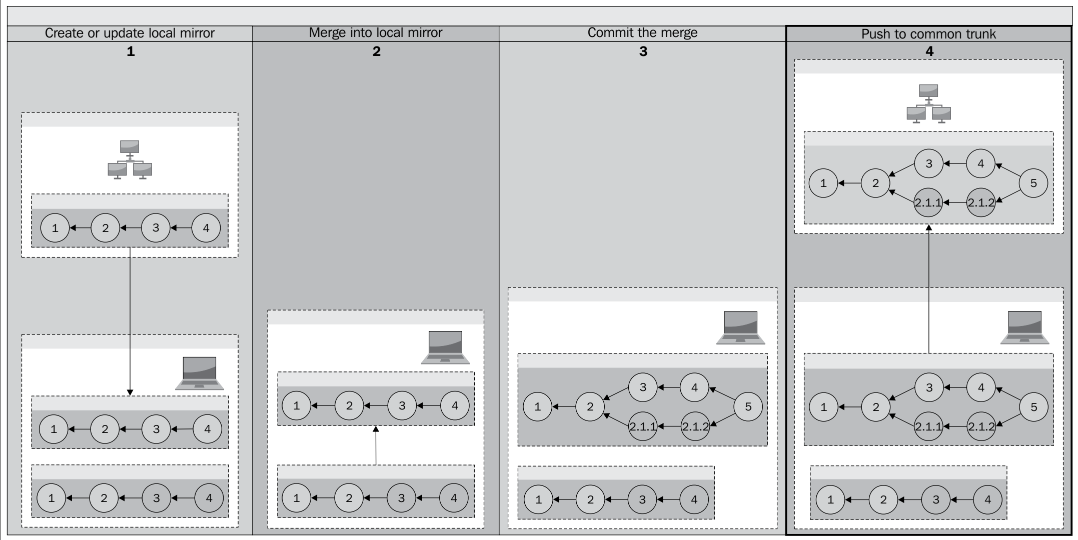
The workflow has several interesting features:

- The official latest version of the project can be clearly identified as the common trunk
- It enforces the good practice of using feature branches
- It facilitates the good practice of "code reviews"

Merging into a common remote trunk

Since merge operations can only be applied to local branches, collaborators must keep a local mirror of the common trunk. Merging a feature branch can be performed by the following steps:

1. Create or update a local mirror
2. Merge from the feature branch into the local mirror
3. Test the improvements well and commit the merge
4. Push from the local mirror to the common remote trunk



Use the `branch` command when creating the local mirrors for the first time, or to recreate a pristine copy. Use the `pull` command to update an existing local mirror that may be out of date. If you have inadvertently changed the local branch, use the `--overwrite` flag with the `pull` command to restore its pristine state.

If the feature branch is not your own but a remote branch created by another team member, then instead of merging from it directly, it is often more practical to create a local mirror branch from it first and use that branch instead. In this way, you can run multiple branch operations on it without any data transfers over the network. For example, you may want to inspect it using the `bzr missing`, `bzr log`, and `bzr diff` commands, or retry the merge using different algorithms.

Before committing the merge, make sure to test the improvements well, and look for any possible regressions. If there are any problems, you may want to ask for help from the author of the changes, or ask to fix bugs first and abort the merge for now.

Finally, push the branch to the common remote trunk to make it available to your team.

Merging feature branches in lock-step

Although multiple users may have write access to the same common trunk, simultaneous write operations are not possible. If two team members try to push their two different branches at the same time, only the first will succeed. After the first push succeeds, the second will fail, because the common trunk at the remote location, and its local mirror have diverged.

The cleanest solution is to create a new local branch based upon the updated common trunk, and repeat the merge.

Alternatively, if you prefer to reuse the local mirror and don't mind throwing away the merge, then you can use the `pull` command with the `--overwrite` flag. This will discard the merge, and fetch the new revisions from the common trunk, restoring it to its pristine mirror state.

Doing "code reviews"

Although the term "code review" is used in software development, its meaning is essentially the same as "peer review", and can be applied in other kind of projects as well. The idea is to review the work done by another team member in order to find and fix the mistakes that may have been overlooked.

When using code reviews, team members never merge their own branches into the common trunk, but ask another member to do it. When performing the merge, the reviewer should review the differences carefully before committing the merge. If there are problems in the branch or room for improvement, the reviewer can either fix the problems by committing new revisions, or abort the merge and ask the original author to fix the problems and ask for a review again.

Code reviews can greatly improve the quality of a project. It also helps to spread awareness of the new changes going into the project, ensuring that there are always at least two members who understand the meaning behind a change.

Summary

In this chapter, you have learned the fundamentals of working with others by sharing branches. We have covered the basic protocols supported by Bazaar, a few simple ways of configuring a remote server for sharing branches, and some practical techniques to work with remote branches efficiently and with confidence. Finally, we wrapped it all up in two example workflows suitable for small teams.

The workflows covered in this chapter are just examples demonstrating a few ways of managing the various branches when collaborating with others. With all the branch operations you've seen, you should be able to slice and dice branches as necessary to implement any workflow you will ever need.

The next chapter will show how Bazaar can work in a fully centralized mode, which is interesting not only because it is a widely used method in many projects today, but also because some of Bazaar's centralized features can have interesting uses even in distributed workflows.

5

Working with Bazaar in Centralized Mode

This chapter explains the principles of the centralized mode and how to work in this mode using Bazaar.

The centralized mode assumes one or more central branches, where collaborators share write access, and require the commit operations of all the users to be synchronized. This is the basic workflow enforced by centralized version control systems. This mode of operation is widely used today in many projects, and it is often preferred in corporate environments.

Although Bazaar is distributed in nature, it includes features to fully support the classic centralized mode. With Bazaar, you can switch in and out of the centralized mode at any time, and implement sophisticated workflows using both centralized and distributed elements.

The following topics will be covered in this chapter:

- The centralized mode
- Using Bazaar in the centralized mode
- Working with bound branches
- Working with multiple branches
- Setting up a central server
- Creating branches on the central server
- Practical use cases

The centralized mode

In the centralized mode, multiple users have write access to one or more branches on a central server. In addition, this mode requires that all commit operations be applied to the central branches directly. This is in contrast with the default behavior of Bazaar, where all commits are local only, and thus private by default.

In order to prevent multiple users from overwriting each other's changes, commits must be synchronized and performed in lock-step—if two collaborators try to commit at the same time, only the first commit will succeed. The second collaborator has to synchronize first with the central server, merging in the changes done by others, and try to commit again. In short, a commit operation can only succeed if the server and the user are on the same revision right before the commit.

First, we will learn about the core operations, advantages, and disadvantages of the centralized mode in a general context. In the next section, we will learn in detail how the centralized mode works in Bazaar.

Core operations

The core operations in centralized mode are checkout, update, and commit:

- **Checkout**: This operation creates a working tree by downloading the project's files from a central server. This is similar to the branch operation in Bazaar.
- **Update**: This operation updates the working tree to synchronize with the central server, downloading any changes committed to the server by others since the last update. This is similar to the pull operation in Bazaar.
- **Commit**: This operation records the pending changes in the working tree as a new revision on the central server. This is different from the commit operation we used in the earlier chapters, because in the centralized mode, the commit must be performed on the central server.

Bazaar supports all these core operations, and it provides additional operations to switch between centralized and decentralized modes, such as bind, unbind, and the notion of local commits, which we will explain later.

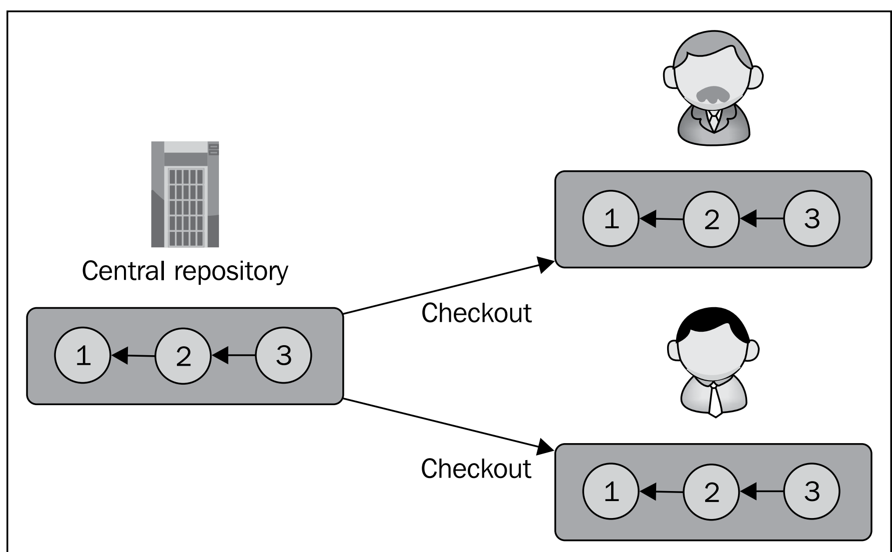
The centralized workflow

Since the centralized mode requires that all the commits be performed on the central server, it naturally enforces a centralized workflow. After getting the project's files using the checkout operation, the workflow is essentially a cycle of update and commit operations:

1. Do a "checkout" to get the project's files.
2. Work on the files and make some changes.
3. Before committing, update the project to get the changes committed by others in the meantime.
4. Commit the changes and return to step 2.

Checkout from the central branch

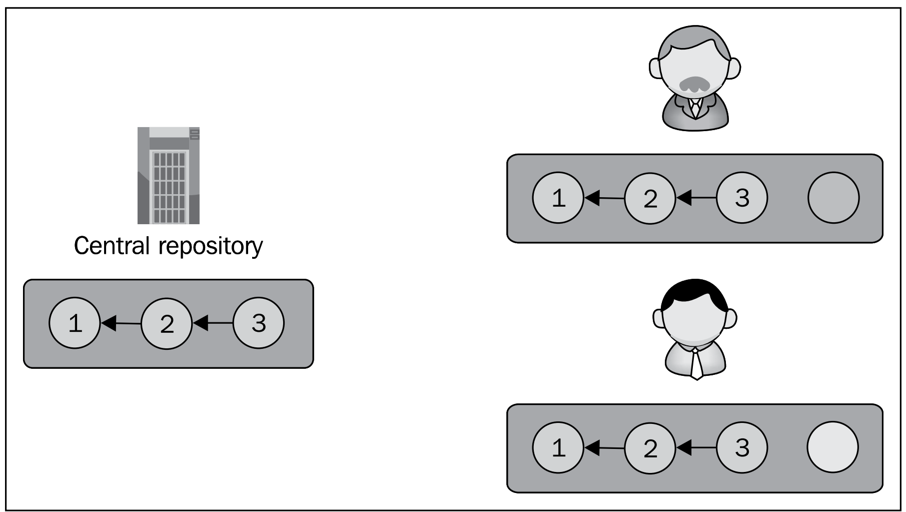
Given the central repository with its branches, the first step for a collaborator is to get the latest version of the project. Typically, you only need to do this once in the lifetime of the project. Later on, you can use the update operation to get the changes that were committed by the other collaborators on the server:



As a result of the checkout, collaborators have their own private copy of the project to work on.

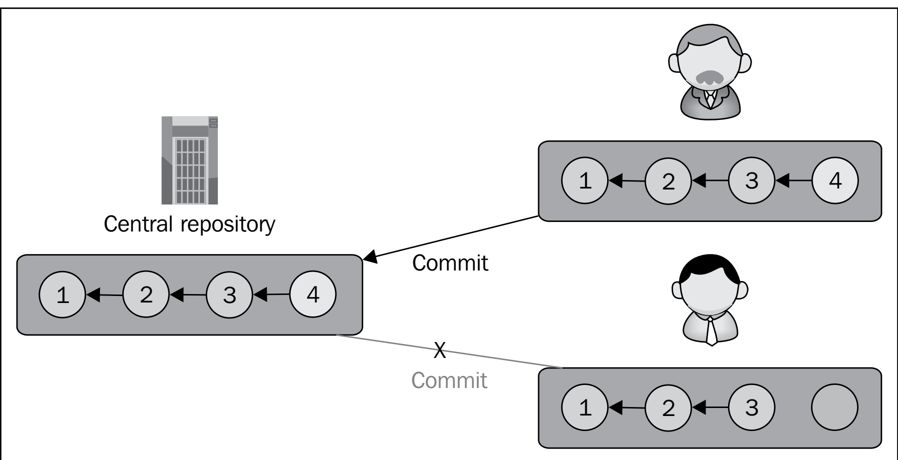
Making changes

Collaborators make changes independently in their own working trees, possibly working on copies of the same files simultaneously. Their environments are independent of each other and of the server too. Their changes are local and typically private until they commit them to the repository:



Committing changes

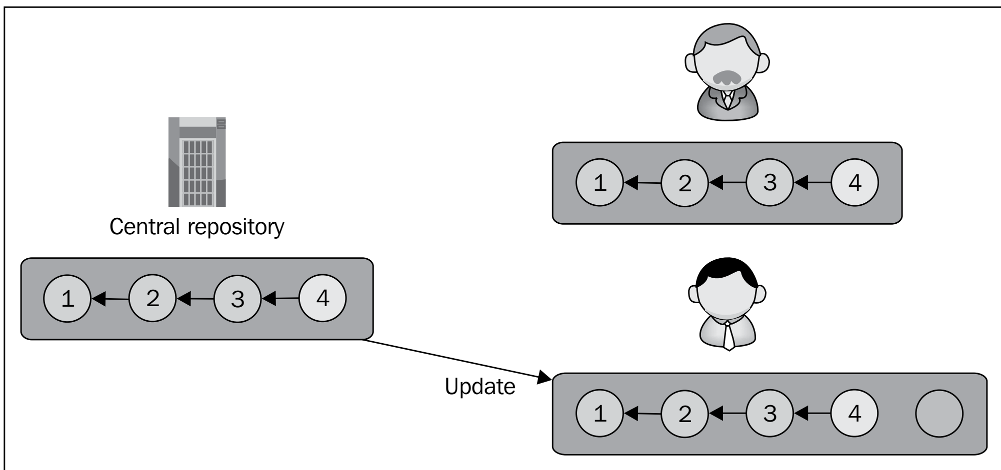
Commit operations are atomic—they cannot be interrupted or performed simultaneously in parallel. Therefore, collaborators can only commit new revisions one by one, not at the same time:



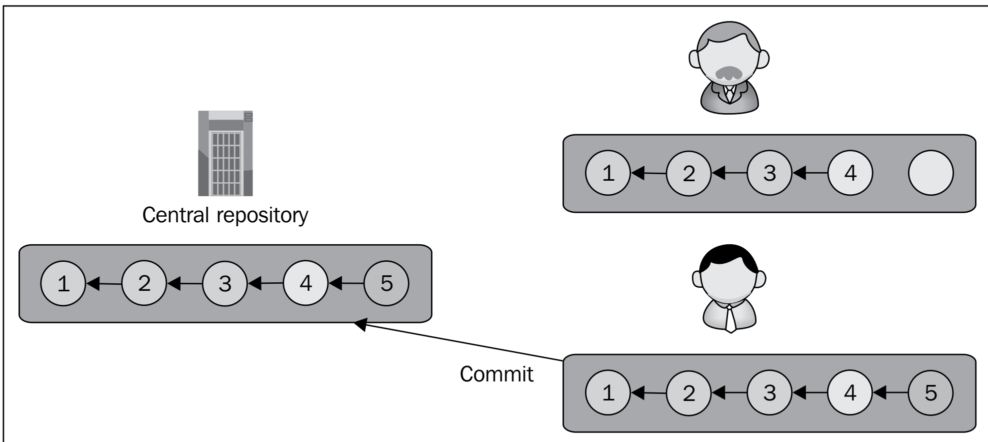
If two collaborators try to commit at the same time as in this example, only the first one will succeed. The second one will fail because his copy of the project will be out of date as compared to the server, where another revision has been added by the other collaborator. At this point, the second collaborator will have to update his working tree to bring it to the latest revision, downloading the revision added by the other user who succeeded to commit first.

Updating from the server

The update operation brings the working tree up-to-date by copying any revisions that have been added on the server since the last update or checkout. If there are uncommitted changes in the working tree, they will be merged on top of the incoming changes:



After the update, the local branch will be on the same revision as the server, and now the user may commit the pending changes:



Handling conflicts during update

When there are pending changes in the working tree, the update operation will try to rebase those changes on top of the incoming revisions. That is, the working tree is first synchronized with the server to be on the same revision, and after that the pending changes are applied on top of the updated working tree.

Similar to a merge operation, if the pending changes conflict with the incoming changes, the conflicts must be resolved manually. Since there is no systematic way to return to the same original pending state, the update operation can be dangerous in this situation. The more pending changes and the more time has elapsed since the last update or checkout, the greater the risk of conflicts.

Advantages

The centralized mode has several useful properties that are worth considering.

Easy to understand

The concept of a central server, where all the changes are integrated and the work of all collaborators is kept synchronized, is simple and easy to understand. In projects using the centralized mode, the central server is an explicit and unambiguous reference point.

Easy to synchronize efforts

Since all the commits of the collaborators are performed on the central server in lock-step, the independent local working trees cannot diverge too far from each other; it's as if they are always at most one revision away from the central branch. In this way, the centralized mode helps the collaborators to stay synchronized.

Widely used

The centralized mode has a long-standing history. It is widely used today in many projects, and it is often preferred in corporate environments.

Disadvantages

The centralized mode has several drawbacks that are important to keep in mind.

Single point of failure

Any central server is, by definition, a potential single point of failure. Since in the centralized mode all commits must go through the central server, if it crashes or becomes unavailable, it can slow down, hinder, or in the worst case completely block further collaboration.

Administrative overhead of access control

When multiple users have write access to a branch, it raises questions and issues about access control, server configuration, and maintenance:

- Who should have write access? An access control policy must be defined and maintained.
- How to implement write access of multiple users on the central branches? The central server must be configured appropriately to enforce the access control policy.
- Whenever a collaborator joins or leaves the project, the server configuration must be updated to accommodate changes in the team.
- Whenever the access policy changes, the server configuration must be updated accordingly.

The update operation is not safe

The centralized mode heavily relies on an inherently unsafe operation—updating the working tree from the server while it has pending changes. Since the pending changes are, by definition, not recorded anywhere, there is no systematic way to return to the original state after performing the update operation.

Unrelated changes interleaved in the revision history

When collaborators work on different topics in parallel, if they continuously commit their changes, then unrelated changes will be interleaved in the revision history. As a result, the revision history can become difficult to read, and if a feature needs to be rolled back later, the revisions that were a part of the feature can be difficult to find.

Using Bazaar in centralized mode

Bazaar fully supports the core operations of the centralized mode by using so-called bound branches. The checkout and update operations are implemented using dedicated commands in the context of bound branches. The commit operation works differently when used with bound branches, in order to enforce the requirements of the centralized mode.

In addition to the classic core operations of the centralized mode, Bazaar provides additional operations to easily turn the centralized mode on or off, which opens interesting new ways of combining centralized and decentralized elements in a workflow.

Bound branches

Bound branches are internally the same as regular branches; they differ only in a few configuration values—the `bound` flag is set to `true`, and `bound_location` is set to the URL of another branch. We will refer to the bound location as the **master branch**.

In most respects, a bound branch behaves just like any regular branch. However, operations that add revisions to a bound branch behave differently—all the revisions are first added in the master branch, and only if that succeeds, the operation is applied to the bound branch.

For example, the commit operation succeeds only if it can be applied to the master branch. Similarly, the push and pull operations on a bound branch will attempt to push and pull the missing revisions in the master branch first.

Since being bound to another branch is simply a matter of configuration, branches can be reconfigured at any time to be bound or unbound.

Creating a checkout

The checkout operation creates a bound branch with a working tree. This configuration is called a **checkout** in Bazaar. This is essentially the same as creating a regular branch and then binding it to the source branch it was created from. The term checkout is also used as a verb to indicate the act of creating a checkout from another branch.

Using the command line

Let's first create a shared repository to store our sample branches:

```
$ mkdir -p /sandbox
$ bzr init-repository /sandbox/central
Shared repository with trees (format: 2a)
Location:
  shared repository: /sandbox/central
$ cd /sandbox/central
```

You can check out from another branch by using the `bzr checkout` command and by specifying the URL of the source branch. Optionally, you can specify the target directory where you want to create the new checkout. For example:

```
$ bzr checkout http://bazaar.launchpad.net/~bzrbook/bzrbook-examples/
hello-start trunk
```

You can confirm that the branch configuration is a checkout by using the `bzr info` command:

```
$ bzr info trunk
Repository checkout (format: 2a)
Location:
  repository checkout root: trunk
        checkout of branch: http://bazaar.launchpad.net/~bzrbook/bzrbook-
examples/hello-start/
         shared repository: .
```

The first line of the output is the branch configuration, in this case a "Repository checkout", because we created the checkout inside a shared repository. Outside a shared repository, the configuration is called simply "Checkout". For example:

```
$ bzr checkout trunk /tmp/checkout-tmp
$ cd /tmp/checkout-tmp/
$ bzr info
Checkout (format: 2a)
Location:
       checkout root: .
  checkout of branch: /sandbox/central/trunk
```

In both the cases the `checkout of branch` line indicates the master branch that this one is bound to.

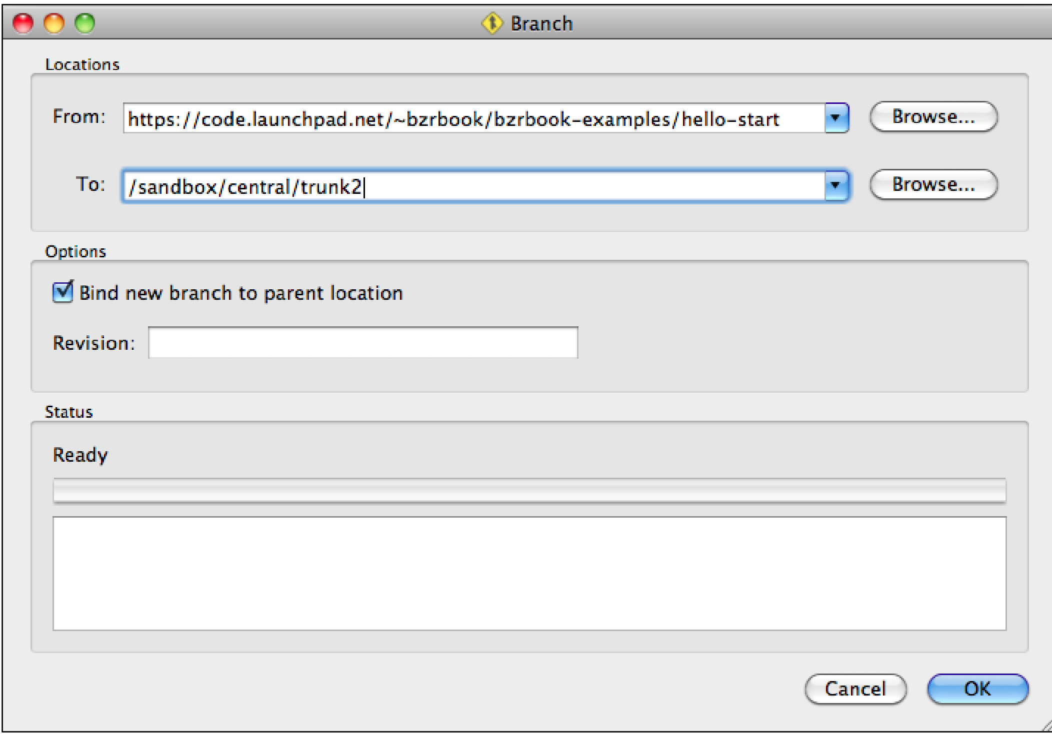
Using Bazaar Explorer

Performing a checkout using Bazaar Explorer can be a bit confusing, because the buttons and menu options labeled **Checkout...** use a special mode of the checkout operation called "lightweight checkouts". Lightweight checkouts are very different from branches; we will explain them in *Chapter 8*, *Using the Advanced Features of Bazaar*.

Use the Branch view to checkout from a branch:

- From the toolbar, click on the large **Start** button and select **Branch...**
- From the menu, select **Bazaar** | **Start** | **Initialize**

In the **From:** textbox, enter the URL of the source branch. In the **To:** textbox, you can either type the path to the directory where you want to create the checkout, or click on the **Browse** button and navigate to it. Make sure to select the **Bind new branch to parent location** box, in order to make the new branch bound to the source branch:

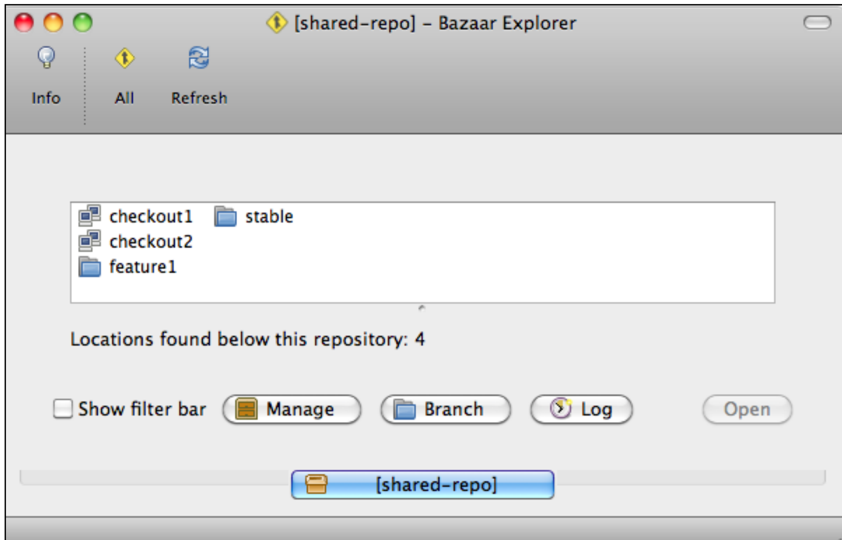


After you click on **OK**, the **Status** box will show the `bzr` command that was executed and its output. For example:

```
Run command: bzr branch https://code.launchpad.net/~bzrbook/bzrbook-
examples/hello-start /sandbox/central/trunk2 --bind --use-existing-dir
Branched 6 revisions.
New branch bound to https://code.launchpad.net/~bzrbook/bzrbook-examples/
hello-start
```

Click on **Close** to return to the status view, which shows the content of the working tree exactly in the same way as in the case of regular branches.

The **Status** view does not indicate whether the branch of the current working tree is bound or not. On the other hand, the repository view uses different icons to distinguish these configurations:



Bound branches are shown with a computer icon, and unbound branches are shown with a folder icon.

Updating a checkout

The purpose of the update operation is to bring a bound branch up-to-date with its master branch. If there are pending changes in the working tree, they will be reapplied after the branch is updated. If the incoming changes conflict with the pending changes in the working tree, the operation may result in conflicts.

As collaborators work independently in parallel, it is very common and normal that a bound branch is out of date due to the commits done by other collaborators. In such a state, the commit operation would fail, and the bound branch must be updated first before retrying to commit.

Similar to a pull operation, the update operation copies the missing revision data to the repository and updates the branch data to be the same as the master branch.

If there are pending changes in the working tree at the time of performing the update, they are first set aside and reapplied at the end. During this step conflicts may happen, the same way as during a merge operation.

Using the command line

You can bring a bound branch up-to-date with its master branch by using the `bzr update` command. To demonstrate this, let's first create another checkout based upon an older revision:

```
$ cd /sandbox/central
$ bzr checkout trunk -rlast:3 last-3
$ cd last-3
$ bzr missing --line ../trunk
You are missing 2 revisions:
6: Janos Gyerik 2013-03-03 updated readme
5: Janos Gyerik 2013-03-03 added python and bash impl
```

That is, our new checkout is two revisions behind the trunk. Let's bring it up to date:

```
$ bzr update
+N  hello.py
+N  hello.sh
 M  README.md
All changes applied successfully.
Updated to revision 6 of branch /sandbox/central/trunk
```

The missing revisions are added to the branch, and the necessary changes are applied to the working tree, resulting in identical branches:

```
$ bzr missing ../trunk
Branches are up to date.
```

Using Bazaar Explorer

To bring a checkout up-to-date with its master, you can either click on the large **Update** button in the toolbar, or navigate to **Bazaar | Collaborate | Update Working Tree....** in the menu.

The user interface does not take any parameters; the operation is applied immediately and its result is shown similar to the command-line interface.

Visiting an older revision

An interesting alternative use of the update operation is to reset the working tree to a past state, by specifying a revision by using the `-r` or `--revision` options. For example:

```
$ cd /sandbox/central/trunk
$ bzr update -r3
-D  .bzrignore
 M  README.md
-D  hello.py
-D  hello.sh
All changes applied successfully.
Updated to revision 3 of branch http://bazaar.launchpad.net/~bzrbook/
bzrbook-examples/hello-start
```

This may seem similar to using `bzr revert`, but in fact it is very different. The changes applied to the working tree will not be considered pending changes. Instead, the working tree is marked as out of date with its master, effectively preventing commit operations in this state:

```
$ bzr status
working tree is out of date, run 'bzr update'
```

Another difference from the `revert` command is that we cannot specify a subset of files; the `update` command is applied to the entire working tree.

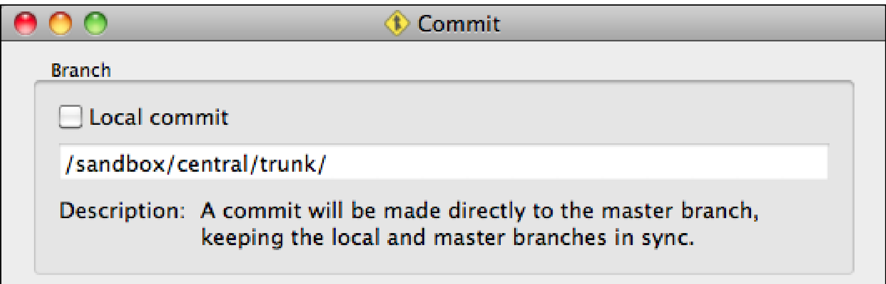
This operation works on unbound branches too. Since an unbound branch can be thought of as being its own master, the `update` command without a `revision` parameter simply restores it to its latest revision.

Committing a new revision

The commit operation works in the same way as it does with unbound branches, however, in keeping with the main principles of the centralized mode, Bazaar must ensure that the commit is performed in two branches—first in the master branch, followed by the bound branch.

The commit operation in the master branch succeeds only if it is at the same revision as the bound branch. Otherwise, the operation fails, and the bound branch must first be synchronized with its master branch using the update operation.

In Bazaar Explorer, the **Commit** view shows an additional explanation when committing in a bound branch, as a kind reminder that the operation will be performed on the master branch first, keeping the local and master branches in sync:



Practical tips when working in centralized mode

The centralized mode is simple and easy to work with in general, except for the update operation. The update operation can be problematic when there are too many pending changes in the working tree, and the central branch has evolved too far since the last time the bound branch was synchronized.

Fortunately, a few simple practices can greatly reduce or mitigate the potential conflicts that may arise during update operations:

- Always perform an update before starting to work on something new. That is, make sure to start a new development based on the latest version of the central branch.
- Break down bigger changes into smaller steps and commit them little by little. Don't let too many pending changes to accumulate locally; try to commit your work as soon as possible.
- In case of large scale changes and whenever it makes sense, use dedicated feature branches. You can work on feature branches locally or share them with others by pushing to the central server.

Working with bound branches

Bazaar provides additional operations using bound branches that go beyond the core principles of the centralized mode, such as:

- Unbinding from the master branch
- Binding to a branch
- Local commits

Essentially, these operations provide different ways to switch in and out of the centralized mode, which is extremely useful when a central branch becomes temporarily unavailable, or if you want to rearrange the branches in your workflow.

Unbinding from the master branch

Sometimes, you may want to commit changes even if the master branch is not accessible. For example, when the server hosting the master branch is experiencing network problems, or if you are in an environment with no network access such as in a coffee shop or in a train.

You can unbind from the master branch by using the `bzr unbind` command. To unbind a branch using Bazaar Explorer, you can either click on the large **Work** icon in the toolbar and select **Unbind Branch**, or using the menu **Bazaar** | **Work** | **Unbind Branch**.

Internally, this operation simply sets the `bound` configuration value to `false`. Since the branch is no longer considered bound, subsequent commit operations will be performed only locally, and the branch will behave as any other regular branch.

You can confirm that a branch was unbound from its master by using the **bzr info** command. For example:

```
$ cd /sandbox/central/
$ bzr checkout trunk mycheckout
$ cd mycheckout/
$ bzr info
Repository checkout (format: 2a)
Location:
  repository checkout root: .
        checkout of branch: /sandbox/central/trunk
         shared repository: /sandbox/central
$ bzr unbind
$ bzr info
```

```
Repository tree (format: 2a)
Location:
  shared repository: /sandbox/central
  repository branch: .
```

That is, the configuration has changed from `Repository checkout` to `Repository tree` and the `checkout of branch` line disappeared from the output.

Binding to a branch

Sometimes, you may want to bind a regular independent branch to another branch, for example to switch to using the centralized mode, or if you previously unbound from a branch and want to bind to it again.

You can bind to a branch by using the `bzr bind` command and specifying the URL of the branch. To bind a branch using Bazaar Explorer, you can either click on the large **Work** icon in the toolbar and select **Bind Branch...**, or use the menu **Bazaar | Work | Bind Branch...**. If you have previously used `unbind` in this branch, then you can omit the URL parameter on the command line, and in Bazaar Explorer the previous location is selected by default.

Internally, this operation simply updates the branch configuration—sets or updates the value of `bound_location` and sets the value of `bound` to `True`. Since the branch is now considered bound, all commit operations will be first applied to the master branch, but the working tree is left unchanged at this point.

Although you can bind any branch to any other branch, it only makes sense to bind to a related branch, typically a branch that is some revisions ahead of the current branch, so that a normal pull operation would bring the local branch up-to-date with its master branch.

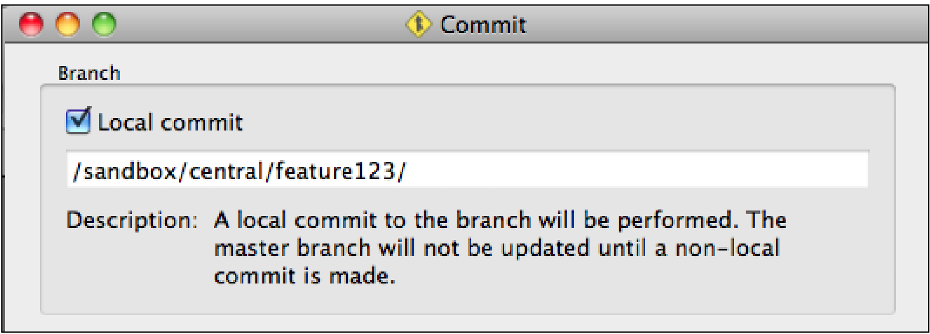
After binding to a branch, you should bring the local branch up-to-date with its master branch by using `bzr update`. Ideally, if the local branch is related to its new master and is just some revisions behind, then the update operation will simply bring it up-to-date by copying the revision data and the branch data of the master, leaving the working tree in a clean state, ready to work in the branch.

However, if the two branches have diverged from each other, then the update operation will perform a merge—first the working tree is updated to match the latest revision in the master branch, after that the revisions that do not exist in the master branch are merged in the same way as in a regular merge operation. This is an unusual use case, but nonetheless a valid operation. After all the changes are applied, you must sort out all conflicts, if any, and you may commit the merge. Since the branch is now a bound branch, the merge commit will be first applied in the master branch, and after that in the bound branch.

Using local commits

If you want to break out of the centralized mode only temporarily, an alternative to unbinding and rebinding later is using so-called local commits. When using local commits, you basically stay in centralized mode, but instead of trying to commit in the master branch, the commit operation is applied only in the local branch. This can be very useful when the master branch is temporarily unavailable but expected to be restored soon.

You can perform a local commit by using the `bzr commit` command with the `--local` flag, or in Bazaar Explorer by selecting the **Local commit** box in the **Commit** view:



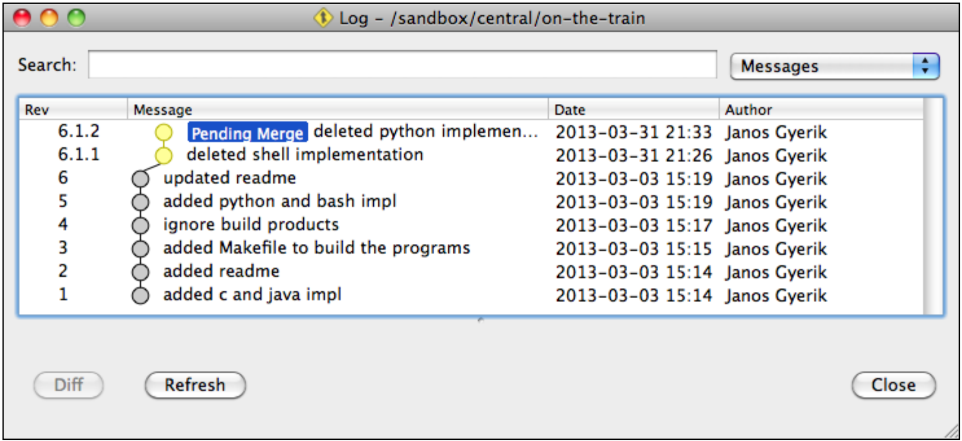
You can continue to perform as many local commits as needed until the master branch becomes available again.

As a result of local commits, the bound branch and the master branch go out of sync. If you try to perform a regular commit in such a state, Bazaar will raise an error and tell you to either continue committing locally, or perform an update and then commit.

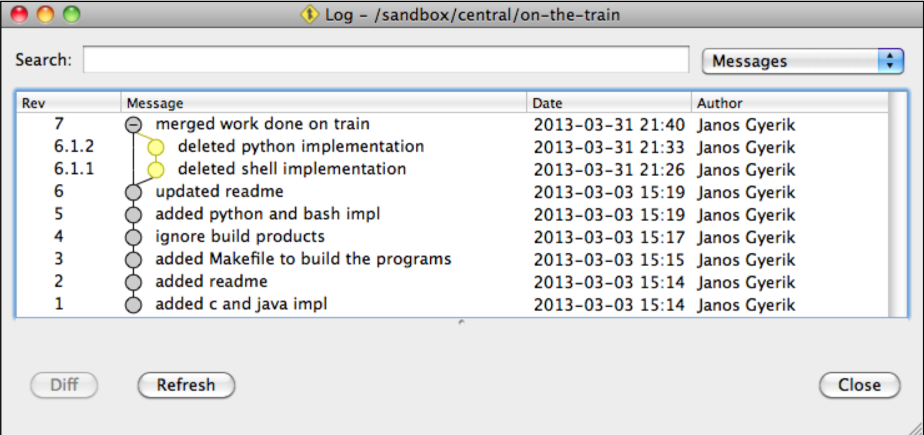
```
$ bzr commit -m 'removed readme'
bzr: ERROR: Bound branch BzrBranch7(file:///sandbox/central/on-the-train/) is out of date with master branch BzrBranch7(file:///sandbox/central/trunk/).
To commit to master branch, run update and then commit.
You can also pass --local to commit to continue working disconnected.
```

It may seem strange at first that we have to do an update even though in this case our local branch is clearly ahead of its master. However, the behavior is consistent with the rule – if a bound branch is not in sync with its master branch, you must always use the update operation to synchronize it.

As usual, the update operation will first restore the working tree to the same state as the latest revision in the master branch. After that, it will perform a merge from the tip of the local branch, applying the changes in the revisions that were committed locally. Finally, it will apply the pending changes that existed at the moment the update operation started. As a result, the working tree will be in a pending merge state, as you can confirm by using the `log` and `status` commands. For example:



After sorting out all conflicts, if any, you may commit the merge. The local commits will appear as if they had been on a branch and the branch has been merged. This makes perfect sense, as indeed this is exactly what happened:



If no new revisions were added in the master branch during your local commits, then a simple way to bring the master up-to-date is to do a `bzr push` operation instead of `bzr update`. It works because in this case the two branches have not diverged; the local branch is simply a few revisions ahead of its master. The push operation appends the missing revisions to the master branch, and the two branches become synchronized again, and you can continue to work and commit normally.

Working with multiple branches

Branch operations work consistently, regardless of whether you use the centralized mode or not.

Although the centralized mode permits multiple collaborators committing unrelated changes continuously in the central branch, it is better to work on new improvements in dedicated feature branches and merge them into the central branch only when they are ready. In this way, the revision history remains easy to read, and if a feature causes problems, then all the revisions involved in it can be reverted easily with one swift move.

Even in a centralized workflow, you are free to use as many local private branches as needed. You can slice and dice your local branches and when a feature is ready, you can merge them into the central branch, and all the intermediate revisions will be preserved in the history.

Team members can work on a feature branch together by sharing the branch on the central server. One of the team members can start working on the feature, and at some point push the branch on the server so that others can checkout from it and start contributing their work. After pushing the branch to the server, the original contributor can switch to the centralized mode using the `bind` command.

When working on a bound branch, keep in mind that in addition to the commit operation, the push and pull operations too will (at least least try to) impact its master branch.

Setting up a central server

In order to use Bazaar in the centralized mode, collaborators need to have write access to the branches on a central server. Here, we explain a few ways of configuring such servers.

Using an SSH server

An easy and secure way to provide write access to branches at a central location is by using an SSH server. In this setup, users authenticate via the SSH service running on the server, and their read and write access permissions to the branches are subject to regular filesystem permissions.

There are several ways of accessing Bazaar branches over SSH:

- Users access the server with their own SSH account
- Users access the branches with a shared restricted SSH account
- Users access the server with their own SSH account over SFTP

Using the smart server over SSH

If Bazaar is installed on the server, remote clients can benefit from the built-in smart server when accessing branches by using the `bzr+ssh://` protocol. In this mode, the `bzr serve` command is invoked on the server side to handle incoming Bazaar commands. This mode is called **smart server**, because remote clients receive assistance from the server, significantly speeding up Bazaar operations.

In addition to Bazaar being installed on the server, the `bzr` command must be in a directory included on the user's `PATH` variable. Otherwise, the absolute path of `bzr` must be specified at the client side, either in the `BZR_REMOTE_PATH` environment variable or in Bazaar's user configuration. For example, if `bzr` is installed in `/usr/local/bin/bzr`, then you can execute Bazaar commands on the remote location as follows:

```
$ export BZR_REMOTE_PATH=/usr/local/bin/bzr
$ bzr info bzr+ssh://user@example.com/repos/projectx
```

Alternatively, the remote path can be specified in the `locations.conf` file in your Bazaar configuration directory as follows:

```
[bzr+ssh://example.com/repos/projectx]
bzr_remote_path = /usr/local/bin/bzr
```

See `bzr help configuration` for more details.

Use the `bzr version` command to the find the location of the Bazaar configuration directory.

Using individual SSH accounts

This is the easiest way to access Bazaar repositories on a remote computer. Users with shell access to a computer can access Bazaar branches by using the `bzr+ssh://` protocol. For example:

```
$ bzr info bzr+ssh://user@example.com/repos/projectx
```

The `path` component in the URL must be the absolute path of the branch on the server; in this example, the branch is in `/repos/projectx`. If the branch is in the user's home directory, then the home directory part can be replaced with `~`; for example, instead of `/home/jack/repos/projectx`, you can use the more simple form `~/repos/projectx`:

```
$ bzr info bzr+ssh://user@example.com/~/repos/projectx
```

To refer to a Bazaar branch in another user's home directory, you can use the `~username` shortcut. For example:

```
$ bzr log bzr+ssh://user@example.com/~mike/repos/projectx
```

In order to let multiple users commit to the same branches, their user accounts must have write permission to the branch and repository files used by Bazaar. One way to do that is by adding the users to a dedicated group, and setting the ownership and access permissions appropriately. Let's call this group `bzrgroup`, and let's set up a shared repository at `/srv/repos/projectx` for members of the group, as follows:

```
$ bzr init-repository /srv/repos/projectx --no-trees
Shared repository (format: 2a)
Location:
  shared repository: /srv/repos/projectx
$ chgrp -R bzrgroup /src/repos/projectx
$ chmod g+s /src/repos/projectx
```

With this setup, the members of `bzrgroup` can create branches and commit to them. With appropriate permissions, other users can be permitted strictly the read-only access.

Using a shared restricted SSH account

Instead of creating individual SSH accounts for each collaborator, an interesting alternative is to use a shared SSH account with command restrictions.

This setup requires that collaborators use the SSH public key authentication when connecting to the server, and that appropriate access permissions to the branches be configured in the `~/.ssh/authorized_keys` file of the shared SSH account.

Let's suppose that:

- There is a shared repository on the server in `/srv/bzr/projectx`
- You want to give Jack and Mike write access to the shared repository
- The shared repository is owned by the user `bzruser`

To make this work, add the following two lines to the `~/.ssh/authorized_keys` file of `bzruser`:

```
command="bzr serve --inet --allow-writes --directory=/srv/bzr/
projectx",no-agent-forwarding,no-port-forwarding,no-pty,no-user-rc,
no-X11-forwarding PUBKEY_OF_JACK

command="bzr serve --inet --allow-writes --directory=/srv/bzr/proj
ectx",no-agent-forwarding,no-port-forwarding,no-pty,no-user-rc,no-X
11-forwarding PUBKEY_OF_MIKE
```

Replace `PUBKEY_OF_JACK` and `PUBKEY_OF_MIKE` with the SSH public key of Jack and Mike, respectively. For example, an SSH public key looks similar to the following:

```
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAo6+TOzByRt9EVUjpMBs5kRft9SSPam
I3cRlvaX4DuMbRqjtfkRTO4tik+MAWaFeIHyO5EsdFBGp+XVH9BMqehXdjAQga4Wa2o
GX/w7bn+O+gdIoJE2wzMlGV2eXcaW2PKdDIqQpUn0n+xX68vjRaCiZmqGXWhVej3cVi9
dtIwIQMrcIF4T+4wONic09UjPXZKbjL2GmkzsR6SMQJBomr4TUcRgyaR5ija9R8Azvs
SdNeDKkVwf83lva3jruwEMute3aZFulM5JqvjFIFqooAlSjWjdniF8ZdweeN1c2Q2Q
H+eCl48hY2drUsdZ+oQH+xp8x6llkZiDWFE/RZLa3Glw== Joe
```

The command parameter restricts the login shell to the `bzr serve` command. In this way, the users will not be able to do anything else on the server except run Bazaar commands. The `--directory` parameter further restricts Bazaar operations to the specified directory. To give only read-only access, simply drop the `--allow-writes` flag.

The other options on the line after `command` are to make the SSH sessions as restricted as possible, as a good measure of security.

When accessing branches in this setup, the `path` component in the branch URL must be relative to the directory specified in the authorization line. For example, the trunk in `/srv/bzr/projectx/trunk` can be accessed as follows:

```
$ bzr info bzr+ssh://bzruser@example.com/trunk
```

The drawback of this setup is that you can only have one configuration line per SSH key.

Using SFTP

If SFTP is enabled on the SSH server, you can access branches without installing Bazaar on the server by using the `sftp://` URL prefix instead of `bzr+ssh://`. For example:

```
$ bzr info sftp://user@example.com/home/mike/repos/projectx
```

This type of access is called "dumb server" mode, because in this case Bazaar is not used on the server side, and thus it cannot provide assistance to the client. In this setup, operations will be much less efficient compared to using the smart server.

Using bzr serve directly

You can use the Bazaar smart server directly to listen to incoming connections and serve the branch data.

Use the `bzr serve` command to start the smart server. By default, it listens on port `4155`, and serves branch data from the current working directory in read-only mode. It has several command-line parameters and flags to change the default behavior. For example:

- `--directory DIR`: This specifies the base directory to serve the branch data from, instead of the current working directory
- `--port PORT`: This specifies the port number to listen on, instead of the default `4155` port
- `--allow-writes`: This allows write operations instead of strictly read-only

Use the `-h` or `--help` flags to see the list of supported command-line parameters.

Branches served in this way can be accessed by URLs in the following format:

```
bzr://host/[path]
```

Here, `host` is the hostname of the server, and `path` is the relative path from the base directory of the server process.

For example, if the server is `example.com`, the smart server is running in the directory `/srv/bzr/repo`, and there is a Bazaar branch at the path `/srv/bzr/repo/projectx/feature-123`, then the branch can be accessed as follows:

```
$ bzr info bzr://example.com/projectx/feature-123
```

The advantage of this setup is that the smart server provides good performance. On the other hand, it completely lacks authentication.

Using bzr serve over inetd

On GNU/Linux and UNIX systems, you can configure `inetd` to start the `bzr serve` command automatically as needed, by adding a line in the `inetd.conf` file as follows:

```
4155  stream  TCP  nowait  bzruser  /usr/bin/bzr /usr/bin/bzr serve
--inet --directory=/srv/bzr/repo
```

Here:

- `4155` is the port number where the Bazaar server should listen for incoming connection.
- `bzruser` is the user account the `bzr serve` process will run as.
- `/usr/bin/bzr` is the absolute path of the `bzr` command.
- `/usr/bin/bzr serve --inet --directory=/srv/bzr/repo` is the complete command to execute when starting the server. The `--directory` parameter is used to specify the base directory of Bazaar branches.

Once configured, this setup works exactly in the same way as using `bzr serve` directly, with the same advantages and disadvantages.

Creating branches on the central server

Creating branches on a server works much in the same way as when creating branches locally. Here, we emphasize on some good practices for optimal performance.

The same way as when working with local branches, it is a good idea to create a shared repository per project to host multiple Bazaar branches. Even if you don't intend to use multiple branches at first, you might want to do that later, and it is easier to have a shared repository right from the start, than migrating an existing branch later.

Another important point is to configure the shared repository to not create working trees by default. Working trees are unnecessary on the server, because collaborators work in their local checkouts, and Bazaar may give warnings during branch operations if the central branch contains a working tree. In order to avoid confusion, it is better to completely omit working trees on the server.

Creating a shared repository without working trees

Similar to when working with local branches, using a shared repository on the server is a good way to save disk space. In addition, when pushing a new branch to the server that shares revisions with an existing branch, the shared revisions don't need to be copied, thus the push operation will be faster.

When creating the shared repository, make sure to use the `--no-trees` flag, so that new branches will be created without trees by default. Although, most probably, you will create new branches using push operations, and most protocols don't support creating a working tree when used with push, nonetheless it is a good precaution to set up a shared repository in this way right from the start.

Reconfiguring a shared repository to not use working trees

You can use the `bzr info` command to check whether a shared repository is configured with or without working trees. For example:

```
$ bzr info bzr+ssh://user@example.com/tmp/repo/
Shared repository with trees (format: unnamed)
Location:
  shared repository: bzr+ssh://user@example.com/tmp/repo/
```

If the first line of the output says `Shared repository with trees` instead of simply `Shared repository`, then you should log in to the server and reconfigure it by using the `bzr reconfigure` command with the `--with-no-trees` flag. For example:

```
$ cd /tmp/repo
$ bzr reconfigure --with-no-trees
$ bzr info
Shared repository (format: 2a)
Location:
  shared repository: .
```

Removing an existing working tree

If you already have branches on the central server with a working tree, then it is a good idea to remove them.

First, check the status of the working tree by using the `bzr status` command. If there are any pending changes, then commit or revert them.

To remove the working tree, use the `bzr reconfigure` command with the `--branch` flag.

Creating branches on the server without a working tree

Although you can use the `bzr init` and `bzr branch` commands directly on the server in the same way as you would do it locally, it would defeat the purpose of the centralized setup, and invite mistakes such as creating working trees by accident.

A common way to create new branches on the server is by using a push operation from your local branch. For example:

```
$ bzr push bzr+ssh://user@example.com/tmp/repo/branch1
Created new branch.
```

After pushing a branch, if you would like to work on it in the centralized mode, then you can bind to the remote branch by using the `:push` location alias:

```
$ bzr bind :push
```

Practical use cases

The key feature of the centralized mode is that it automatically keeps bound branches synchronized with their master branch. This opens interesting possibilities that can be useful in many situations, regardless of the workflow or the size of a team. To give you some idea here, we briefly introduce a few example use cases.

Working on branches using multiple computers

If you use multiple computers to work on a project, for example, a desktop and a laptop, or computers at different locations, then you probably need a way to synchronize your work done at physically different locations.

Although you can synchronize branches between the two locations by using mirror operations such as `bzr push` and `bzr pull`, they are not automatic, and thus you may easily find yourself in a situation that you cannot access some changes you did on another computer, because you forgot to run `bzr push` before you switched off the machine, for example.

Using the centralized mode can help here, because the synchronization between two branches is automatic, as it takes place at the time of each commit. You can start using the centralized mode by converting the branch you used to push to into a master branch, and binding to it with your other branches.

Let's say you have two computers, `computerA` and `computerB`, they both can access a branch at some location `branchX`, and you work on the branch sometimes by using `computerA`, and at other times by using `computerB`. (Whether `branchX` is hosted on `computerA` or `computerB` or a third computer doesn't matter, the example will still hold true.)

You can keep your work environments synchronized by using the `bzr push` and `bzr pull` operations, by adopting the following workflow on both the computers when working on branches you want to share:

1. Pull from `branchX`.
2. Work, make changes, and commit.
3. Push to `branchX`.

This can be tedious and error-prone; for example, if you forget to push your changes on one computer, then you might not be able to access those changes after switching to the other computer, as it may have been powered down, or be inaccessible directly over the network.

Using the centralized mode would simplify the workflow to only two steps:

1. Update from `branchX`.
2. Work, make changes, and commit.

Not only there is one less step to do, but since in this case `branchX` is automatically updated at every commit, the possibility of forgetting to run `bzr push` is completely eliminated.

You can convert your existing setup to using centralized mode simply by binding to `branchX` on both the computers, and then using the `update` command to synchronize. Assuming that both branches have no pending changes and both have been pushed to `branchX` as their last operation, you can convert them by using the following commands:

On `computerA`:

```
$ bzr pull
$ bzr bind :push
```

On `computerB`:

```
$ bzr bind :push
$ bzr update
```

After this, you can start using `branchX` in the centralized mode, as a cycle of the `bzr update` and `bzr commit` operations.

Synchronizing backup branches

An easy way to back up a branch is by pushing it to another location. For example:

```
$ bzr push BACKUP_URL
```

`BACKUP_URL` can be a path on an external disk, a path on a network share or network filesystem, or any remote URL.

However, the push operation is not automatic; it must be executed manually every time you want to update the backup.

Another way is to bind the branch to the backup location, effectively using it in the centralized mode. In this case, all commits in the bound branch will be automatically applied to its master branch too, keeping the backup up-to-date at all times.

You can convert the branch to this setup, simply by binding to the `push` location:

```
$ bzr bind :push
```

Since this practically means switching to the centralized mode, it is important to have fast access to `BACKUP_URL`, otherwise the delay at every commit might be annoying.

If you need to break out of the centralized mode, for example when the `BACKUP_URL` is temporarily unavailable for some reason, then simply run `bzr unbind`. And after `BACKUP_URL` becomes available again, you can bring the remote branch up-to-date with `bzr push`, and re-bind to it by using `bzr bind` without additional parameters to return to the centralized mode.

Summary

In this chapter, we explained the core principles of the centralized mode with its advantages and disadvantages. Bazaar fully supports the centralized mode by using bound branches, and we have demonstrated, with examples, how you can switch in and out of this mode at any time. We have covered a few simple ways of setting up a central server, where team members can have shared write access to branches, and a few practical use cases.

The centralized mode in Bazaar is very flexible. It can be used for more than just to imitate the workflow of centralized version control systems. Essentially, it provides automatic synchronization of two branches, which can be practical in many situations, even as a part of more sophisticated distributed workflows.

The next chapter will explain common distributed workflows and how to implement them using Bazaar. Distributed workflows are the most scalable, and thus suitable for projects of any size.

6

Working with Bazaar in Distributed Mode

This chapter explains the common distributed workflows and how to implement them using Bazaar. Distributed workflows are suitable for projects of any size, and these are the only workflows that are scalable enough to use in very large projects.

Distributed workflows are essentially about organizing branches in a certain way, and naturally involve a lot of branch operations. If you have a good understanding of the various branch operations, especially the techniques covered in the previous chapters, then there should be no big surprises for you here. The techniques in this chapter should serve as new practical examples of working with branches, further solidifying your knowledge.

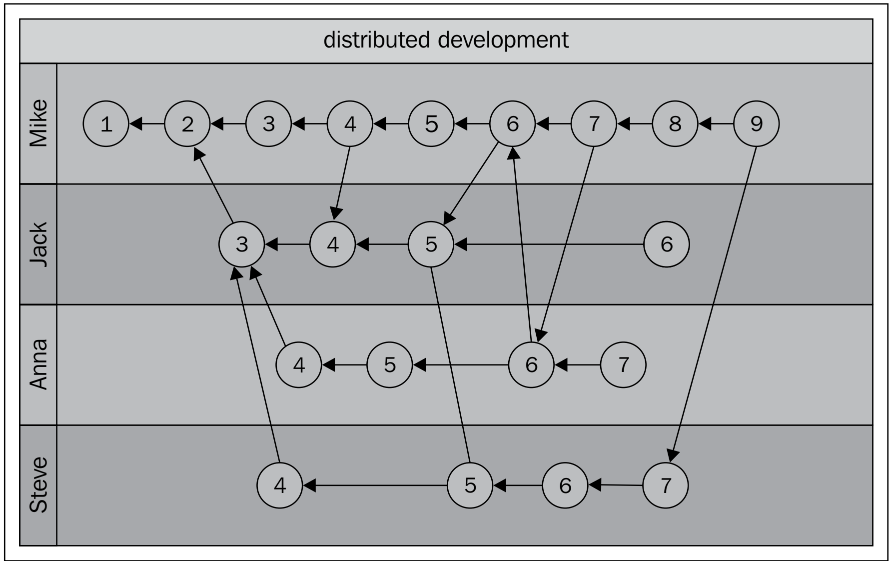
The following topics will be covered in this chapter:

- Using a human gatekeeper
- Using an automatic gatekeeper
- Using a shared mainline

The distributed mode in general

The essence of the distributed workflow is that collaborators don't have write access to a common central branch or to each other's branches. They only have write access to their own branches, and can propose those branches to others to merge from, and likewise, they can merge from the branches of others.

In their most primitive form, the branches of collaborators simply evolve in parallel, each collaborator working independently, occasionally merging from another collaborator; for example:



The arrows in this graph represent child-parent relationships, child revisions pointing to one or more parent revisions that they were derived from. All collaborators can only commit in their own branches. They cannot write to the branches of others, and can only share and propose their own branches for merge. That's the only way to get one's work into other users' branches.

The graph does not identify any of the branches as central, and indeed there is no central branch. Development takes place in a distributed manner, with no clear "official" branch. However, and especially, as the number of collaborators grows, the need for designating a branch as the mainline emerges. After all, without an "official" branch, it is impossible to know which branch to use as the base when starting a new development, for example, when a new member joins the team.

Designating a central branch does not require any special setup. It is only a convention, ideally explained in the project's documentation or website, not a hard rule. The only technical requirement of a mainline branch is simply that it should be accessible by the intended audience.

Ideally, the mainline branch should be a branch which has merged much of the work done in all the other branches, something stable, mature, and well-maintained. In this example, Mike's branch seems a good choice, as it has merged most of the revisions, though not all, from all the other branches. However, such a branch remains the mainline only as long as it is well-maintained, regularly merging the work that is being done in other branches.

The challenge in a distributed collaboration is bringing all the work going on in the various branches together into the mainline branch of the project. That said, there is nothing really difficult or complicated about this, and it can be accomplished easily by using the usual branching and merging operations. Doing so is not a technical issue, but more about good organization and communication between the members.

The graph of branches and revisions in the preceding example is the most primitive form of a distributed workflow. We can barely call it a workflow—it is a jumble of branches, with no apparent system or organization, and no mainline branch. Essentially, this is a peer-to-peer workflow—the team members are completely free and independent, they merge from each other whenever they want. This working style is not scalable if there are sufficiently many members in the project. The goal of distributed workflows is to organize the branches in such a way that it makes good, logical sense.

There are many ways of organizing branches in a distributed workflow; the most suitable method depends upon the project and its members. We will introduce some common techniques that you can use as they are, or as a baseline to build more suitable solutions depending upon your use case.

Before we get into the details of specific techniques, let's clarify some of the main principles of the distributed mode in general.

Collaborators work independently

First and foremost, all collaborators work independently; their workspaces are physically disconnected from the mainline and other collaborator branches. They can implement any workflow locally in their own workspaces and use as many branches as they want.

Collaborators can share their work with each other in an ad-hoc manner if they want, by publishing their local branches at some location where others have read access. This can be accomplished by pushing selected, or all branches to a central repository server, an SSH server, a web server, a shared folder on the local area network, or just about any other way that permits read access to the intended audience.

The ultimate purpose of all the work done in the independent branches is to merge back into the mainline development, or release branches, and thus become easily accessible as a part of the official version of the project.

The mainline branch is just a convention

Even in distributed workflows, typically there are one or more mainline branches that are commonly understood and accepted as the official version of the project. Having an official mainline branch makes good sense, as it makes the workflow easy to understand, and it can serve as the starting point for new development branches.

However, a branch being "the mainline" is just a convention. In terms of technical details and configuration, it is no different from any other regular Bazaar branch. The mainline is the mainline simply because the drivers of the project agree that it is. Any other branch can become the mainline, if necessary. For example, in an open source project if the current mainline branch becomes unmaintained or disputed, then another branch that is better maintained can emerge as the de-facto new mainline.

In short, distributed workflows have central branches too just like in a centralized workflow. They give great flexibility to collaborators, but the end result is the same—collaborator branches get merged into mainline branches, enriching and driving forward the project.

Collaborators write only to their own branches

In distributed workflows, collaborators have write access only to their own branches. This greatly simplifies access control—there is no need to configure and maintain access control. Only a single person has write access to any branch. Access control cannot get simpler than that.

Branches can be made visible to others for collaboration or sharing, but there is really no need to give write access to anybody else other than the branch owner.

In order to get their work into the mainline branches, collaborators propose their branches for merging to maintainers of the mainline. Alternatively, it is possible to create merge directives, which can be sent by e-mail and are very similar to, but much more powerful than, conventional patch files.

The distributed mode gives great flexibility

In distributed workflows, there are no technical restrictions with regard to the method of sharing work. A distributed version control tool doesn't get in the way—collaborators are free to use it in whatever way in their local workspaces, their work ultimately culminating in a branch to propose for merging into the project's mainline.

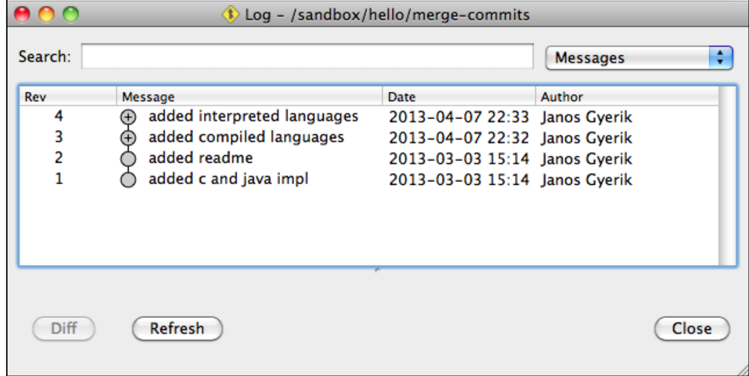
Whether a branch is fit or not to merge into the mainline is never a technical issue. If the work is good, there are many ways in which it can be merged into the mainline, with proper tracking of the revision history and attribution to the original authors.

Many technical issues inherent in centralized workflows simply don't exist in the distributed mode, such as the hassle of configuring access control, the dangers in update operations, or having a single point of failure. Having fewer technical rules and restrictions, the distributed workflow is more simple, easier to set up and maintain, and much more scalable.

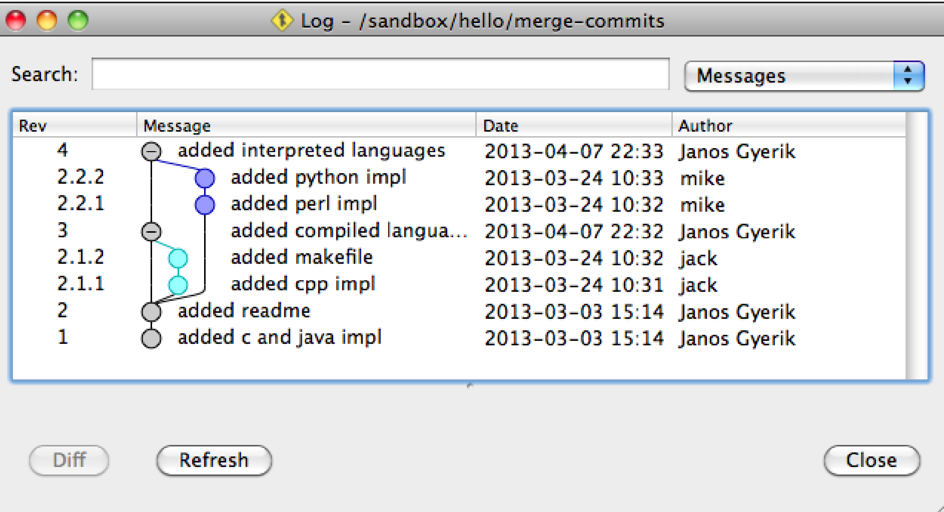
Encouraging feature branches

Since collaborators cannot write to each other's branches, the only way to get your work into the branch of another collaborator is by convincing him to merge from your branch. If the proposed branch is about a single feature, bugfix, or other specific improvement, merging it should be relatively easy. However, if the branch contains a mix of unrelated changes, then the merge proposal is likely to get rejected because it is unclear and confusing. As such, distributed workflows naturally encourage the use of feature branches.

Feature branches keep the history clean and well organized by grouping related changes together. In this way, you can read the merge commits of feature branches as the large steps in the evolution of the project, and you can always drill down to the individual commits to see the full details. For example:



The same history with merge commits expanded looks similar to the following screenshot:

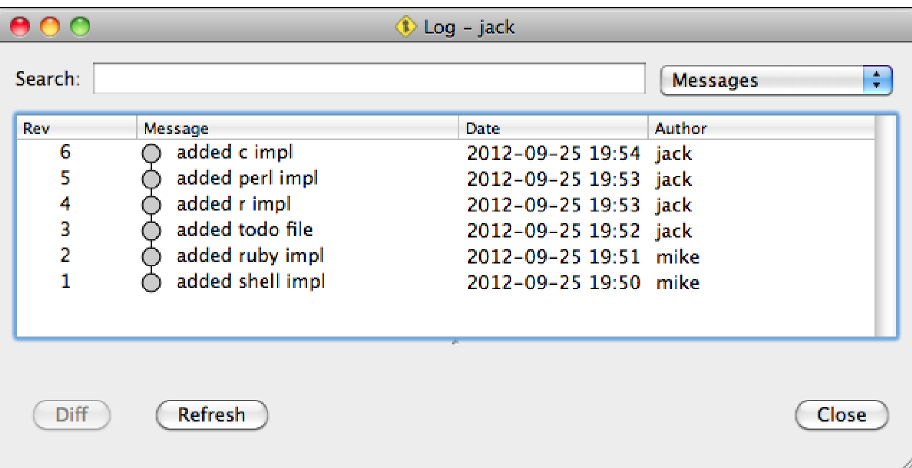


This also makes rolling back an entire feature trivially easy, by reverting the single commit that merged the feature branch.

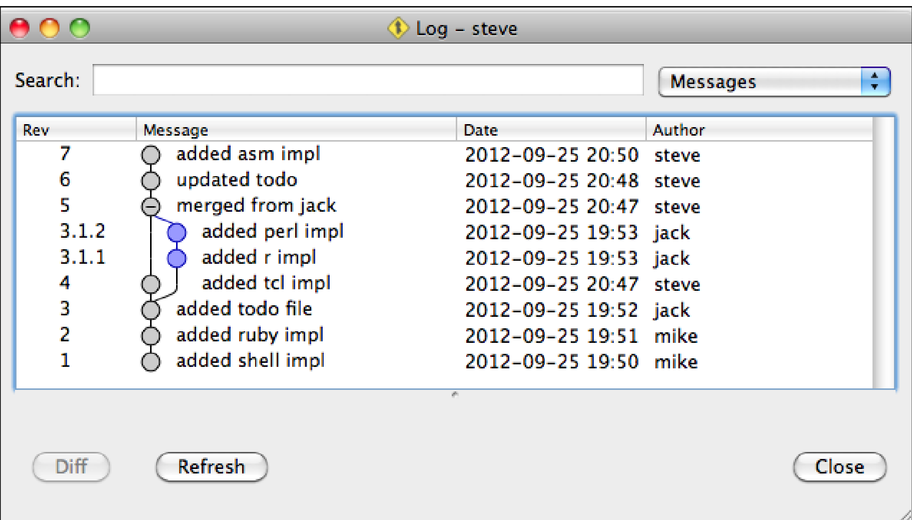
The revision history depends on the perspective

It is a somewhat minor, but sometimes important detail to remember that the graph of the revision history may depend on the perspective of each collaborator. When looking at the revision history of a branch, revisions added by the owner are called **mainline revisions**, revisions merged from other branches are called **merged revisions**. Mainline revisions are numbered with increasing integers, while merged revisions are numbered using a dot notation.

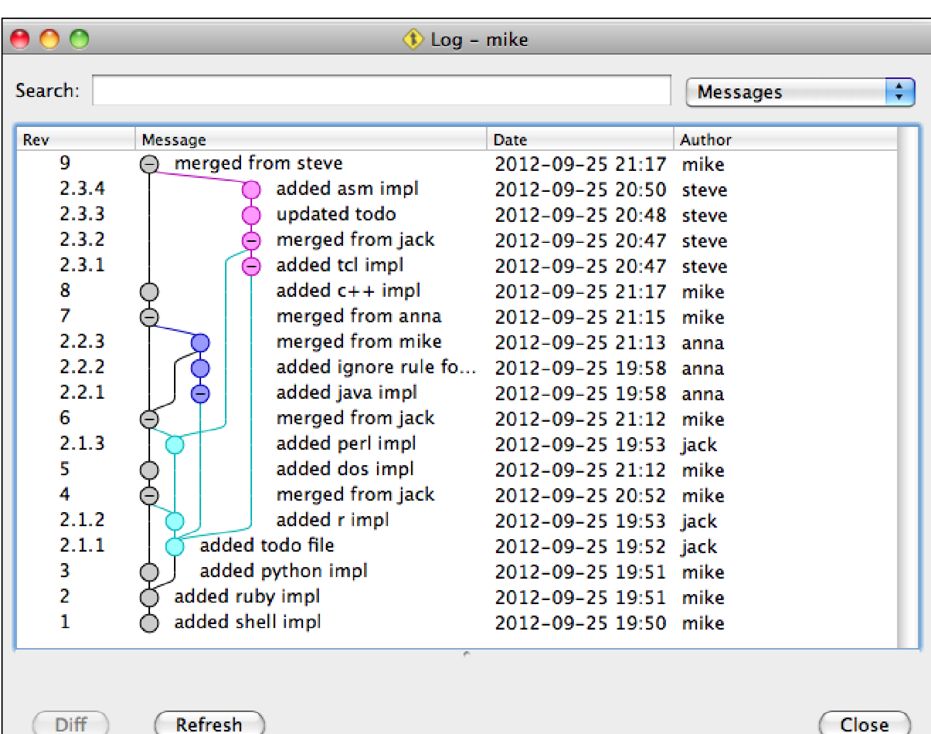
In the preceding example, Jack's view of the history is the most simple—he never merged from other branches. Thus his view of the history is basically a straight sequence of revisions following one another:



Steve has merged Jack's revision **4** and **5**. Therefore, in his view of the history these show up as merged revisions renamed as **3.1.1** and **3.1.2**, respectively:



Mike's perspective is even more complex, as he merged from all other branches:



If we create a new branch from Mike's at revision **2.3.4**, we get a perfect clone of Steve's branch, and therefore the same perspective as his. In this case, revisions **2.3.x** are renamed to **4**, **5**, **6**, and **7**, naturally, as they are mainline revisions in Steve's branch.

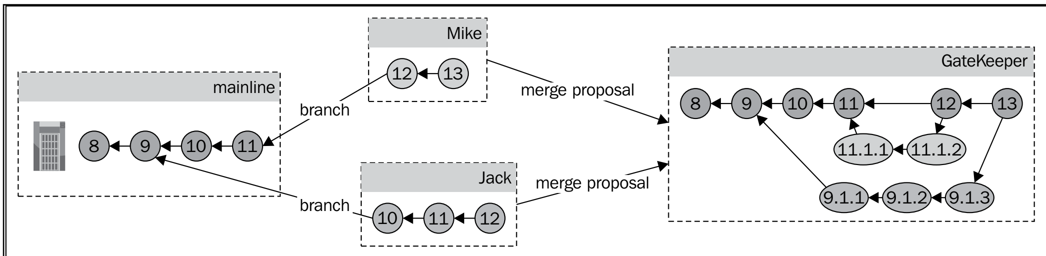
The human gatekeeper workflow

This is one of the most common distributed workflows. In this workflow, collaborators have read-only access to the mainline branch, and can propose their own branches for merging into the mainline. The maintainer of the mainline is the gatekeeper, who reviews merge proposals and either accepts and merges the branch into the mainline or rejects the proposal with comments.

If a branch was rejected, its author can fix the problems and commit them in the same branch, and propose it again for merge. This cycle can continue for as long as necessary, until finally the branch can be accepted and merged into the mainline.

Overview

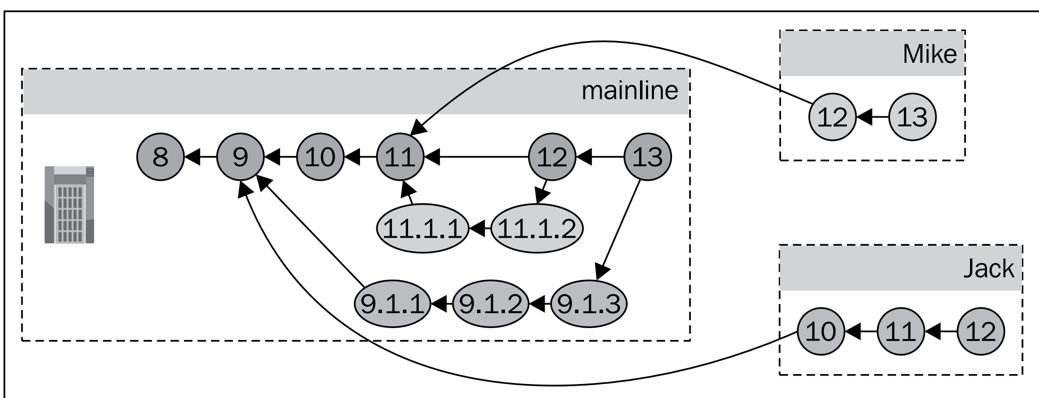
The general flow with two collaborators and a gatekeeper looks similar to the following:



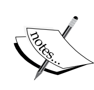
The important points to be noted are as follows:

- Collaborators do not have write access directly on the mainline branch
- Collaborators can propose their branches for merging to the gatekeeper
- It is up to the gatekeeper whether or not to accept or reject a merge proposal

In the same way that Jack and Mike branched from the mainline and worked on their local branch, the gatekeeper too does the same. In this example, the gatekeeper first merged and accepted Mike's branch, and then merged and accepted Jack's branch. At this point, the gatekeeper can push his branch to the mainline, resulting in the following:



At this point, since the mainline contains all the revisions of Mike and Jack, they can update their local branches by using `bzr pull` to make them exact mirrors of the mainline. Had they added new revisions after the time their branches were merged, `bzr pull` would not have worked, as the branches would have diverged.



> Ideally, the proposed branch should be a feature branch, with revisions that implement a single feature, bugfix, or specific improvement. To work on another feature, it is better to start afresh by creating a new branch from the current mainline rather than reusing the same one.

Setting guidelines to accept merge proposals

To reduce the turnaround of merge proposals and rejections, it is a good idea to keep a public list of the guidelines used when evaluating merge proposals. In this way, collaborators can know in advance what to watch out for, and avoid common pitfalls, increasing the chance that their merge proposal will be accepted, thereby making the review process more smooth and efficient. The guidelines can be general practices such as:

- The changes should not break anything
- Pass automated tests, such as unit tests or schema validation
- Conform to the general best practices of the relevant domain
- The changes should be about a single feature, bugfix, or some specific improvement
- The changes should be in line with the project strategy, not deviating from the main direction

Since the gatekeeper is a human, inevitably there may be some subjective criteria such as:

- Coding style (in software development projects)
- Writing style (in professional writing or translation)
- The changes should be "readable"; the gatekeeper may reject anything that is not clear to understand

The gatekeeper's job is easiest if the branch proposed for merge passes all the guidelines and automated tests. In that case, it can be simply accepted and merged into the mainline.

If the proposed branch does not meet some of the guidelines, it is best to reject the proposal with appropriate comments listing what to fix. The gatekeeper should keep rejecting a branch multiple times, if necessary, until it meets all the guidelines.

If the proposed branch does not meet all the guidelines but represents a significant improvement, then it might be tempting for the gatekeeper to work on the branch himself in order to make it pass the guidelines. In short, this way the improvement will get into the mainline faster, thanks to skipping the turnaround time between the gatekeeper and the collaborator. However, in the long term, this may very well end up slower and more costly for the gatekeeper. If the gatekeeper does not consistently reject branches that violate the guidelines, then the collaborator may never learn to play by the rules, and the gatekeeper will have to keep fixing the same mistakes over and over again.

The guidelines should be well understood by all the collaborators of the project to avoid frustration and unnecessary turnarounds. Collaborators who are lazy to read and understand all the guidelines will eventually get it after their merge proposals are rejected a few times.

The longer or more rigorous the list of criteria, the more difficult to join the project. This can be a very important point in open source software projects. In a new project, you probably don't want to impose too many rules at first, as that can discourage early contributors.

On the other hand, if there are not enough guidelines, it is likely to result either in a lot of rejected proposals, or a lot of extra work for the gatekeeper. The right balance depends upon the project and the team.

The role of the gatekeeper

At the minimum, the gatekeeper should enforce the common guidelines and best practices of the project, thereby ensuring continued high quality.

Merge proposals should not be accepted blindly, even if they have passed the common guidelines. It is crucial that the gatekeeper fully understands the changes introduced by a merge proposal, and its overall impact on the project. Therefore, naturally, the gatekeeper should be somebody with a firm grasp of the entire project and its future direction.

In addition to knowing the project through and through, the gatekeeper role typically involves a lot of interaction with the authors of merge proposals, for clarification or discussion on the new changes. Therefore, communication skills are also very important.

Creating a merge proposal

In order to propose a branch for merging, the collaborator must make the branch available (visible) to the gatekeeper in some way. There are several ways to do that:

- Using a Bazaar hosting site
- Sharing the branch URL with the gatekeeper
- Creating and sending a merge directive

Using a Bazaar hosting site

Ideally, projects should use a Bazaar hosting site such as Launchpad.net, where members of the project can have their own workspaces to share branches with the gatekeeper and other collaborators. Using such a site can greatly simplify the process of submitting and evaluating merge proposals.

Launchpad is a collaboration and hosting platform for software projects. The merge proposal process works as follows:

1. Upload the completed feature branch to your account on Launchpad using a push operation
2. Use the web interface to visit your branch and propose it for merging into another branch, along with a description and other options that may help the gatekeeper and other reviewers. The merge proposal triggers an e-mail notification to the gatekeeper to bring attention to the new branch ready for merging

A Bazaar hosting site, such as Launchpad, can be very useful as the central hub of a project, where collaborators can find the mainline branches, or push their own branches to propose for merging. Launchpad has many other useful features, which we will cover in the next chapter.

Sharing the branch URL with the gatekeeper

The gatekeeper can review and merge from any branch that is accessible by a protocol supported by Bazaar. For example, the collaborator can publish a branch with `bzr push` to a website, FTP server, remote filesystem, SSH server, or anywhere that is accessible by the gatekeeper.

After making a branch available, the collaborator should tell the URL of the branch to the gatekeeper, along with a brief summary of the changes.

For example, if you run the website `http://example.com/`, and the files of the website are served from the directory `/var/www/example.com/`, which you can access using SSH, then you can push your branch with the following command:

```
$ bzr push bzr+ssh://user@example.com/var/www/example.com/feat12
```

As a result, the branch will become visible at the URL `http://example.com/feat12`, and the gatekeeper can run Bazaar commands to inspect it and merge from it. For example:

```
$ cd /path/to/local/shared/repository
$ bzr branch http://example.com/feat12
$ bzr info feat12
$ bzr qlog feat12
$ cd mainline
$ bzr merge ../feat12
```

Sending a merge directive

If it is not possible to make a branch accessible to the gatekeeper via a URL, the best alternative is to generate a merge directive and send it by an e-mail.

A merge directive is like a "mini-branch" packaged into a single file, which can be applied to other branches by using `bzr merge` or `bzr pull`. A merge directive contains only the necessary revisions to merge from a source branch to a submit branch. By default, the source branch is the current branch, and the submit branch is either a previously saved submit branch or the parent branch.

To demonstrate the use of merge directives, let's fetch two sample branches into a shared repository:

```
$ cd /sandbox
$ bzr init-repo using-merge-directives
Shared repository with trees (format: 2a)
Location:
  shared repository: using-merge-directives
$ bzr branch lp:~bzrbook/bzrbook-examples/hello-start trunk
Branched 6 revisions.
$ bzr branch lp:~bzrbook/bzrbook-examples/hello-fix-c fix-c
Branched 8 revisions.
```

Now, we have two branches—the trunk, and a feature branch that fixes a bug. Imagine that you have fixed a bug in your local branch, but you have no way to give access to this branch to the gatekeeper. In such a situation, your next best option is to create a merge directive and e-mail it to the gatekeeper.

Creating a merge directive

You can create a merge directive by using the `bzr send` command and specifying the destination branch, where the merge should be applied. You must specify either an e-mail address with the `--mail-to` option or a filename with the `--output` or `-o` option. If you specify an e-mail address, Bazaar will open the default e-mail application, pre-filled with the content of the merge directive. Alternatively, you can save the merge directive to a file and e-mail it later.

For example, we can create a `merge` directive from the example `fix-c` branch to the trunk, as follows:

```
$ cd /sandbox/using-merge-directives/fix-c
$ bzr send --output -
Bundling 2 revisions.
# Bazaar merge directive format 2 (Bazaar 0.90)

# revision_id: janos@axiom-20130303203100-3uy33a4q96ux5u9c
# target_branch: ../trunk/
# testament_sha1: 1686e71d4453af6b4b086831179bf55faac7729b
# timestamp: 2013-04-04 06:20:56 +0200
#   examples/hello-start
# base_revision_id: janos@axiom-20130303141948-m5zhycy23bkvs2xv
#
# Begin patch
=== modified file 'hello.c'
--- hello.c     2013-03-03 14:14:35 +0000
+++ hello.c     2013-03-03 20:31:00 +0000
@@ -1,5 +1,5 @@
-#include "stdio.h"
+#include <stdio.h>

 int main() {
```

```
-    printf("Hello World!");
+    printf("Hello World!\n");
 }
# Begin bundle
```
IyBCYXphYXIgcmV2aXNpb24gYnVuZGxlIHY0CiMKQlpoOTFBWSZTWcYlJSIAApdfgEA
QeGP//1LQ
```
...
```

The merge directive starts with a header, with important parameters describing the mini-branch, such as the storage format used by the revisions, the latest revision ID, and the base revision ID.

By default, the merge directive includes an optional patch, which can be helpful especially when the changes are small, like in this example, so that the recipient of the merge directive can get a quick idea of the changes just by reading the e-mail. With larger changes, this might not be all that useful as it is easier to read large changes using Bazaar Explorer's **Diff** view. In this case, it may be better to completely omit the patch using the `--no-patch` flag.

When using the `--mail-to` option to e-mail the merge directive instead of saving it in a file, Bazaar will launch the e-mail client configured in the global `mail_client` setting. You can change this setting by using Bazaar Explorer, from the menu option **Setting | Configuration | User Configuration** or by launching `bzr qconfig`, or by editing the `bazaar.conf` file in your Bazaar configuration directory. The "default" value in this setting means Bazaar will use the preferred e-mail client configured in your system.

Merging from a merge directive

A merge directive can be used in the `bzr merge` and `bzr pull` operations as if it was a regular branch. To demonstrate this, let's create a merge directive from the `fix-c` branch to the trunk, and then try to merge it in the trunk:

```
$ cd /sandbox/using-merge-directives/fix-c
$ bzr send -o ../merge-directive.out ../trunk
 M  hello.c
All changes applied successfully.
$ bzr diff
=== modified file 'hello.c'
```

```
--- hello.c     2013-03-03 14:14:35 +0000
+++ hello.c     2013-04-08 05:04:26 +0000
@@ -1,5 +1,5 @@
-#include "stdio.h"
+#include <stdio.h>

 int main() {
-    printf("Hello World!");
+    printf("Hello World!\n");
 }
$ bzr status -v
modified:
  hello.c
pending merges:
  Janos Gyerik 2013-03-03 use more modern include-style
    Janos Gyerik 2013-03-03 c impl should add newline
```

The result is exactly the same as when merging from a real branch—changes are applied, and the revision history will be correctly preserved.

Merge directives without revision content

If the source branch is visible by a public URL, or if it has a public mirror, then it can be a good idea to omit the bundle from the merge directive in order to make it lighter, since in this case, the recipient can find the revisions in the public URL. For this to work, the public URL of the source branch must be specified on the command line or in the branch configuration file `.bzr/branch/branch.conf` with the `public_branch` setting. Use the `--no-bundle` flag to create a merge directive without a bundle. For example:

```
$ cd /sandbox/using-merge-directives/fix-c
$ bzr send ../trunk/ -o- --no-bundle --no-patch lp:~bzrbook/bzrbook-examples/hello-fix-c
# Bazaar merge directive format 2 (Bazaar 0.90)
# revision_id: janos@axiom-20130303203100-3uy33a4q96ux5u9c
# target_branch: ../trunk/
# testament_sha1: 1686e71d4453af6b4b086831179bf55faac7729b
# timestamp: 2013-04-08 06:45:31 +0200
```

```
# source_branch: lp:~bzrbook/bzrbook-examples/hello-fix-c
# base_revision_id: janos@axiom-20130303141948-m5zhycy23bkvs2xv
#
```

In this case, the merge directive file is much smaller, and instead of a bundle at the end, the public URL of the branch is included in the header as `source_branch`. When running this command, Bazaar verifies that the public URL is indeed a Bazaar branch and that it contains the latest revision of the current branch, otherwise the recipient won't be able to perform the merge.

Rejecting a merge proposal

The gatekeeper should carefully verify a merge proposal before accepting it, and put it through various tests. For example:

- Try to merge from the branch and see if there are any conflicts. This could be a warning sign, though it may not necessarily mean that the author did something wrong.
- Verify that the project is still working well after the merge.
- Run automated or manual non-regression tests.
- Look for inefficiencies that may cause problems and should be improved before merging.
- Verify that the general guidelines of the project are followed correctly.
- Needless to say, the changes should be in line with the long-term strategy of the project.

If there are any problems at any step, the gatekeeper may need assistance from the author to complete the merge. In order to ensure the continued high quality of the project, the gatekeeper must be wise, and should reject merge proposals that are not good enough.

When rejecting a merge proposal, the gatekeeper should explain to the author about the necessary improvements to make, in order to get the branch accepted. The author should continue working on the branch and commit more revisions that fix the issues that were pointed out by the gatekeeper. When ready, the author should propose the branch for merging again.

This cycle should continue as long as necessary, until the branch is approved by the gatekeeper. It is not fun for either party. Evaluating branches that have obvious problems that could have been avoided by following the guidelines is a waste of time for the gatekeeper, while getting rejected is frustrating for the branch author. It is important to remain patient, tolerant, and respectful during the process.

Although some problems can be fixed by the gatekeeper, it is better to let the branch contributor do it, in order to learn and stop making the same mistakes in future.

By rejecting merge proposals, the gatekeeper has the power to enforce the best practices documented in the project, even if some collaborators may be reluctant to do so.

Accepting a merge proposal

As always, when merging from a remote branch, it is a good idea to first fetch the remote branch, ideally into a shared repository. For example:

```
$ cd /path/to/shared/repo
$ bzr branch BRANCH_URL
```

In this way, you can run various commands to inspect the branch without unnecessarily paying the network overhead in each operation. For example:

```
$ cd the_branch
$ bzr info
$ bzr qlog
$ bzr missing ../mainline
$ bzr diff --old ../mainline
```

If you notice issues with the branch at this point, you can point them out to the branch author and ask to work on the branch some more. After the author updates the branch, you can do a `bzr pull` to bring your local mirror up-to-date.

If the branch passes the initial tests, try to merge it into your local mirror of the mainline, after making sure that the mirror is clean and up-to-date. For example:

```
$ cd ../mainline
$ bzr status
$ bzr pull
$ bzr merge ../the_branch
```

If the merge results in conflicts, which may be a warning sign, it does not necessarily mean that it is the fault of the branch author. Investigate, and if necessary, ask the branch author to help resolve the conflicts. You can also try to redo the merge by using a different algorithm with `bzr remerge`, or by completely aborting the merge with `bzr revert`.

After all conflicts are resolved, make sure to understand the meaning of the changes and verify carefully that the project is still working well, running automated or manual non-regression tests, and validating the common guidelines of the project.

If everything is in order, commit the merge with a short summary of the changes made in the branch, and push it to the central server to make it available to other team members:

```
$ bzr commit -m 'implemented feature X'
$ bzr push
```

> If doing `bzr push` for the first time, you may have to specify the parent location with `bzr push :parent`.

The gatekeeper must be wise and responsible, and therefore very careful when accepting changes in order to ensure the continued high quality of the project.

When working with a user-friendly Bazaar hosting site, such as Launchpad, the `bzr push` step should trigger an automatic e-mail to notify the author that the merge proposal was accepted and the branch was successfully merged.

Reusing a branch

Whenever possible, it is best to create a clean new branch from the mainline for each new feature, bugfix, or other specific improvement. When a feature is complete, propose the branch for merging and start a completely new branch from the latest version of the mainline in order to work on the next improvement.

However, sometimes this may not be practical, and it may be tempting to continue working in the same branch, even after it has already been merged into the mainline; for example, in situations similar to the following:

- There are many configuration files in the working tree that are required when working on the project, but cannot be added to version control because they are specific to the local working environment of each collaborator
- After the branch was proposed for merge and while waiting for the gatekeeper to accept or reject, you need to start working on the next feature that depends on the changes in the pending merge proposal
- The working tree is quite large, and thus keeping multiple working trees will be a waste of disk space

The cleanest way of reusing a branch is to wait until the merge proposal is accepted and merged into the mainline, then synchronize the local branch with the mainline using a pull operation. The pull operation will copy all the missing revisions and convert the branch to a perfect mirror of the mainline, and you may continue to work on the next improvement or bugfix.

Another way to re-use a branch is to merge from the mainline, but this leads to a messy history that's difficult to read, and the gatekeeper may also have issues with the criss-cross merges when the mainline and a collaborator branch are merged into each other repeatedly.

This effectively means working on the branch in lockstep with the gatekeeper:

1. You begin new work from a clean state, synchronized with the mainline.
2. When your improvement is completed, submit a merge request and wait for the gatekeeper to review the merge proposal and take action.
3. The gatekeeper may reject the proposal and ask you to improve the branch.
4. After the merge proposal is accepted and the branch is merged into the mainline, you can pull from the mainline to return the branch to the clean, synchronized state, and begin working on the next improvement.

This is a clean way for re-using a branch for multiple improvements, with the limitation that you have to work on improvements one by one, and refrain from committing new revisions while a merge proposal is still pending, practically working on the branch in lock-step with the gatekeeper.

There is a way for re-using a working tree to work on multiple branches by using lightweight checkouts and switching branches. This is an advanced setup, which will be explained in *Chapter 8*, *Using Advanced Features of Bazaar*.

Commander/Lieutenant model

As the project grows, it may become increasingly more difficult for the gatekeeper to oversee all the changes going into the different parts of the project. When the project reaches a point where the gatekeeper's job becomes impossible, the workflow can be scaled up by adding more gatekeepers, and splitting their responsibilities over different parts or modules of the project.

In very large projects, there can be several gatekeepers who oversee different parts of the project. This is often called the **Commander/Lieutenant** or **Dictator/Lieutenant** model. In this model, there are two levels of gatekeepers—**Lieutenants** review the merge proposals of the collaborators within their defined perimeters, but instead of merging collaborator branches into the mainline, they merge them into their own branches. The **Commander** works mostly with Lieutenants, reviewing their merge proposals and merging them into the mainline. In other words, the Commander is the gatekeeper of Lieutenants.

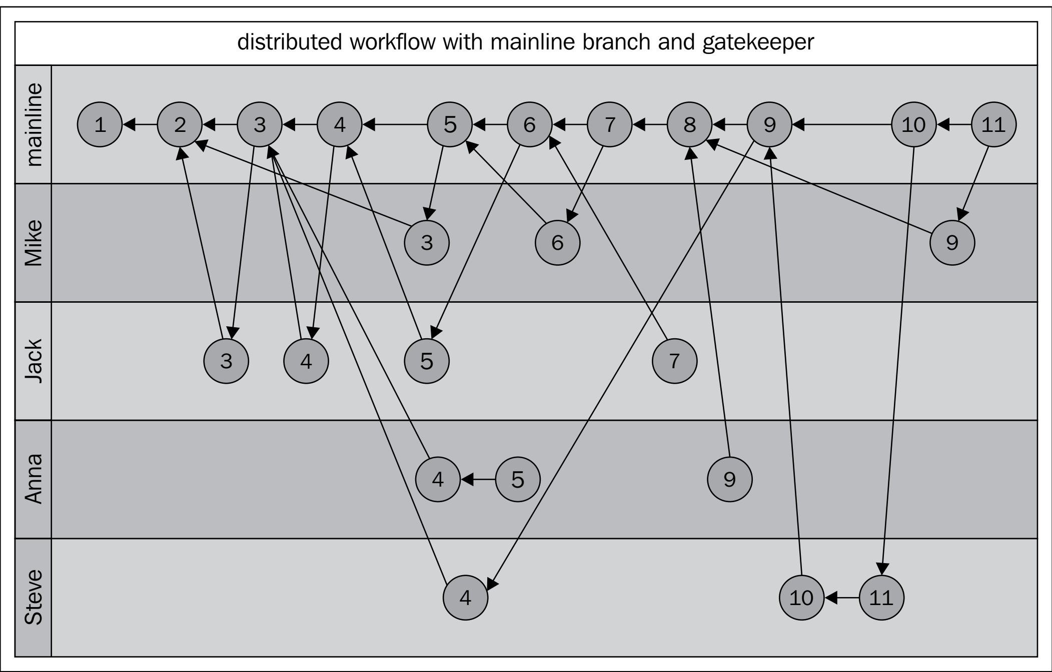
At the level of the Commander, it may be practically impossible to understand in depth all the individual changes going into the project. Instead, the Commander must focus on the higher-level logic of the proposed changes, and trust the Lieutenants' judgment on lower-level details.

Switching from the peer-to-peer workflow

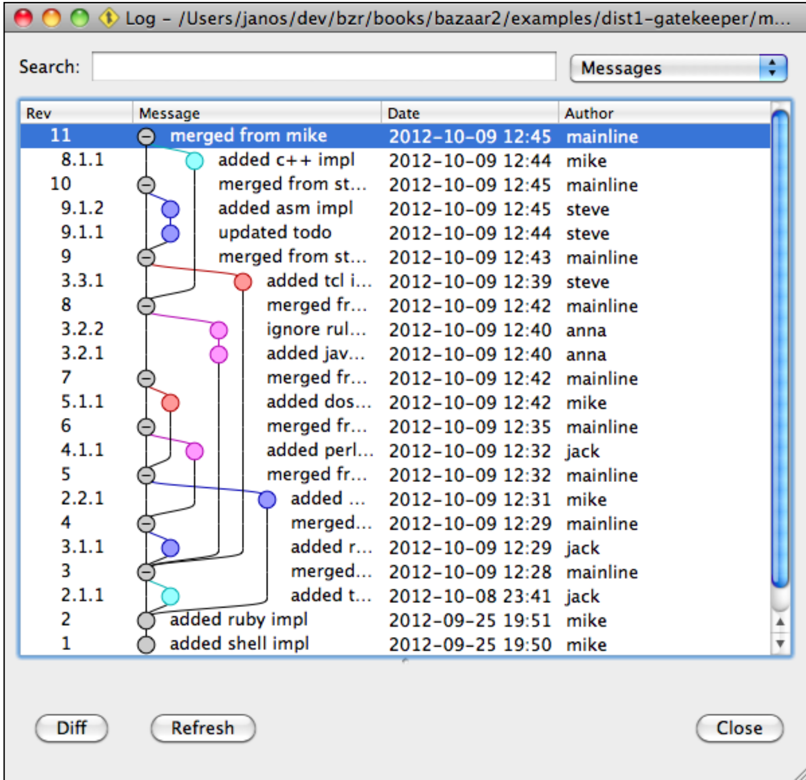
Switching from the peer-to-peer workflow to the human gatekeeper workflow requires the following changes in the working style:

- Dedicate a mainline branch that is only updated by the gatekeeper
- The new work should start from the mainline branch, not from the branch of another collaborator
- Collaborators should avoid merging from each other directly
- Collaborators should avoid re-using the same branch for multiple features, and always start the new work in a clean, new branch based upon the mainline branch
- The mainline branch should have mostly merge commits only, no other changes

If the peer-to-peer example at the beginning of the chapter had been using the human gatekeeper workflow, the revision graph would have become something similar to the following:



We can arrive at this graph by replacing the revisions that were merges from collaboratorB to collaboratorA with a merge from collaboratorB to the mainline, followed by a new branch from mainline to collaboratorA. Bazaar Explorer does a much better job at rendering such graphs:



This is not the best example, because there are too many branches here, with only a single revision. In reality, feature branches often have several revisions, and grouping them together gives a very useful, high-level overview of the larger steps in the evolution of the project.

The automatic gatekeeper workflow

The automatic gatekeeper workflow is a variant of the human gatekeeper workflow, except that the role of the gatekeeper is implemented by a program or script performing automated tasks instead of a human. This setup has some interesting advantages:

- It provides a mainline branch that is always up-to-date automatically, giving the illusion of a central branch, even though collaborators still work independently in a distributed manner
- It automates repetitive tasks, such as running non-regression tests and other validations of the common guidelines, automatically rejecting merge proposals that don't pass

Naturally, unlike a human gatekeeper, an automated process cannot evaluate the merge proposals from a strategic perspective. Any change that passes the automated test will be merged into the mainline even if it is not a good idea from a functional point of view.

Having an automated gatekeeper is probably better than not having one at all. Basically, it allows for an up-to-date mainline branch without human interaction, much like in a centralized workflow, but while still enjoying all the benefits of distributed version control.

Patch Queue Manager (PQM)

PQM is a software tool that implements the role of an automatic gatekeeper of Bazaar branches. It is implemented in Python and it uses Bazaar's libraries and API to perform the necessary branch operations.

PQM must be configured with write access to a dedicated branch, where it can push accepted merge proposals. PQM performs the following operations when processing a merge proposal:

1. Branch from the mainline.
2. Merge from the proposed branch.
3. Commit the merge.
4. Push to the mainline.

If any of these steps fail, the branch is rejected with an e-mail notification to the author.

Ideally, automated non-regression tests should be configured in a pre-commit hook of the branch, which is triggered when PQM tries to commit. The implementation of such a hook is specific to the project, and thus not a part of PQM itself.

PQM is merely the framework for automating the often repetitive tasks of accepting and rejecting merge proposals. It is the responsibility of the maintainers of the project to implement thorough non-regression tests triggered by a pre-commit hook.

The PQM project is hosted on Launchpad, and used extensively by Ubuntu projects (`https://launchpad.net/pqm`).

See the project website for more information and detailed setup instructions.

Revision history graph

The end result of the revision history graph is essentially the same as in the human gatekeeper workflow—the mainline has only merge commits, and all other branches should be feature branches. For a realistic example, get the source code of Bazaar itself and view its revision history:

```
$ bzr init-repo /tmp/bazaar
$ cd /tmp/bazaar
$ bzr branch lp:bzr  # will take some time!
$ bzr qlog bzr
```

The shared mainline workflow

In this workflow, the mainline branch is shared among a selected set of collaborators, possibly all of them. Collaborators do not commit directly to the mainline, but instead work on new features and bugfixes in local feature branches. When a feature branch is ready, either its author merges it into the mainline or asks another collaborator to perform a review and merge the branch.

There are two main ways of updating the mainline branch:

- Using an unbound branch with pull and push operations
- Using a bound branch with update and commit operations

Both the methods achieve the same result but work slightly differently. Essentially, these are just different working styles of updating remote branches; the preferred method may be a matter of taste.

In the examples demonstrating both the cases, we make the following assumptions:

- Local branches are in a shared repository located at `/sandbox/repo`
- The URL of the shared mainline branch is `MAINLINE_URL`
- The URL of the feature branch to merge is `FEATURE_URL`

The feature branch to merge may be a remote branch of another collaborator, or your own branch in the local shared repository.

Updating the mainline using push operations

The basic idea of updating the mainline branch using push operations is that you keep a pristine local mirror of the mainline branch, only for the purpose of pull, merge, and push operations.

Before you begin a merge operation, the local mirror must be up-to-date (that is, at the same revision as the mainline), so that after the merge is completed, you can push from it to the mainline.

Updating the mainline using a new local mirror

In a nutshell, the following are the steps to create a local mirror of the mainline branch, merge the feature branch into it, and push the result back into the mainline:

```
$ cd /sandbox/repo
$ bzr branch MAINLINE_URL mainline
$ bzr merge FEATURE_URL
$ bzr commit -m 'implemented feature X'
$ bzr push :parent
```

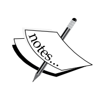
There are several things to keep in mind while performing these steps:

- The local mirror of the mainline branch will be created in `/sandbox/repo/mainline`, and you should not use it for anything else except for the purpose of merging other branches. Once created, it can be re-used to merge more branches in the future.
- If the feature branch is a remote branch, you probably want to create a local branch from it first by using `bzr branch`. Before performing the merge, you probably want to inspect the recent revision history of the branch by using `bzr log`, and maybe compare it with the mainline by using the `bzr missing` and `bzr diff` or `bzr qdiff` commands.

- After performing the merge, you may have to sort out conflicts, possibly contacting the authors of the conflicting changes. Before committing, you should run the various manual and automated non-regression tests of the project, and verify carefully that everything still works well, including the new evolutions introduced by the feature branch.
- When committing the merge, make sure to write an informative log message that summarizes well the changes that were made by the feature branch. Later, when you and other collaborators view the revision history of the project, the intermediary revisions of the feature branch will be hidden by default. Thus it is important to write a nice log message for merge commits.
- In the `bzr push` step, you can use the `:parent` shortcut to refer to the parent branch location; in this case, the mainline branch we branched from. After this, Bazaar will save this location as the push branch, so you don't need to specify it again.

At the end of the push operation, the mainline branch and its local mirror will be identical, and you can re-use the local mirror in the future to merge other branches.

While going through the preceding steps, if somebody else adds a change to the mainline, then the last step with the push operation will fail. In this case, you should start over by creating a clean new local mirror and perform the merge again.



In this case, instead of creating a clean new mirror again with `bzr branch`, you can re-use the existing branch by discarding all the changes and overwriting it with the remote mainline branch by using `bzr pull --overwrite`.

Re-using an existing local mirror

If you already have a local mirror of the mainline branch, and you kept it in a clean state without making any local changes, then you can re-use it to merge another branch with the following steps:

```
$ cd /sandbox/repo/mainline
$ bzr pull
$ bzr merge FEATURE_URL
$ bzr commit -m 'implemented feature X'
$ bzr push
```

The main difference compared to using a clean, new branch is that you can use `bzr pull` to bring the existing branch up-to-date with the remote mainline branch instead of creating a completely new branch by using `bzr branch`. This should be slightly faster, because in this case the working tree already exists, and it doesn't need to be completely recreated from scratch.

In terms of copying the revision data, this method might be slightly faster, but the difference is probably negligible, as most revisions should already exist in the shared repository.

If you have any pending changes or the branch has diverged from the mainline, then you can overwrite it from the mainline branch by using `bzr pull --overwrite`, discarding any local changes.

Another minor difference is that you don't need to specify the push location, as it is remembered from the last push operation.

Updating the mainline using a bound branch

The basic idea of updating the mainline branch using a bound branch is that you keep a pristine checkout of the mainline branch only for the purpose of merging branches.

Before you begin a merge operation, the checkout must be up-to-date (that is, at the same revision as the mainline), so that after the merge is completed, the commit operation will succeed in the mainline.

Updating the mainline using a new checkout

In a nutshell, these are the steps to create a checkout of the mainline branch, merge the feature branch into it, and commit the result into the mainline:

```
$ cd /sandbox/repo
$ bzr checkout MAINLINE_URL mainline
$ bzr merge FEATURE_URL
$ bzr commit -m 'implemented feature X'
```

There are several things to keep in mind while performing these steps:

- The local mirror of the mainline branch will be created in `/sandbox/repo/mainline`, and you should not use it for anything else except for the purpose of merging other branches. Once created, it can be re-used to merge more branches in the future.

- If the feature branch is a remote branch, you probably want to create a local branch from it first by using `bzr branch`. Before performing the merge, you probably want to inspect the recent revision history of the branch by using `bzr log`, and maybe compare it with the mainline by using the `bzr missing` and `bzr diff` or `bzr qdiff` commands.
- After performing the merge, you may have to sort out the conflicts, possibly by contacting the authors of the conflicting changes. Before committing, you should run the various manual and automated non-regression tests of the project, and verify carefully that everything still works well, including the new evolutions introduced by the feature branch.
- When committing the merge, make sure to write an informative log message that summarizes well the changes that were made by the feature branch. Later, when you and other collaborators view the revision history of the project, the intermediary revisions of the feature branch will be hidden by default. Thus it is important to write a nice log message for merge commits.

Keep in mind that in case of checkouts, the commit operation is applied directly to the master branch; in this case the mainline. After the commit, the two branches will be identical, and you can re-use the checkout in the future to merge other branches.

While going through the preceding steps, if somebody else adds a change to the mainline, then the commit operation will fail, and Bazaar will tell you to run `bzr update` to bring the checkout up-to-date. If the update operation is successful, you should repeat the non-regression tests and verify carefully that everything still works well. If everything is in order, you can proceed with the commit to complete the merge. Otherwise, if the update operation results in too many conflicts, then it may be better to abort the merge with `bzr revert` and start over.

Reusing an existing checkout

If you already have a checkout of the mainline branch, and you kept it in a clean state without making any local changes, then you can re-use it to merge another branch with the following steps:

```
$ cd /sandbox/repo/mainline
$ bzr update
$ bzr merge FEATURE_URL
$ bzr commit -m 'implemented feature X'
```

The main difference compared to using a clean new branch is that you can use `bzr update` to bring the existing branch up-to-date with the remote mainline branch instead of creating a completely new branch by using `bzr checkout`. This should be slightly faster, because in this case the working tree already exists, and it doesn't need to be completely recreated from scratch.

Choosing a distributed workflow

Each distributed workflow presented here has some advantages and disadvantages; the right one (most practical one) depends upon the project and the team.

The human gatekeeper workflow is the most restrictive, with strong control in the hands of the gatekeeper, who decides proactively which changes should go into the project and which should be rejected. By using a hierarchy of gatekeepers, this workflow can be infinitely scalable. This workflow is an excellent choice for medium to large projects. It is also well suited for open source projects, where a wide audience of contributors is welcome, but some measure of control is necessary to keep the mainline branch clean and on the right track.

The automatic gatekeeper workflow can be a bit difficult to set up, but once it is put in place, it gives the benefit of a mainline branch that is always up-to-date with little to no regular maintenance needed. The workflow eliminates the hassle of having to perform the often repetitive merge operations, at the expense of less control regarding what does and what doesn't go into the project. If you like openness, where all team members are equal and their work is trusted, then this workflow can be a good choice. It is also suitable as the Commander in the Commander/ Lieutenants model, since it is already largely based on trust in the Lieutenants, and it may not be practical for the Commander to review all the changes in the incoming branches from the Lieutenants.

The shared mainline workflow is somewhat similar to the automatic gatekeeper, in the sense that all the collaborators are equally trusted. However, in this case, the merge operations must be performed manually by the collaborators, which may possibly be error-prone. Another drawback is that the mainline must be configured with shared write access by multiple collaborators, which is an overhead compared to the classic model of distributed collaboration, where collaborators only have write access to their own branches.

These workflows are only examples of organizing distributed branches in a logical way. You can use these models as baselines, and with your, by now, solid understanding of the various branch operations, you can probably come up with new and different workflows of your own.

Summary

In this chapter, we explained the basics, and the various advantages of working in a distributed manner, such as great flexibility, and the natural tendency to use feature branches extensively.

We covered some of the common workflows used in distributed collaboration, and how to implement them using Bazaar. These workflows are essentially about organizing branches and combining the various branch operations in a certain way, suitable for projects of any scale. The workflows explained here can be used as examples to build on when designing your own custom workflows adapted to your projects.

The next chapter will explain how to integrate Bazaar into various collaborative development environments, such as Launchpad, various bug trackers, and repository browsing tools.

7
Integrating Bazaar in CDE

A version control system is but one of the many components in the set of necessary tools required to collaborate with others on a project. This chapter explains how to integrate Bazaar with various collaborative development environments.

The following topics will be covered in this chapter:

- What is a CDE?
- Working with Launchpad
- Integrating Bazaar with Redmine
- Integrating Bazaar with Trac
- Linking commits to bug trackers
- Web-based repository browsing with Loggerhead

What is a CDE?

A **Collaborative Development Environment** (**CDE**) is a collection of online collaboration tools used to manage the various aspects in the development of a project. You can expect the following features from a CDE:

- Version control system hosting with web-based repository browsing
- Bug/issue tracking system
- Task management system/to-do list
- Document management system
- Translation management system
- Wiki
- Mailing list
- Forum or bulletin board system for discussions

The essential elements depend upon the project and its collaborators involved. You might not need some of these features, or you might need all of them and more. Often, there is no single system covering all of these functionalities; a CDE may be composed of independent tools that can work well with each other.

Before deciding to use Bazaar in an existing infrastructure, it is an important question to ask whether it can or not integrate well and work together with the existing tools in the given environment. In this chapter, we will take a look at how Bazaar can be used with various CDE tools that implement at least some of the preceding functionalities, such as the following:

- **Launchpad**: This provides Bazaar hosting and web-based repository browsing, bug tracking, task management, translation management, and mailing lists
- **Redmine**: This provides web-based VCS browsing, bug tracking, task management, document management, and wiki
- **Trac**: This provides web-based VCS browsing, bug tracking, task management, document management, and wiki
- **Loggerhead**: This provides web-based Bazaar repository browsing

Working with Launchpad

Launchpad is an open source software hosting website for projects using Bazaar as the version control system. It has various additional components to facilitate collaboration, such as the following:

- **Repository browsing**: Viewing and browsing the files and directories of the project, including current and past revisions
- **Milestone and release management**: Defining milestones that can be targeted by bug fixes and new feature specifications
- **Specifications tracking**: Creating and editing new feature specifications together with the others
- **Bug tracking**: Creating bug reports, linking bugs to branches or commits, targeting bug fixes to milestone releases
- **Answers**: Tracking user support
- **Translation management**: Providing a collaborative interface to edit translations of the project files in multiple languages

Launchpad was designed to work with Bazaar, therefore no integration is necessary; Bazaar and Launchpad work well together out of the box.

Launchpad has an excellent online tour that walks you through the main features at `https://launchpad.net/+tour/index`.

In this section, we will focus mainly on Bazaar hosting, branch management, and bug-tracking features.

Many very large and famous projects use Launchpad, such as Ubuntu (including most of their subprojects like Unity), MySQL, Inkscape, Zope, OpenStack, and more. Naturally, Bazaar itself is also hosted on Launchpad.

Creating a Launchpad account

Having an account on Launchpad allows you to upload personal branches, host the codebase of open source projects, manage bugs, create teams and mailing lists, and so on.

In order to be able to perform write operations on branches hosted on Launchpad, such as commit and push, you must upload your SSH public keys to associate with your Launchpad account. Write operations by the `bzr` command or Bazaar Explorer can be authorized on Launchpad if your SSH public keys are correctly configured in your account details on Launchpad, and your Launchpad username is correctly set in your local Bazaar configuration.

Creating an account

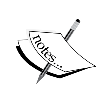
To create an account, visit Launchpad at `https://launchpad.net/`, and follow the instructions on the screen:

Launchpad is an OpenID provider—you will be able to use your Launchpad account as an OpenID login, with the URL:
`launchpad.net/~USERNAME`

Choosing a good and short username is important, because it will be part of the URLs of your personal branches, as follows:

`lp:~USERNAME/PROJECT/BRANCH`

When collaborating with others, you will often need to exchange branches and tell others about your branches. As such, you and your peers may have to type your username frequently, thus it is probably a good idea to choose something good and short.



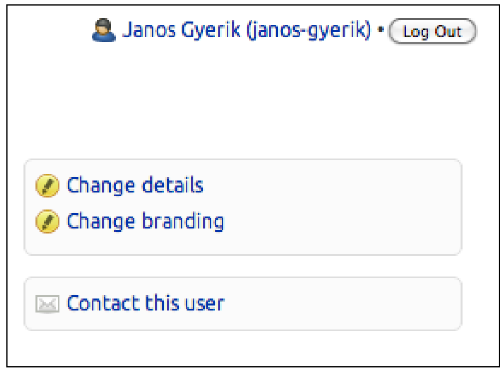
You can change your username later. However, if you use your Launchpad OpenID on other websites or services, keep in mind that changing your username will change your OpenID too, and thus you will lose access to those services. It's better to decide your username once and never change afterwards.

Configuring your Launchpad account

The homepage of your Launchpad account can be derived from your username, as follows:

```
https://launchpad.net/~USERNAME
```

At the top-right corner, there is a link named **Change details**, which takes you to a page where you can change your personal details and preferences:



Configuring SSH public keys

On your Launchpad home page, there are many settings that you can adjust by clicking on the pencil icons next to them. To edit your SSH public keys, click on the pencil icon next to the SSH keys setting.

It is normal if Launchpad asks you to re-enter your username and password when editing SSH keys. This is an additional security measure to protect your account. If a malicious user can access this screen when you leave your computer unattended for a few minutes, he/she could register his/her own SSH keys and overwrite your branches and any other branches to which your account has permissions.

On the **Change your SSH keys** page, you can add and remove SSH public keys that are allowed to write to Bazaar branches linked to your account, such as your own branches and branches of other teams who invited you to work on their project. To authorize an SSH key, copy the public key content and paste it in the large textbox.

In GNU/Linux, Mac OS X, and similar systems (UNIX, FreeBSD), your SSH public key is usually in the file `~/.ssh/id_rsa.pub`. If you don't have one already, the following help page explains very well how to create an SSH key:

`https://help.launchpad.net/YourAccount/CreatingAnSSHKeyPair`

In Windows, you can use `puttygen.exe` to generate keys, which is part of the PuTTY tool, and you can download it from the following URL:

`http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html`

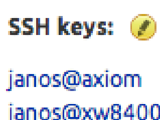
However, when generating keys using PuTTY, you will have to convert your public key for it to be in a single-line format, as follows:

```
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAxZT0202gdMDSnlbbs0dNXZd
il2aKyEgovnCUik7kMDeC2+aF46eHIrQGUhGfL3UmIdS61wFbvdG38c4yXmg
i87yJbb9SlVl2OmDDvU9TI/emWL71JT0viRxRVlYP9vlOF+r3fqaSW76PIalLE
uJJIol3Xwp/o9tPkc1Pqz40B7xJNGR8YVXBZci3WMX68yqk98Kqhp9KcLmzKArMjF
9gslDmXakFQnJ9VFH6kGCMjKRq60DQpwnyqLzSlaX41mIjo7ZezWQIKZBKz3adw7u/r
TqBMrctP2jxMHmLwg/slhOjL5jBZnWgFeEMCuxsexaaK8t+S7pAHe6kYp7AY06TJw==
 janos@axiom
```

The line has three parts, each separated by a single space:

- Type of the key, typically `ssh-rsa` or `ssh-dsa`
- The public key, a Base64-encoded long string
- A comment, often in the format `USERNAME@HOSTNAME`

The `comment` part of the key will be shown on your Launchpad home page, so that you can see at a glance the SSH keys you have authorized to access your account. For example, as follows:



For added security, whenever your SSH keys are edited, Launchpad sends a notification to your registered e-mail address.

Associating bzr with Launchpad

In order to write to branches on Launchpad with the push and commit operations using the `bzr` command or Bazaar Explorer, you must tell Bazaar your Launchpad username. You can do that either with the `bzr launchpad-login` command, or its shorter alias `bzr lp-login`. When used without parameters, the command shows the currently configured Launchpad username:

```
$ bzr lp-login
No Launchpad user ID configured.
```

By default, it is not configured, of course. Specify the Launchpad username to set and save it in your user configuration:

```
$ bzr lp-login bzrbuddy
```

The command verifies if the specified user exists on Launchpad, and if successful it updates the value of `launchpad_username` in your Bazaar configuration (`~/.bazaar/bazaar.conf`). Bazaar operations on Launchpad branches will check this configuration value to make the association between your actions and your Launchpad user.

To confirm that the Launchpad username was correctly set, you can run `bzr lp-login` again without parameters; it should simply print the username; for example:

```
$ bzr lp-login
bzrbuddy
```

Testing your setup

To confirm that your SSH keys and Launchpad user are correctly configured, you can run any operation on a Launchpad branch. For example, the `info` command on the official branch of the Bazaar project will provide the following output:

```
$ bzr info lp:bzr
Repository branch (format: 2a)
Location:
  shared repository: http://bazaar.launchpad.net/~bzr-pqm/bzr/bzr.dev/
  repository branch: http://bazaar.launchpad.net/~bzr-pqm/bzr/bzr.dev/

Related branches:
  parent branch: sftp://robertc@escudero/srv/www.bazaar-ng.org/rsync/bzr/
bzr.pqm/
```

If you have not informed Bazaar about your Launchpad username, read-only operations would still work, but you would get a warning similar to the following:

```
You have not informed bzr of your Launchpad ID, and you must do this to
write to Launchpad or access private data.  See "bzr help launchpad-
login".
```

If you have configured a username, Bazaar will try to authenticate using your SSH keys, even in the case of read-only operations, such as getting the info. Bazaar will look for your SSH key matching any of the public keys registered for your configured Launchpad username. If it cannot find a matching private key, the operation will fail. For example:

```
$ bzr lp-login bzrbook  # any user you don't own!
$ bzr info lp:bzr
Permission denied (publickey).
ConnectionReset reading response for 'BzrDir.open_2.1', retrying
Permission denied (publickey).
bzr: ERROR: Connection closed: Unexpected end of message. Please check
connectivity and permissions, and report a bug if problems persist.
```

Using private keys works in the same way as when authenticating to an SSH server. If you have a working setup to log in to an SSH server, you don't need to perform any additional configuration for Launchpad.

Hosting personal branches

Branches on Launchpad must belong to a user and a project. To upload branches that are not associated with any project, you can use a special project called `+junk`, which is designed exactly for this.

You can access the personal branches using URLs in the following format:

```
lp:~USERNAME/+junk/BRANCHNAME
```

Here, `USERNAME` is your Launchpad username and `BRANCHNAME` is any nickname you can pick for your branch when you create it.

Branches in the `+junk` project are commonly called **personal branches**, or sometimes **non-project branches**. Despite of the name as personal branches, these are not private; anybody can see your personal branches on your Launchpad home page, browse their content, and branch from them.

You can read more about personal branches on the following pages:

- https://help.launchpad.net/Code/PersonalBranches
- https://answers.launchpad.net/launchpad/+faq/226

In the next examples, we will use the username `bzrbuddy` when demonstrating Bazaar operations on Launchpad branches. Replace it with your own Launchpad username when working with your branches.

Uploading personal branches

You can upload a personal branch by using a push operation. As a test, let's create an empty branch and push it to Launchpad. For example:

```
$ bzr init /tmp/empty-sample
Created a standalone tree (format: 2a)
$ cd /tmp/empty-sample/
$ bzr push lp:~bzrbuddy/+junk/empty1
Created new branch.
```

Note that when you push a branch to Launchpad for the very first time, you will get a prompt to accept the host key of Launchpad as follows:

```
The authenticity of host 'bazaar.launchpad.net (91.189.95.84)' can't be established.
RSA key fingerprint is 9d:38:3a:63:b1:d5:6f:c4:44:67:53:49:2e:ee:fc:89.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'bazaar.launchpad.net' (RSA) to the list of known hosts.
```

This is the same as when connecting to an SSH server for the very first time. After you say yes, the host key will be saved in your known hosts registry, and you won't be prompted for this again.

To confirm that the branch was created, visit your Launchpad home page and click on the **Code** tab, or use a direct URL; for example, `https://code.launchpad.net/~bzrbuddy`.

This page lists all of your branches:

Above the list of branches there is the `bzr` command to use to push personal branches. In this example, it is `bzr push lp:~bzrbuddy/+junk/BRANCHNAME`.

Keep in mind that your personal branches are not private; anybody visiting your Launchpad home page can see them, browse their content, and branch from them.

Personal branches are a good way to store non-private branches temporarily, or to share with others, or just to play with Launchpad.

Using personal branches

Personal branches on Launchpad work in the same way as any other remote branches. Since these branches are public, anybody can perform read-only operations on them, such as `branch`, `checkout`, `merge`, `info`, `revno`, and `ls`. For example:

```
$ bzr checkout lp:~bzrbuddy/+junk/empty1 /tmp/empty1-checkout
$ bzr branch lp:~bzrbuddy/+junk/empty1 /tmp/empty1-branch
Branched 0 revisions.
$ bzr ls lp:~bzrbuddy/+junk/empty1
$ bzr revno lp:~bzrbuddy/+junk/empty1
0
$ bzr log lp:~bzrbuddy/+junk/not-so-empty1
```

Deleting branches

To delete branches, visit your Launchpad home page, click on the **Code** tab, then click on the branch you want to delete. On the detailed view of the branch, there is a toolbox at the right-hand side with the option to delete the branch:

When you click on **Delete branch**, Launchpad will ask you for a confirmation.

Hosting a project

Launchpad is designed to host entire projects with all their Bazaar branches. It is free for open source projects. There is an excellent documentation on hosting projects on Launchpad at `https://help.launchpad.net/Code/QuickStart`.

For the purposes of the book, we will cover only the absolute essentials, focusing on the points of integration with Bazaar.

Using the Sandbox site

While experimenting with the various features of Launchpad, it may be a good idea to use the Sandbox environment instead of the official site. There is a link to the Sandbox right at the front page of `https://launchpad.net/`, or you can visit it directly at `https://qastaging.launchpad.net/`.

The sandbox environment is a copy of the official site, with user accounts, projects and all other content copied over periodically. It is not safe to do any real work there because anything you enter will be erased or over-written at some point. However, it is perfect for experimenting without affecting your real users, teams, and projects.

Some of the examples use the official site, while others use the Sandbox. You can use this conversion table to convert between the URLs of the two sites:

Launchpad	**Sandbox**
`lp:~USERNAME`	`lp://qastaging/~USERNAME`
`lp:PROJECTNAME`	`lp://qastaging/PROJECTNAME`
`lp:~USERNAME/+junk/BRANCHNAME`	`lp://qastaging/~USERNAME/+junk/BRANCHNAME`
`lp:~USERNAME/PROJECTNAME/BRANCHNAME`	`lp://qastaging/~USERNAME/PROJECTNAME/BRANCHNAME`

Creating a project

To create a project on Launchpad, you must be logged in, visit the Launchpad front page (`https://launchpad.net/`), and click on the **Register a project** link:

The **Register a project on Launchpad** page asks you to enter various details about your project, which should be fairly straightforward; in any case, here are some additional remarks:

- **URL**: Choose the URL wisely, as it will be a part of all the branch URLs that you will create later for the project. Although it is possible to change the URL later, it can be extremely disruptive for your team and contributors. It is best to pick a good name once and never change it later.
- **Licenses**: If you are registering an open source project, you must specify the license.

After you complete the registration steps, the project will become available at the URL you specified. for example:

`https://launchpad.net/bzrbook-examples`

The last part of the URL, `bzrbook-examples`, is the project's Launchpad ID, and will appear in all the branch URLs related to the project.

Uploading project branches

You can upload branches for a project by pushing to a target URL in the following format:

```
lp:~USERNAME/PROJECTNAME/BRANCHNAME
```

For example:

```
$ bzr push lp:~bzrbuddy/bzrbook-examples/tmp2
Using default stacking branch /+branch-id/707963 at chroot-88678160:///~bzrbuddy/bzrbook-examples/
Created new stacked branch referring to /+branch-id/707963.
```

Since the Launchpad username is part of the branch URL, users effectively have their own namespace for storing branches. Naturally, you can only upload branches in a user's namespace if your SSH key is authorized in the configuration of the corresponding user.

A very important feature of Launchpad is that anybody can associate branches with a project by pushing to a URL where the `PROJECTNAME` part corresponds to the Launchpad ID of the project. In this way, anybody can contribute to a project, without having to obtain access permissions. Of course, whether the maintainers of the project will use the branch or ignore it is a different matter, and we will come back to that on the subject of merge proposals.

In the output of the preceding push operation example, we created a new stacked branch. Stacked branches are another advanced space-saving technique of Bazaar, similar to shared repositories. It is beyond the scope of this chapter; we will explain it in *Chapter 8, Using Advanced Features of Bazaar*.

Viewing project branches

To view all the branches related to a project, visit the **Code** tab of the project's page. In case of the `bzrbook-examples` project, this corresponds to the following URL:

```
https://code.launchpad.net/bzrbook-examples
```

In the main part of the page, you can see the list of all the branches by all the contributors, with basic information about each branch, such as the URL, status, date of last modification, and the last commit message. For example:

Branches with status: Any active status | by most interesting

Name	Status	Last Modified	Last Commit
lp:bzrbook-examples **Series:** trunk	Development	2012-12-15	*This branch is empty.*
lp:~bzrbuddy/bzrbook-examples/tmp2	Development	5 hours ago	15. perl, python, r, ruby, bash implement...
lp:~bzrbuddy/bzrbook-examples/tmp1	Development	5 hours ago	15. perl, python, r, ruby, bash implement...
lp:~bzrbook/bzrbook-examples/distributed-adhoc-mike	Development	2013-04-07	9. merged from steve
lp:~bzrbook/bzrbook-examples/distributed-adhoc-anna	Development	2013-04-07	7. added a.out to ignore patterns
lp:~bzrbook/bzrbook-examples/distributed-adhoc-steve	Development	2013-04-07	7. added asm impl
lp:~bzrbook/bzrbook-examples/distributed-adhoc-jack	Development	2013-04-07	6. added c impl
lp:~bzrbook/bzrbook-examples/hello-fix-c	Development	2013-04-05	8. use more modern include-style

The preceding screenshot provides a list of branches. There are comboboxes to filter by status, and to sort by various criteria. You can also sort by clicking on the column headers.

Viewing your own branches

The **Code** tab of a project shows all the branches related to the project.

The **Code** tab of your account shows all your branches regardless of the project.

Setting a focus branch

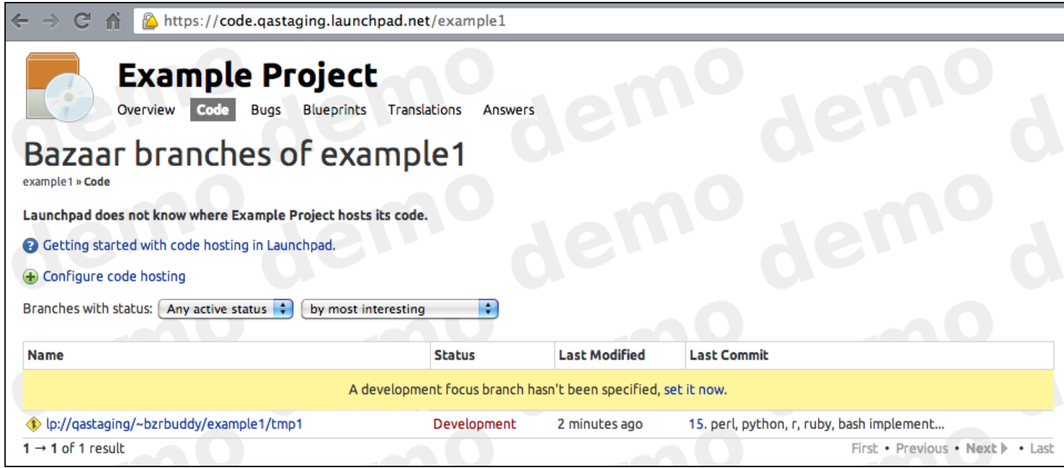
Every project should have a development focus branch, which is accessible in the Bazaar project itself commands by a simplified URL in the following format:

```
lp:PROJECTNAME
```

For example, you can access the current development focus branch of Bazaar by the URL `lp:bzr`, or the MySQL project by `lp:mysql`, and so on.

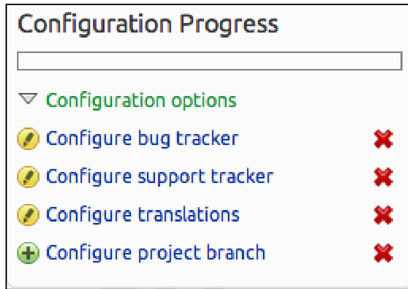
The development focus branch is typically used as the starting point of new feature branches. When you start contributing to a project, ideally you should branch from the development focus branch, implement your improvement, and push your branch to Launchpad to make it visible by the maintainers of the project.

When a project doesn't have a focus branch yet, Launchpad reminds you to set it. For example, the **Code** tab of a newly created dummy project on the Sandbox site shows this when you have at least one branch, but you haven't designated a focus branch yet:

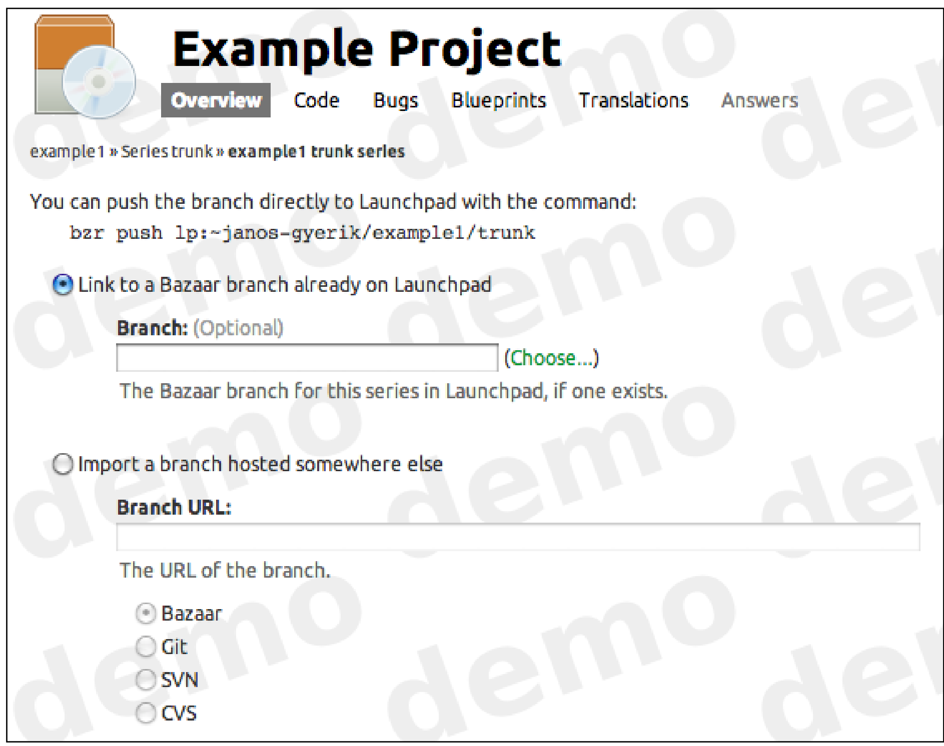


Near the top of the page, Launchpad tells us **Launchpad does not know where Example Project hosts its code**, and in the table with the list of branches, there is a warning message **A development focus branch hasn't been specified, set it now**.

One way to set a focus branch is to click on the **set it now** link in the **Code** tab as shown in the preceding screenshot. Another way is on the **Overview** tab, by clicking on **Configure project branch** in the **Configuration Progress** box at the right-hand side of the page:



Both the ways lead to a page where you can specify an existing Launchpad branch, or even an external branch if it has a public URL that Launchpad can import from:



Finally, another easy alternative to set the focus branch is to push directly to the official URL of the project. For example, in this case `lp:example1`.

The official URL of the project is actually an alias to the configured focus branch. If you push to this URL, Bazaar creates the branch at the location `lp:~USERNAME/PROJECTNAME/trunk`, automatically using your configured Bazaar username, the name of the project, and `trunk` as the branch name. At the same time, the operation sets the official URL to point to the pushed location.

Although anybody can upload branches associated with a project by using the appropriate `PROJECTNAME` part in the URL of a push operation, only drivers of a project can set the focus branch or push to the `lp:PROJECTNAME` location. When you create a project on Launchpad, your account is automatically assigned as the driver of the project. You can confirm and change this setting on the **Overview** tab of the project.

You can let multiple users write to the focus branch by creating a team on Launchpad, adding all privileged users as members, and setting the team as the driver of the project.

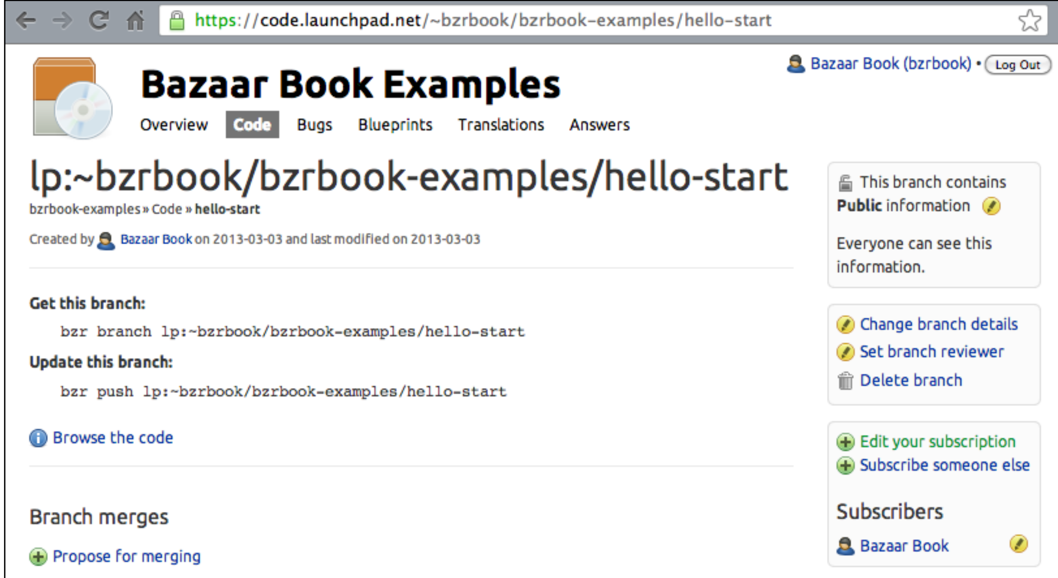
Using series

Branches can also be grouped within so-called "series", which are usually associated with the different releases of the project that are maintained in parallel. For example, the Bazaar project itself has a separate series for all the supported releases, such as "2.5", "2.4", which are accessible by the URLs `lp:bzr/2.5`, `lp:bzr/2.4`, respectively.

You can register a series in the **Overview** tab of a project by using the **Register a series** link.

Viewing and editing branch details

To view the details of a branch, find it in the list of branches in the **Code** tab of the project and click on the URL of the branch in the **Name** column. This is the home page of the branch, showing many important details and providing access to many important functions:



Near the top of the page is the basic information about the branch, such as its URL, the user who created it, and the time of creation.

You can take many important and interesting actions on this page:

- Click on **Browse the code** in the middle to view the files and directories of the branch
- Click on **Change branch details** at the right-hand side to edit the branch details
- Click on **Delete branch** at the right-hand side to delete the branch
- At the right-hand side, you can view and edit the list of users subscribed to notifications triggered by changes to the branch
- Click on **Propose for merging** to propose the branch to merge into another branch on Launchpad
- Click on **Link a bug report** to associate the branch with bugs registered on Launchpad

If you scroll further down, you can see the owner of the branch, the status of the branch, and recent revisions added to the branch.

Using merge proposals

Merge proposals are crucial in a distributed workflow. Since collaborators can only write to their own branches, the only way to get their work into the mainline or to other collaborator branches is to propose them for merging. Launchpad has excellent features to track merge proposals using a web interface.

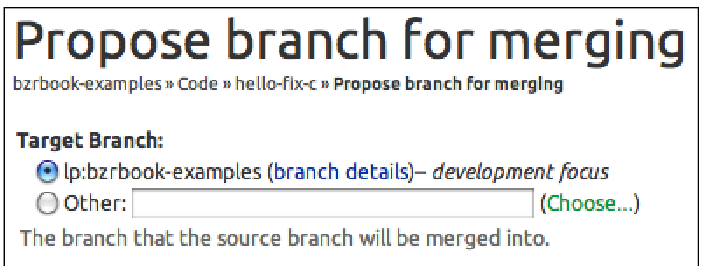
We will demonstrate the process of merge proposals using the following example branches:

- **Bugfix branch**: `lp:~bzrbook/bzrbook-examples/hello-fix-c`
- **Mainline branch**: `lp:~bzrbook/bzrbook-examples/hello-trunk`

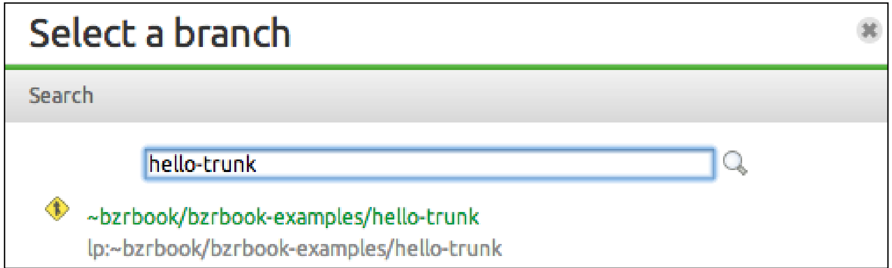
Creating a merge proposal

To propose a branch for merging, open the branch details page and click on **Propose for merging**.

In the form that appears, the only required field is **Target Branch**. Normally, the development focus branch is selected by default, but you can specify another branch by using the **Other** option.



When you specify a branch, you must use the complete URL. If the branch is hosted on Launchpad, you can click on **Choose...** and search for branches by using a keyword in the name of the branch or in the name of the project:



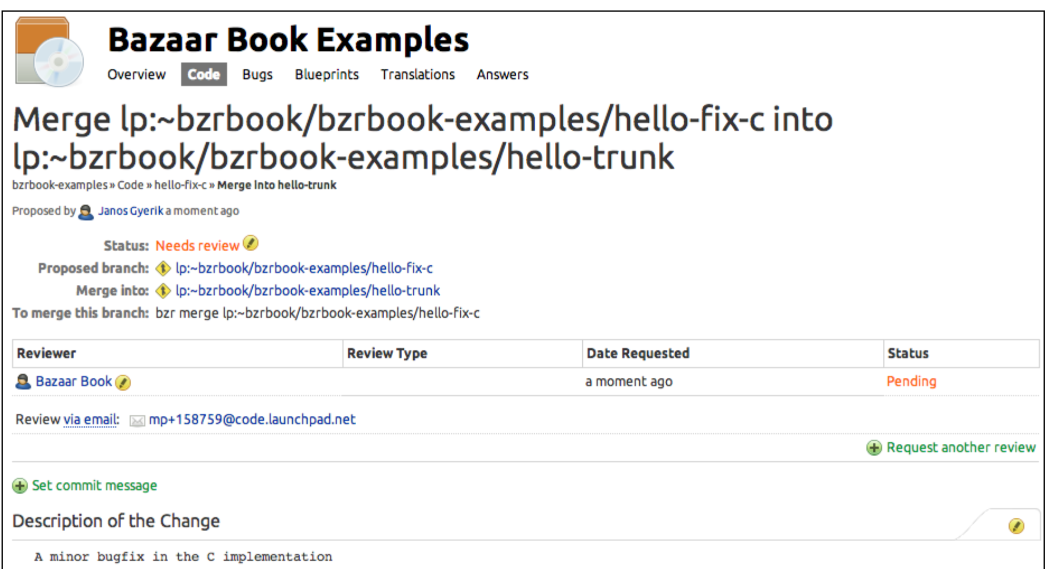
The target branch does not have to be hosted on Launchpad. It can be any Bazaar branch that is accessible by Launchpad's servers. If you specify a foreign branch, Launchpad will schedule to import it, so it becomes available directly on Launchpad. The branch will be imported at `lp:~USERNAME/PROJECTNAME/BRANCHNAME`, where `BRANCHNAME` is derived from the end part of the foreign branch URL.

Ideally, you should fill the other fields too, most importantly the **Description** box to explain the changes in the proposed branch. You may also want to specify the commit message that should be used when the branch is merged after it is approved.

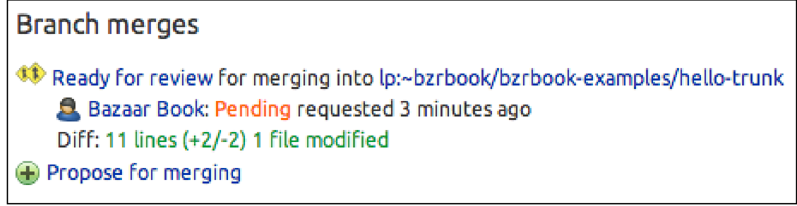
When ready, click on the **Propose Merge** button at the bottom of the page. This will trigger an e-mail notification sent to the specified reviewer, or by default to the project's maintainer, unless you deselected the **Needs review** box. The notification e-mail includes the locations of the source and target branches, and the differences of the two branches in a diff format.

Viewing and editing a merge proposal

A merge proposal has its own dedicated page where you can view and edit its details.



The following page appears right after submitting the merge proposal, or you can access it later from the branch details page:



Everything that you entered while creating the branch proposal can be changed here, except for the locations of the source and target branches.

The author and the reviewers of the branch can invite additional users to participate in the review process. The status of each review is tracked near the top of the page, where reviewers can make changes appropriately, until they ultimately approve or disapprove the change.

The author and the reviewers can enter their comments directly on this page, or by replying to any of the notification e-mails. In this way, they can discuss the details of a merge proposal, or ask the author for additional changes and fixes, until the merge proposal can be approved or disapproved.

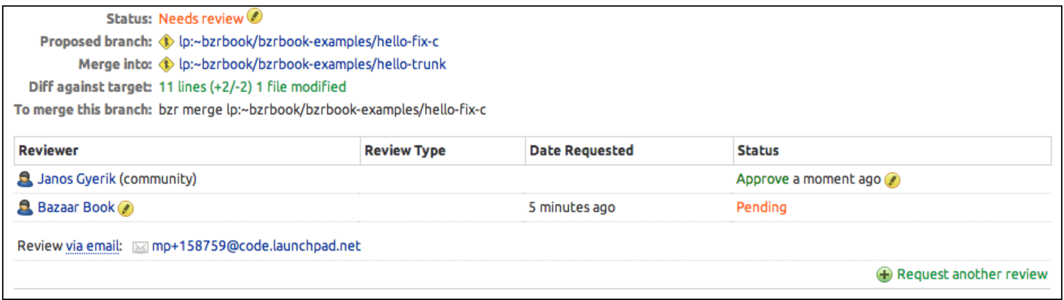
Approving / rejecting a merge proposal

There are two kinds of statuses of a merge proposal tracked on Launchpad—the status of reviews of the changes in the branch, and the status of the conclusion based on all the reviews.

There can be one or more Launchpad users assigned to review a merge proposal, and each one of them can take several actions, such as approve, disapprove, ask for more information, or point out things to fix.

In small projects, there is typically only one reviewer; in larger projects, it may make sense to have more. Any project member can invite more users to participate in the review process of a merge proposal.

Based on the result of all the reviews, the owner of the branch or the driver of the project can make the final conclusion whether to approve or disapprove the merge proposal. This final decision is indicated by the **Status** value near the top of the page, right above the table with the status of all the reviews:



The status of the merge can be changed in several ways:

- Perform the merge and push the updated target branch to Launchpad
- Click on **Status** and change it manually
- Use the e-mail interface with an appropriate command

By performing the merge and pushing the updated target branch to Launchpad, the status of the merge is automatically updated to **Merged** to reflect this action. The merge proposal page gives a hint on how to perform the merge in this example:

To merge this branch: *bzr merge lp:~bzrbook/bzrbook-examples/hello-fix-c*

Assuming the merge proposal was approved by the reviewers and was well-tested, you can perform the merge by performing the following steps:

1. Get the target branch by using `bzr branch` or `bzr checkout`.
2. Merge the proposed branch by using the command given by Launchpad.
3. Commit the merge with a good summary as a comment.

If you got the branch in the first step using `bzr branch`, then you must push the branch to its parent by using `bzr push :parent`, in order to update the Launchpad. If you used `bzr checkout` in the first step, then Launchpad is automatically updated by the last commit. Either way, the end result is the same, and if you reload the merge proposal page, it should now show **Merged** as the value of **Status**.

Note that the other methods of changing the status **Approved** or **Merged** do not perform the merge. Changing the status in this way may be a good way to indicate to the team that the proposal has been approved, but you should not forget to perform the branch and push it to Launchpad.

Using the e-mail interface to handle a merge proposal

Another easy way to change the status of merge proposals is by entering commands in an e-mail sent to a special e-mail address that is associated with the merge proposal and processed by Launchpad. This e-mail address is the one used by the notification e-mails sent from Launchpad about updates on the merge proposal, or you can find it on the merge proposal's page; in our current example, it is as follows:

Review via email: *mp+158759@code.launchpad.net*

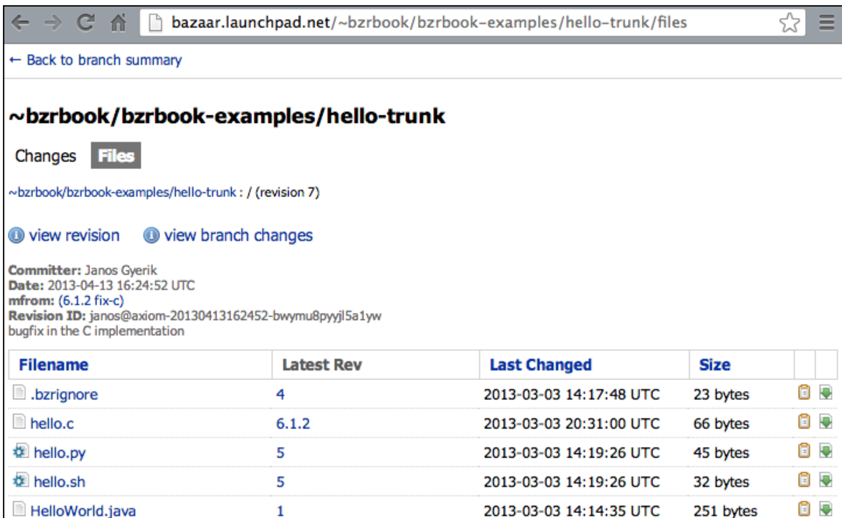
The commands must be entered on separated lines, and each line must start with a space. The following commands are supported:

- `review approve`: This concludes the review and marks it as approved
- `review disapprove`: This concludes the review and marks it as disapproved
- `review abstain`: This abstains from deciding
- `review resubmit`: This tells the collaborator to rework the change and resubmit the merge proposal
- `review needs-fixing`: This tells the collaborator that some fixes are needed
- `review needs-info`: This tells the collaborator that more information is needed
- `merge approved`: This approves the merge proposal
- `merge rejected`: This rejects the merge proposal
- `reviewer NAME`: This invites another Launchpad user to review the merge proposal

Other lines in the e-mail will be used as a comment message, appended to the page. You can find more details in the documentation at `https://help.launchpad.net/Code/Review`.

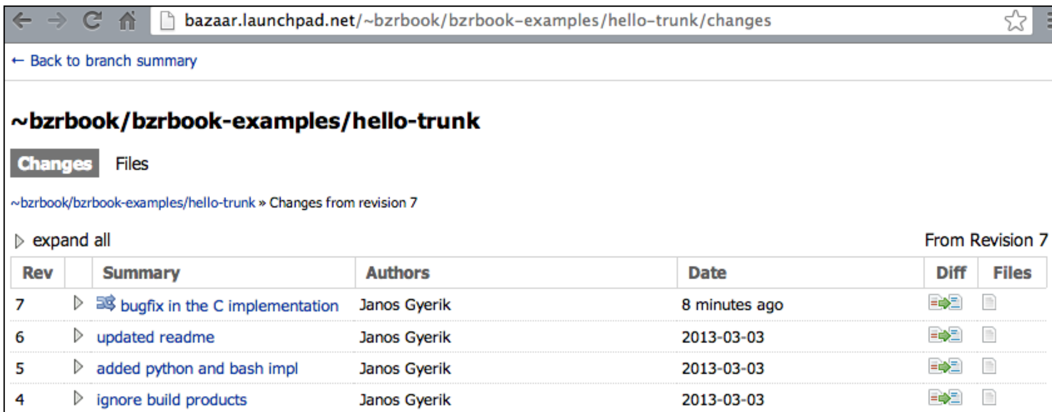
Browsing the content of a branch

You can browse the contents of a branch by clicking on the **Browse the code** link on the branch details page. In the **Files** tab, you can see a list of files at the latest revision:



Above the list of files, you can see a detailed information about the revision, such as the committer, the date, and the commit message.

If you click on the **Changes** tab at the top, you will be able to see a list of changes in a somewhat similar way as the log viewer of Bazaar Explorer:



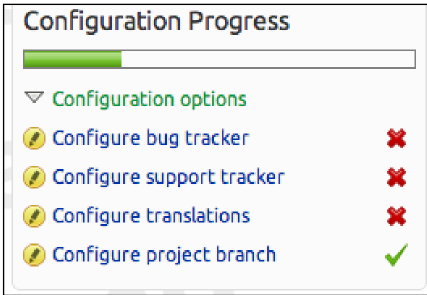
There are several other features available here:

- View the differences in a specific revision or in a file
- Download a revision as a diff or a tarball
- View the full content of any file
- List the revisions that changed a file
- View a file with each line annotated with revision information that changed it

The “browse code" pages on Launchpad are powered by a software called **Loggerhead**. You can find more information about it later in this chapter.

Using the bug tracking system

In order to use the bug tracking system of Launchpad, you must enable it in the configuration of the project, by using the **Configure bug tracker** link at the right-hand sidebar:



On the bug tracker configuration form, simply select Launchpad; the other fields are optional. When done with the editing, click on **Change** at the bottom.

Launchpad projects can work with external bug trackers too, such as Trac, Mantis, Bugzilla, and Redmine. The complete list of compatible bug trackers is maintained at `https://help.launchpad.net/Bugs/RemoteTrackerCoverage`.

Reporting bugs

On the **Bugs** tab, click on **Report a bug** and follow the instructions.

First, the page asks for the single-line summary of the bug, which will be the title of the bug showed in the listings. After this step, Launchpad will search for the existing bugs reported for the project and show the ones that might be similar. This is in order to reduce duplicate bug reports of the same issue.

On the next page, you can enter the description of the bug, as detailed as possible. You can enter details such as **Status**, **Importance**, **Milestone**, and **Tags**, and assign the bug to a particular user. Normally, these fields are better to leave for the maintainer of the project to enter. The default values are **New** and **Undecided** for **Status** and **Importance**, respectively.

Linking commits to bugs

See the *Linking commits to bug trackers* section for a general explanation and the specific steps to be taken when using Launchpad with various bug trackers.

Useful tips when using Launchpad

There are a few additional tips that may be good to know when using Launchpad.

Deleting or renaming a project

There is no user interface for these actions. If you are really sure you want to do this, the current official way is to create a question on the Launchpad project itself:

`https://launchpad.net/launchpad`

The karma system

On your account page, you may have noticed a Karma value. By using Launchpad, you accumulate Karma points. The more active you are, the more Karma points you will collect. However, Karma points expire with inactivity. You can learn more about how Karma points work by clicking on the question mark icon next to it, or at the following URL:

`https://help.launchpad.net/YourAccount/Karma`

Hosting private projects

It is possible to host private projects on Launchpad. You can read more about the various commercial hosting options at the following URL (part of the Launchpad Tour):

`https://launchpad.net/+tour/join-launchpad#commercial`

Integrating Bazaar into Redmine

Redmine is a flexible project management web application that integrates repository browsing, bug tracker, wiki, forums, and so on. It supports Bazaar repositories natively, and it is quite easy to link a Redmine project to Bazaar.

Configuring Redmine itself is beyond the scope of this book; here, we assume that you already have a working Redmine installation and focus on how to enable Bazaar for it:

1. Go to the global site, navigate to **Administration** | **Settings** | **Repositories**, and make sure that Bazaar is enabled. The `bzr command` must be installed and accessible by Redmine. If `bzr` is not on the `PATH` variable used by Redmine, then you can specify the absolute path explicitly in the `config/configuration.yml` file with the `scm_bazaar_command` setting. You will need to restart Redmine after this change.
2. Go to the project's **Settings** | **Modules** page, and make sure that the **Repository** module is enabled.
3. Go to the project's **Settings** | **Repository** page, set **SCM** to **Bazaar**, enter the absolute path to the Bazaar branch, and enter the encoding used by commit messages, for example, `UTF-8`.

After this, you should be able to browse the Bazaar branch by using the **Repository** tab, as follows:

You can browse the contents of versioned files and directories and see other details such as the following:

- View the files and directories at a specific revision
- View the differences between any two revisions
- View each line of a file annotated with the revision information that changed it
- View the list of revisions that changed a file
- View the log of revisions in a branch

Integrating Bazaar into Trac

Trac is a web application that integrates repository browsing, bug tracker, and wiki. Support for Bazaar repositories can be enabled in Trac by installing the **Trac Bazaar plugin** (also known as **trac+bzr**).

Configuring Trac is beyond the scope of this book; here, we assume that you already have a working Trac installation and focus on how to enable Bazaar support for it.

Enabling the plugin globally

The best way to install Trac is by using your operating system's package manager. Look for a package named `tracbzr`. Alternatively, you can install it by using **Pip**, the Python package manager, or from source.

```
$ pip install tracbzr bzr==2.5
```

The Bazaar libraries are a runtime dependency of the plugin, that's why we need to install `bzr` too. We specified an explicit version of `bzr`, because by default Pip installs the latest unstable version of `bzr`, which might not always work well.

Another important point is to use the same Python version as the one Trac is running with. For example, if you are normally using Python 2.7, but Trac is running with Python 2.6, then in the preceding command you should use `pip-2.6` instead. You can confirm the Python version used by pip by using the command `pip --version`.

After the plugin is installed, enable it in the `trac.ini` file of the Trac project's environment by editing or adding a components section as follows:

```
[components]
tracbzr.* = enabled
```

Finally, add your Bazaar branch locations by using the `trac-admin` command of Trac:

```
$ trac-admin ENV repository add NICK PATH bzr
```

Here:

- `ENV` is the path to the Trac project environment
- `NICK` is a short name to identify the location in Trac
- `PATH` is the path to a directory; it can be a branch, a shared repository, or just a plain directory

The `bzr` parameter at the end of the command is to indicate that this is a Bazaar repository, so that Trac knows the right plugin to use when working with it.

To remove repository locations from Trac, use the `remove` command:

```
$ trac-admin ENV repository remove NICK
```

After adding or removing locations, you may need to resync Trac's database:

```
$ trac-admin ENV repository resync NICK
```

You can resync all locations at once by using the following command:

```
$ trac-admin ENV repository resync '*'
```

Enabling the plugin for one project only

If you don't want to install the `tracbzr` Python module system-wide, then another option is to package the module as an `egg` file and drop it into the `plugins` directory in the project environment.

You may be able to get an `egg` file from the Launchpad page of the Python module:

`https://launchpad.net/trac-bzr`

Look for the **Downloads** section at the right-hand side; there are usually several `egg` packages corresponding to different Python versions.

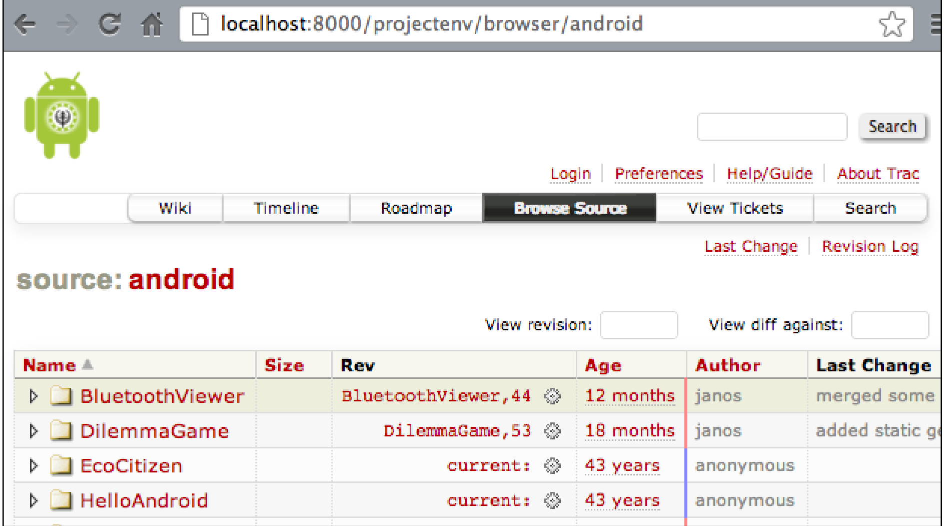
If there is no download file for your version of Python, or if you prefer to build the package yourself, that's easy enough to do. In this case, download the tarball and build the package with the `bdist_egg` command as follows:

```
$ tar zxf TracBzr-0.4.2.tar.gz
$ cd TracBzr-0.4.2/
$ python setup.py bdist_egg
$ ls dist/
TracBzr-0.4.2-py2.7.egg
```

This creates the `egg` package inside the `dist/` directory. Copy the package to the `plugins` directory of your Trac project environment and restart Trac.

Browsing Bazaar branches

If the Bazaar plugin has been successfully configured, then the **Browse Source** tab should be visible, and you should be able to browse the directory tree of Bazaar branches:



You can browse the contents of versioned files and directories and see other details such as the following:

- View the files and directories at a specific revision
- View the differences between any two revisions
- View each line of a file annotated with the revision information that changed it
- View the list of revisions that changed a file
- View the log of revisions in a branch

Beware that there are some limitations of the plugin, as documented in the *Limitations* section of the plugin's homepage:

`http://pypi.python.org/pypi/TracBzr`

Getting help

For more details, see the plugin's Launchpad page:

`https://launchpad.net/trac-bzr`

Or the plugin's project homepage:

`http://pypi.python.org/pypi/TracBzr`

Linking commits to bug trackers

If you are using a bug tracker in your project, you can link Bazaar commits to bug reports. When a commit is linked to a bug report, you will be able to:

- See the bug report's URL in the output of `bzr log`
- Have a clickable hyperlink to the bug report when viewing the revision history using Bazaar Explorer
- Depending upon the bug tracker, you may be able to see the details of the linked commits on the bug report's page

In order to link commits to bugs, you must do two things:

1. Edit the branch configuration to set the bug tracker.
2. Specify the bug ID using the `--fixes` flag when committing revisions.

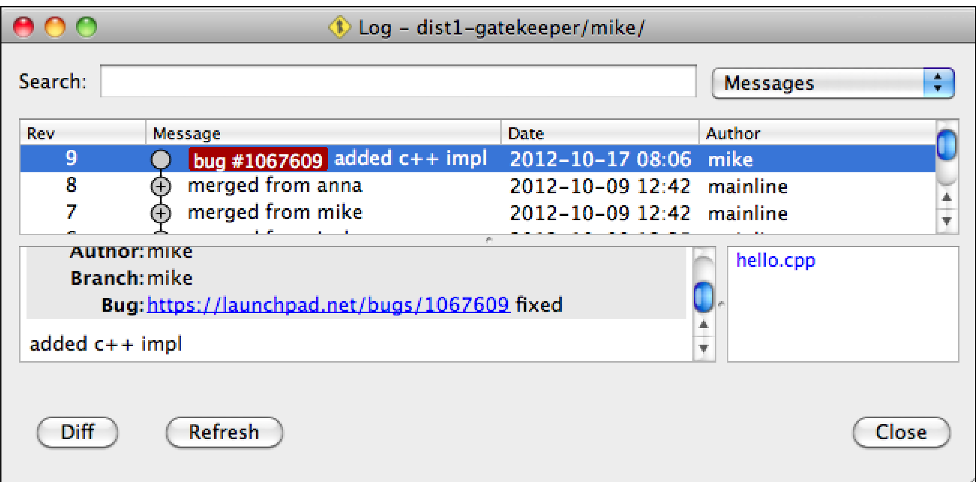
Depending upon the bug tracker, the configuration is slightly different. Here, we will demonstrate how to link to bugs on Launchpad, Bugzilla, Redmine, and Trac. If you use a different bug tracker, then these examples may help you figure out the right steps.

When viewing the revision history by using `bzr log`, linked bugs appear as follows:

```
$ bzr log --short --limit 1
    9 mike      2012-10-17
      fixes bug: https://launchpad.net/bugs/1067609
      added c++ impl
```

Notice the `fixes bug` line with the URL of the bug report on Launchpad. This revision was committed by using the `--fixes lp:1067609` option. Linked bug reports are shown only in the short and long formats of the `log` command, not in the single-line format.

When viewing the revision history using Bazaar Explorer, the revisions associated with bug reports are highlighted with the bug's ID, and the bug's URL is shown in the commit's details. You can click on the highlighted label of the linked bug to open the bug report in a browser:



Configuring bug trackers in Bazaar

Bug trackers can be configured globally or per branch. A bug tracker configuration in Bazaar typically has two important pieces of information:

- A short ID to identify the bug tracker
- The parameterized URL of the tracker

The tracker ID is used when linking to a bug report with the `--fixes` flag of the `commit` command. For example:

```
$ bzr commit -m 'added c++ impl' --fixes lp:1067609
Committing to: /sandbox/integration/mike/
added hello.cpp
Committed revision 9.
```

The parameter `--fixes` is in the format `TRACKERID:BUGID`;in this case, the tracker ID is `lp`, which Bazaar automatically associates with Launchpad. The bug ID is, of course, the ID of the bug in the bug tracking system; in this case on Launchpad.

The general format of a bug tracker configuration looks similar to the following:

```
bugtracker_TRACKERID_url = URL/{id}
```

Here:

- `TRACKERID` is your choice of a short name to identify the bug tracker
- `URL` is the URL template of bug report pages
- `{id}` is what Bazaar will substitute with the bug's ID when constructing the bug's URL to display in the revision history

For example:

```
bugtracker_redmine1_url = http://example.com/redmine/issues/{id}
```

This configuration identifies a bug tracker named `redmine1`, so that if you create a commit as follows:

```
$ bzr commit -m 'test commit' --fixes redmine1:123
```

Then the bug report's link will be shown in the revision history as follows:

```
$ bzr log -l1 -S
   10 Janos Gyerik      2012-10-17
      fixes bug: http://example.com/redmine/issues/123
      test commit
```

Bug tracker configurations can be added at different places:

- `~/.bazaar/bazaar.conf`: This is the global configuration file, add your change in the `[DEFAULT]` section
- `.bzr/branch/branch.conf`: This is the configuration file of the branch
- `~/.bazaar/locations.conf`: This is the global configuration file of repository locations

To test a bug tracker configuration, try to do a test commit in a test branch by using the new tracker ID. The commit operation will fail if there is a problem in the configuration. For example:

```
$ bzr commit -m 'test commit' --fixes myNonExistent:123
bzr: ERROR: Unrecognized bug myNonExistent:123. Commit refused.
```

Linking to public bug trackers

Bazaar has additional support for famous public bug trackers in order to simplify the configuration:

URL	Tracker ID	Example
`https://bugs.launchpad.net/`	`lp`	`lp:12345`
`http://bugs.debian.org/`	`deb`	`deb:12345`
`http://bugzilla.gnome.org/`	`gnome`	`gnome:12345`

You can use these trackers without any additional configuration.

Linking to Launchpad

In projects hosted on Launchpad you can use the `lp` tracker without any additional configuration.

Using Launchpad has some additional benefits. For example, after you push a branch containing references to bugs on Launchpad, the referenced bug report pages will have a new section titled **Related branches**, where the branches will be listed. Similarly, on the branch details page, links to the bug reports will be added automatically in the **Related bugs** section.

You can also manually link branches to bugs, by using the **Link a bug report** link in the **Related bugs** section. Linking branches to bug reports is very convenient because the branch details page will show all the linked bugs with their statuses and importance.

Linking to Bugzilla

Bazaar has additional support for Bugzilla in order to simplify the configuration:

```
bugzilla_TRACKER_url = URL
```

That is, you don't need the `{id}` parameter. The URL should be the base URL of the project in Bugzilla. For example:

```
bugzilla_mybugz_url = http://example.com/
```

In this way, the bug report URLs will be generated in the following format:

```
http://example.com/show_bug.cgi?id=123
```

Linking to Trac

Bazaar provides additional support for Trac in order to simplify the configuration:

```
trac_TRACKER_url = URL
```

That is, you don't need the `{id}` parameter. The URL should be the base URL of the project environment in Trac. For example:

```
trac_mytrac_url = http://example.com/trac
```

In this way, the bug report URLs will be generated in the following format:

```
http://example.com/trac/ticket/123
```

Linking to other bug trackers

To link to other bug trackers, you must use the format with the generic prefix `bugtracker_` as the configuration name, and the `{id}` parameter in the URL. For example:

```
bugtracker_redmine1_url = http://example.com/redmine/issues/{id}
```

See `bzr help bugs` for more details.

Advanced integration with bug trackers

In this section, we focused mainly on linking Bazaar commits to bug trackers, so that you can easily open bug reports from Bazaar Explorer, or by using the links in the output of `bzr log`.

Depending upon the bug tracker, sometimes the reverse is also possible, and the bug tracker can link back to Bazaar branches, showing details about the commits related to bugs, similar to what Launchpad does.

One such example is `bugzilla-vcs`, an extension of Bugzilla that provides integration with Bazaar and other version control systems. For more details, see their project website:

`https://code.google.com/p/bugzilla-vcs/`

Other bug trackers may also have the extension to provide a similar functionality.

Web-based repository browsing with Loggerhead

Loggerhead is a web viewer for Bazaar branches. It lets you do the following:

- Browse the branch history
- View files at a given revision
- Annotate files showing the origin of each line

Loggerhead powers the repository browsing features on Launchpad. If you prefer to host your Bazaar repositories yourself, you can install Loggerhead on your own server.

Installing Loggerhead

The best way to install Loggerhead is by using your operating system's package manager. Look for a package named `loggerhead`. In case that is not an option for you, we explain how to install Loggerhead by using `pip` and `virtualenv`.

When installing web tools written in Python, it is always recommended to use `virtualenv` in order to isolate the tool's Python dependencies from the rest of the Python packages in the system. This is a good way to give Loggerhead a try, and since all of its files and dependencies will be contained within a single directory, it is also easy to clean up after testing.

First, let's create the “virtual environment" where we will install Loggerhead:

```
$ virtualenv --distribute loggerhead
New python executable in loggerhead/bin/python
Installing distribute...........................................
..................................................................
..................................................................
..........................done.
Installing pip................done.
```

A directory named `loggerhead` was just created, where all Python libraries and scripts that we are going to install will be stored, such as Loggerhead and its dependencies. This works by setting up environment variables such as `PATH` and `PYTHONPATH` appropriately, which we can do easily by sourcing the activation script of the virtual environment:

```
$ . loggerhead/bin/activate
(loggerhead)$
```

When a virtual environment is activated, the shell prompt is changed to indicate the name of the virtual environment; in this case the prompt became `(loggerhead)$` instead of `$`. This simply means that now environment variables such as `PATH` and `PYTHONPATH` are configured in a way that anything we install using `pip` or `python setup.py install`, they will be installed within the directory of the virtual environment.

Next, let's install Loggerhead and its dependencies using `pip`:

```
(loggerhead)$ pip install paste bzr simplejson
# ... (skip)
(loggerhead)$ pip install loggerhead
# ... (skip)
```

There is one more dependency, which is not available via pip, called **SimpleTAL**, a template language. We need to install this in the old fashioned way, from a tarball. You can get the latest version of the Python 2.x series from the following URL:

`http://www.owlfish.com/software/simpleTAL/py2compatible/download.html`

Unpack and install the python module with:

```
(loggerhead)$ tar zxf SimpleTAL-4.3.tar.gz
(loggerhead)$ cd SimpleTAL-4.3
(loggerhead)$ python setup.py install
# ... (skip)
```

That's it, now we are ready to run Loggerhead!

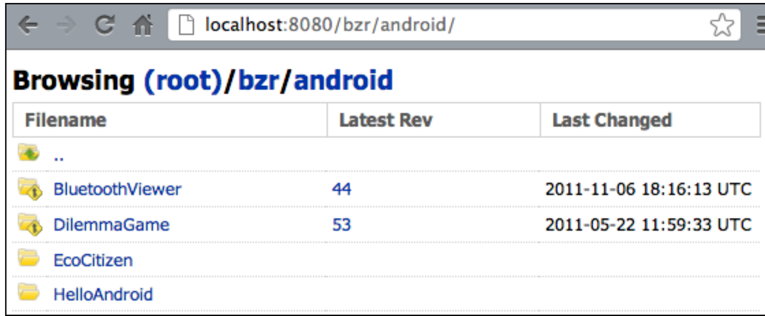
Running Loggerhead locally

An easy way to test Loggerhead is to run it by using its built-in web server:

```
(loggerhead)$ serve-branches file:///srv/bzr
```

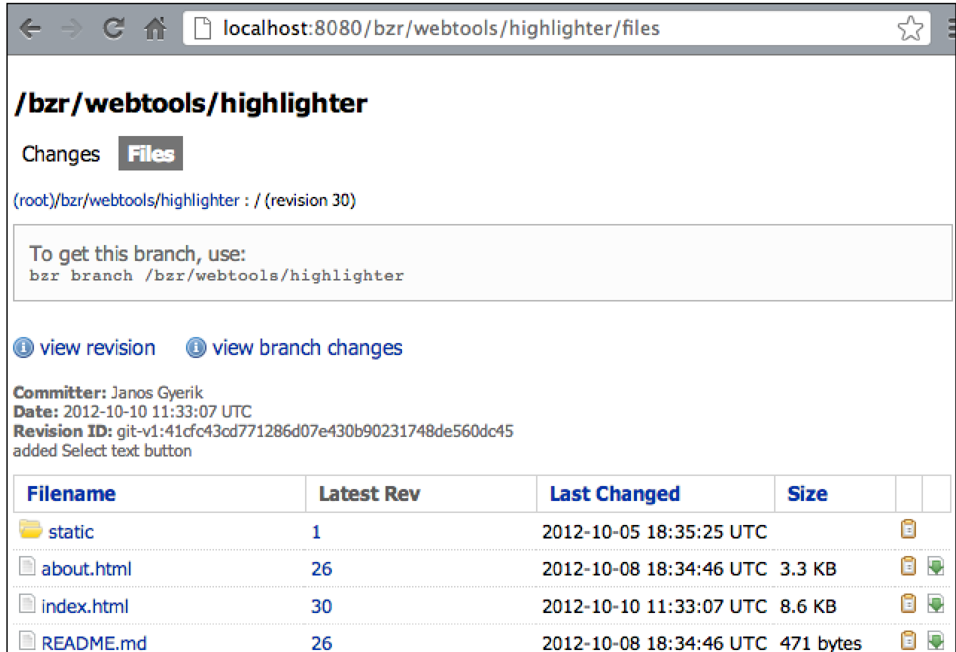
The single parameter in this example is the path to a directory, typically the parent directory or the shared repository that contains several Bazaar branches.

By default, the web server will listen on port `8080` of `localhost`. Thus if you visit `http://localhost:8080/`, you should see the contents of the specified root directory and browse its contents:



In this example, there is a mix of Bazaar branches and regular directories, which can be distinguished by their icons, and the extra information by Loggerhead such as the latest revision number and timestamp.

When inside a branch, the view is much like on Launchpad:



For example, we can download this branch with the following command:

```
$ bzr branch http://localhost:8080/bzr/webtools/highlighter
```

Running Loggerhead in production

There is a lot more to Loggerhead than we can cover here. For a brief introduction of other configuration options, such as running Loggerhead behind an Apache web server, see the documentation in the Bazaar admin guide:

`http://doc.bazaar.canonical.com/beta/en/admin-guide/code-browsing.html#loggerhead`

To see the complete list of command-line options of serve-branches, use the `-h` or `--help` flags.

Summary

In this chapter, we introduced the various collaborative development environments that Bazaar is known to work well with, such as Launchpad, various bug tracking systems, and Loggerhead. You should be able to use Bazaar together with these tools and benefit from their features in order to keep your projects organized.

There is much more to be discovered about integrating Bazaar with other tools, and due to space limitations, we cannot cover everything here. Hopefully, the examples in this chapter will help you integrate Bazaar with other systems in your toolset.

In the next chapter, we will learn how to use some of the more advanced features of Bazaar, along with additional practical tips and tricks that you may find useful, which will further improve your productivity.

8

Using the Advanced Features of Bazaar

By now, you should have a solid understanding of Bazaar's philosophy and must be able to perform all the most important version control operations with ease. This chapter will show additional practical tips that are not essential to using Bazaar, but can be very useful and make you more productive.

The following topics will be covered in this chapter:

- Using aliases
- Undoing commits
- Shelving changes
- Using lightweight checkouts
- Re-using a working tree
- Using stacked branches
- Signing revisions using GnuPG
- Configuring a hook to send e-mails upon commit

Using aliases

Aliases are helpful to shorten long commands that you use often. Take for example the `log` command. Running `bzr log` without parameters in large projects would produce an awful lot of an output, so it is compulsory to add the `--limit` parameter. Furthermore, the default output format is long. This may be too much detail for everyday use, and the single-line format may be informative enough for most purposes. Thus, we end up with, for example, the following:

```
$ bzr log --limit 5 --line
```

This is rather long to type if you use it often. Luckily, we can shorten it with an alias; let's call it `l5`, as follows:

```
$ bzr alias l5='log --limit 5 --line'
```

You can use aliases as if they were Bazaar commands. For example:

```
$ bzr l5 lp:bzr
6573: Patch Queue Manager 2013-02-07 [merge] (jameinel) Fix bug #1107464,
6572: Patch Queue Manager 2012-12-10 [merge] (vila) Fix LC_ALL=C test
failur...
6571: Patch Queue Manager 2012-10-25 [merge] (gz) Set approved revision
and ...
6570: Patch Queue Manager 2012-10-14 [merge] (jelmer) Fix trivial syntax
err...
6569: Patch Queue Manager 2012-10-11 [merge] (vila) Clarify how
`mergetool` ...
```

To view the definition of an alias, run `bzr alias` with the name of the alias as a parameter. For example:

```
$ bzr alias l5
bzr alias l5="log --limit 5 --line"
```

To see all your currently defined aliases, run `bzr alias` without any parameters. For example:

```
$ bzr alias
bzr alias l="log --line -l10 -n0"
bzr alias l5="log --limit 5 --line"
bzr alias ll="log --line -l10"
bzr alias s="status"
```

To remove an alias, use the `--remove` flag:

```
$ bzr alias --remove ll
```

Aliases are stored in the `[ALIASES]` section in the user configuration file `~/.bazaar/bazaar.conf`:

```
[ALIASES]
l = log --line -l10 -n0
s = status
l5 = log --limit 5 --line
```

You can either use the `bzr alias` command or edit the configuration directly; the end result will be the same.

You can even override a standard Bazaar command with an alias if you prefer:

```
$ bzr alias log='log --short'
```

This example effectively overrides the default output format of `bzr log`, using the short format instead of the default long format.

To temporarily disable an alias, for example, to fall back on the default Bazaar command, you can use the `--no-aliases` flag.

Undoing commits

You can undo one or more of the most recent commits by using the `uncommit` operation. This can be useful, for example, if you want to amend your last commit by changing the log message or adjusting the set of changes to include. The `uncommit` operation moves the branch tip marker one or more revisions back, without changing the working tree, so that you can make any necessary adjustments and commit again.

To see how it works, let's grab a sample branch:

```
$ bzr branch lp:~bzrbook/bzrbook-examples/uncommit /tmp/uncommit
Branched 6 revisions.
```

You can undo the last commit by using the `bzr uncommit` command without any parameters:

```
$ cd /tmp/uncommit/
$ bzr uncommit
    6 Janos Gyerik      2013-04-20
      changes just to demonstrate shelving

The above revision(s) will be removed.
Uncommit these revisions? ([y]es, [n]o): yes
You can restore the old tip by running:
  bzr pull . -r revid:janos@axiom-20130420203136-bi9iglm2cevc8tfq
```

Bazaar shows the short log message of the revisions it is about to uncommit, and prompts for confirmation. To undo the listed revisions, press *Y* on your keyboard; otherwise, the operation will be aborted.

As a result, the branch tip will be moved backwards to point to the previous revision, while the working tree will be unchanged. The revision pointed to by the branch tip previously is not deleted; you can still access it by its revision ID and restore if necessary by using the hint in the output of the `uncommit` command.

The changes that were a part of the uncommitted revision will now appear pending, which you can confirm by using the `status` command:

```
$ bzr status
renamed:
  images/ => maps/
modified:
  guests.txt
  menu.txt
```

At this point, you will have various options, such as the following:

- Commit again but using a different log message, for example, to fix a typo
- Commit again but using only a subset of all changes, for example, to split a large commit to smaller chunks of changes
- Revert everything and continue in a completely different direction

You can uncommit more than one revision by specifying a revision parameter by using `-r` or `--revision`. All the revisions between the current branch tip and the specified revision will be uncommitted; for example, the command `bzr uncommit -r23` will leave the branch at revision `23`.

To uncommit using Bazaar Explorer, click on the large **Work** button in the toolbar and select **Uncommit Revisions...**, or navigate to **Bazaar | Work** and select the **Uncommit Revisions...** option.

Keep in mind that it is strongly discouraged to uncommit revisions in branches that are shared with others, as your collaborators may have already started using the branch before the tip has changed, leading to confusion.

Shelving changes

The shelving changes feature lets you to temporarily set aside changes in the working tree. You can select specific changes to be put "on a shelf" from where you can restore them later. Once on the shelf, the changes are reverted in the working tree to move them out of the way.

This is useful when in the following scenarios:

- You want to keep your commits clean—some unrelated changes got mingled into your current main focus work and you want to set them aside to commit later.
- You want to merge or pull from another branch but some pending changes that you don't want to commit yet are blocking your way
- You want to revert some but not all of the changes within the files

An important thing to keep in mind is that the shelf is not part of the repository, but is stored inside the working tree. As a consequence, it is not propagated through branch operations, and if the working tree is deleted, the shelf will also be gone.

Using shelves is significantly easier in Bazaar Explorer as compared to the command line. However, at the time of this writing, this feature of Bazaar Explorer does not work reliably in all systems, thus we will focus on the command-line interface. Throughout the examples, replace the `bzr shelve` and `bzr unshelve` commands with `bzr qshelve` and `bzr qunshelve`, respectively, whenever possible.

Putting changes "on a shelf"

The command to set aside changes "on a shelf" is `bzr shelve`. Similar to other Bazaar commands, you can specify a set of files or directories, or run the command without any parameters to shelve all the changes.

By default, the command runs interactively—it presents each hunk of change one by one and prompts to decide whether to put the change on the shelf or not.

> **Hunk** is a term commonly used when working with traditional UNIX-style patches. It contains a changed line in unified diff format, with a few lines of context before and after the changed line. If the changed lines are so close to each other that their hunk would overlap, then the two changes will appear in a single hunk.

To demonstrate how shelving works, let's grab a sample branch and revert the last commit to have an interesting set of changes:

```
$ bzr branch lp:~bzrbook/bzrbook-examples/shelving /tmp/shelving
Branched 6 revisions.
$ cd /tmp/shelving
$ bzr revert -rlast:2
 M  guests.txt
R   maps/ => images/
 M  menu.txt
```

Let's shelve some of the changes in `menu.txt` only:

```
$ bzr shelve menu.txt
--- menu.txt    2013-04-20 20:31:36 +0000
+++ menu.txt    2013-04-20 20:34:36 +0000
@@ -1,7 +1,7 @@
 Menu
 ====
 Guacamole
-Corn chipz
+Corn chips
 Spicy tomato sauce
 Minestrone soup
```

```
 Buffalo wings
Shelve? ([y]es, [N]o, [f]inish, [q]uit): yes
@@ -15,4 +15,4 @@
 Beef burrito
 Mixed burrito
 Onion soup
-Tacoz
+Tacos
Shelve? ([y]es, [N]o, [f]inish, [q]uit): no
Selected changes:
 M  menu.txt
Shelve 1 change(s)? ([y]es, [N]o, [f]inish, [q]uit): yes
Changes shelved with id "1".
```

For each hunk in the file, Bazaar asks to shelve or not and gives you a set of choices:

- `[y]es`: To shelve this change, press *Y*
- `[n]o`: To not shelve this change, press *N*
- `[f]inish`: To shelve this change and the changes selected so far and all the remaining changes, press *F*
- `[q]uit`: To not shelve anything and exit now, press *Q*

After reviewing all the changes, Bazaar offers a final choice; this time also indicating the number of changes that have been selected for shelving, and a summary of the changes in a similar way as in the output of a commit operation.

If we check the status now, Bazaar tells us that a shelf exists:

```
$ bzr status
renamed:
  maps/ => images/
modified:
  guests.txt
  menu.txt
1 shelf exists. See "bzr shelve --list" for details.
```

Notice that `menu.txt` still appears to be changed, which is normal because in `bzr shelve` we said "yes" to the first change but "no" to the second one. We can confirm this by using `bzr diff`:

```
$ bzr diff menu.txt=== modified file 'menu.txt'
--- menu.txt    2013-04-20 20:31:36 +0000
+++ menu.txt    2013-04-20 20:37:47 +0000
@@ -15,4 +15,4 @@
 Beef burrito
 Mixed burrito
 Onion soup
-Tacoz
+Tacos
```

The first change has been reverted in the file and added to the shelf. The second change is still there in the file, since we said "no" earlier when Bazaar asked to shelve it or not.

You can bypass the interactive mode and shelve all changes in the specified files by using the `--all` flag. For example:

```
$ bzr shelve --all -m 'guests and menu changes' guests.txt menu.txt
Selected changes:
 M  guests.txt
 M  menu.txt
Changes shelved with id "2".
```

This time we have specified a message by using the `-m` option, which will be useful later when viewing the list of shelves.

Shelving works not only for content changes but other kinds of changes too, such as renames, and deletions. In cases like these, when it doesn't make sense to show a diff, Bazaar will simply ask if the change should be shelved or not. For example:

```
$ bzr shelve
Shelve renaming "maps" => "images"? ([y]es, [N]o, [f]inish, [q]uit): no
No changes to shelve.
```

Listing and viewing shelved changes

To see the list of shelves, use the following command:

```
bzr shelve --list,
```

For example:

```
$ bzr shelve --list
  2: guests and menu changes
  1: <no message>
```

This shows the ID and the message string of each shelf. The ID is incremented by one each time a new shelf is created. Using a meaningful message string when creating a shelf is important in order to give a hint about the content of the shelf when viewing the list of shelves later.

To see a quick summary of changes on a shelf, use `bzr unshelve --dry-run`:

```
$ bzr unshelve --dry-run
Using changes with id "2".
Message: guests and menu changes
 M  guests.txt
 M  menu.txt
```

To see the content of a shelf, use `bzr unshelve --preview`:

```
$ bzr unshelve --preview 1
Using changes with id "1".
 M  menu.txt
=== modified file 'menu.txt'
--- a/menu.txt  2013-04-20 20:47:31 +0000
+++ b/menu.txt  2013-04-20 20:53:06 +0000
@@ -1,7 +1,7 @@
 Menu
 ====
 Guacamole
-Corn chipz
+Corn chips
 Spicy tomato sauce
 Minestrone soup Buffalo wings
```

The `bzr unshelve` command works with the last shelf by default. To use a different shelf, specify the ID of the shelf, as we did in this previous example.

Restoring shelved changes

To restore changes from a shelf, use the `bzr unshelve` command. For example:

```
$ bzr shelve --all
Selected changes:
R   images/ => maps/
Changes shelved with id "3".
$ bzr unshelve
Using changes with id "3".
R   maps/ => images/
All changes applied successfully.
Deleted changes with id "3".
```

This command uses the most recent shelf by default, in order to ensure that the changes are restored in the correct order. If you want to restore another shelf first, you have to specify the ID of the shelf explicitly. For example:

```
Using changes with id "1".
 M  menu.txt
All changes applied successfully.
Deleted changes with id "1".
```

Be careful when restoring shelves in a different order from the default, as this can work well only when the changes on the different shelves are unrelated to each other, otherwise there may be conflicts as a result.

By default, the restored shelf is automatically removed. To restore a shelf without removing it, use the `--keep` flag:

```
$ bzr shelve --list
  2: guests and menu changes$ bzr unshelve --keep
Using changes with id "2".
Message: guests and menu changes
 M  guests.txt
 M  menu.txt
All changes applied successfully.
```

To remove a shelf without applying the changes in it, use the `--delete-only` flag:

```
$ bzr unshelve --delete-only
Deleted changes with id "2".
```

Using shelves to revert partial changes in a file

The `revert` command doesn't have an option to revert changes in a file partially, but we can achieve this effect by using shelving and the `--destroy` flag.

When using the `bzr shelve` command with the `--destroy` flag, the selected changes will be cleaned up in the working tree instead of saved on a shelf, thus effectively reverting them.

Another option to achieve the same effect is to shelve the changes you want to delete, and then either never unshelve them, or delete the shelf by using `bzr unshelve --delete-only`.

Using shelves to commit partial changes in a file

The `commit` command doesn't have an option to commit changes in a file partially, but you can achieve this effect by using shelving operations.

The way to commit partial changes is to first shelve unwanted changes. Run `bzr shelve`, possibly specifying the set of files to consider, and say "yes" to everything except the changes you want to include in the commit. As a result, all the unwanted changes will be out of the way on a shelf, and you can go ahead with your commit.

After the commit, you can either unshelve the changes you shelved earlier, to continue working with them and perhaps include in the next commit, or discard them by using `bzr unshelve --destroy`.

Using lightweight checkouts

A lightweight checkout is a special configuration with only a working tree and a branch, but without a repository. In this configuration, the branch is bound to another branch, named the **master branch**, in the same way as in a regular checkout. Since there is no local repository, all the operations that access the revision data must go through the master branch. In practice, this has the following consequences:

- The checkout operation is much faster as compared to a regular checkout or branching, because the full revision history is not copied over from the master branch.

- Operations using the revision history must access the master branch. If the master branch is remote, then all such operations must go across the network.
- Since there is no local repository, it is not possible to commit locally. All commits are applied to the master branch.

A lightweight checkout can be a suitable option if the branch history is huge, but you have very fast and reliable remote access. Another possible use occurs when you want to get just the working tree of a project quickly and you don't intend to use operations that involve the revision history.

Throughout the examples using lightweight checkouts, we will use a shared repository at `/sandbox/light` and the branch `/sandbox/light/sample` created with the following commands:

```
$ bzr init-repository /sandbox/light
Shared repository with trees (format: 2a)
Location:
  shared repository: /sandbox/light
$ cd /sandbox/light
$ bzr branch lp:~bzrbook/bzrbook-examples/hello-start hello
Branched 6 revisions.
```

Creating a lightweight checkout

To create a lightweight checkout, use the `bzr checkout` command with the `--lightweight` flag:

```
$ bzr checkout --lightweight hello light123
$ bzr info light123
Lightweight checkout (format: 2a)
Location:
  light checkout root: light123
   checkout of branch: hello
    shared repository: .

Related branches:
  parent branch: bzr+ssh://bazaar.launchpad.net/~bzrbook/bzrbook-
examples/hello-start/
```

As usual, the first line of the output of `bzr info` tells the type of the branch; in this case, `Lightweight checkout`.

You can shorten the rather long `--lightweight` flag as simply `--light`.

To create a lightweight checkout using Bazaar Explorer, click on the large **Start** button in the toolbar and select **Checkout...**, or navigate to **Bazaar | Start** and select the **Checkout...** option.

Converting a checkout to a lightweight checkout

You can convert a regular checkout to a lightweight checkout by using the `bzr reconfigure` command:

```
$ bzr checkout hello/ checkout123
$ bzr reconfigure --lightweight-checkout checkout123/
$ bzr info checkout123/
Lightweight checkout (format: 2a)
Location:
  light checkout root: checkout123
   checkout of branch: hello
    shared repository: .
```

You can shorten the rather long `--lightweight-checkout` flag as simply `--lightweight` or even `--light`.

Since a lightweight checkout doesn't have a repository, the reconfiguration implies that the local repository of the original checkout must be destroyed. If this cannot be done safely, for example, because there are local commits that have not been applied to the master branch yet, then the reconfiguration will fail, and Bazaar will warn you that you must synchronize first. For example:

```
$ bzr checkout -rlast:2 hello/ hello-2
$ cd hello-2/
$ bzr reconfigure --light
bzr: ERROR: '/sandbox/light/hello-2/' is not in sync with /sandbox/light/
hello/.  See bzr help sync-for-reconfigure.
```

As the output says, see `bzr help sync-for-reconfigure` for more details.

Converting a branch to a lightweight checkout

When converting a branch or a tree configuration to a lightweight checkout, you may need to specify the location of the master branch to bind to by using the `--bind-to` option. For example:

```
$ bzr branch hello branch123
Branched 6 revisions.
$ cd branch123/
$ bzr reconfigure --light --bind-to ../hello
```

Without the `--bind-to` option, Bazaar will try to re-use a previously saved bound location, push location or parent location, in that order.

Converting from a lightweight checkout

You can convert a lightweight checkout to other configuration types by using the `reconfigure` command with the appropriate flag:

- `--checkout`: This converts to a bound branch with a working tree
- `--tree`: This converts to an unbound branch with a working tree
- `--branch`: This converts to an unbound branch without a working tree

Since all the preceding configuration types use a local repository, the reconfigure operation will create a new local repository and copy all the revisions from the master branch. Keep in mind that this may take some time, especially if the master branch is remote.

Re-using a working tree

Very often, it can be useful to re-use a working tree to work on multiple branches.

- If the working tree is very large, it can be a waste of disk space and difficult to have multiple working trees at the same time
- If the project requires complicated configuration per working tree, then it can be troublesome and inefficient to repeat the setup procedure for every branch

Re-using a working tree is a matter of organizing your local branches in a certain way:

- Configure a shared repository to not create working trees by default
- Create only a single branch with a working tree, in other words a checkout, keeping all other branches tree-less

- Switch the associated branch of the checkout by using `bzr bind` followed by `bzr update` and `bzr revert`, or by using `bzr switch`
- Use a lightweight checkout for the working tree for extra safety

Setting up the example

Let's create a shared repository with no working trees by default:

```
$ bzr init-repo /sandbox/reusing --no-trees
Shared repository (format: 2a)
Location:
  shared repository: /sandbox/reusing
```

Let's create a few sample branches:

```
$ cd /sandbox/reusing/
$ bzr branch lp:~bzrbook/bzrbook-examples/common-two-features trunk
Branched 3 revisions.
$ bzr branch -r1.2.3 trunk/ feature1
Branched 4 revisions.
$ bzr branch -r1.1.2 trunk/ bug1
Branched 3 revisions.
```

Since the shared repository is configured to create no trees, we can confirm that all these branches have no working trees:

```
$ ls *
bug1:

feature1:

trunk:
```

Finally, we need to create a checkout with a working tree that we will use to switch between branches and perform version control operations:

```
$ bzr checkout trunk/ work
```

As we can confirm, a checkout always has a working tree:

```
$ ls work/
hello.py    hello.rb    hello.sh    screenshots todo.txt
```

Preparing to switch branches

There is one very important precaution to take before switching branches—make sure you don't have uncommitted changes in the working tree. As the whole point of switching branches is overwriting the working tree, you should make sure to commit all the important changes before going ahead with the switch. Although the methods explained here do not destroy the uncommitted changes by default, they can get mingled with other changes in the process, making them extremely difficult to recover. A wise thing to do is to commit all the pending changes before switching branches.

Switching to another branch using core commands

The working tree we created is currently bound to the trunk:

```
$ cd work
$ bzr info
Repository checkout (format: 2a)
Location:
  repository checkout root: .
        checkout of branch: /sandbox/reusing/trunk
         shared repository: /sandbox/reusing
```

We can switch to another branch by using `bzr bind` followed by `bzr update`, followed by `bzr revert`:

```
$ bzr bind ../bug1/
$ bzr info
Repository checkout (format: 2a)
Location:
  repository checkout root: .
        checkout of branch: /sandbox/reusing/bug1
         shared repository: /sandbox/reusing
```

There is no output after `bzr bind`, because the command doesn't change the working tree. When binding to a new location, the working tree must be updated by using `bzr update` to synchronize it with the repository.

```
$ bzr update
All changes applied successfully.
```

```
Updated to revision 3 of branch /sandbox/reusing/bug1
Your local commits will now show as pending merges with 'bzr status', and
can be committed with 'bzr commit'.
```

The way update works is, if the original branch contains revisions that are not in the new target branch, then those revisions will be treated as local commits.

If there had been "real" local commits before this step, they would still be local commits, naturally following the regular commits that have been converted to local commits. However, notice the danger here—these real local commits exist only in this branch; by definition, they have not been committed in other branches. The next step in completing the switch is `bzr revert`, which will erase these local commits. In a realistic situation, you should not have any local commits when switching between branches, as that would risk losing work.

In our example, we are not working on local commits; we simply want to switch our working tree to another branch. So, at this point, we want to move these changes out of the way with `bzr revert`:

```
$ bzr revert
-   hello.rb
 M  todo.txt
$ bzr status
unknown:
  hello.rb
```

Although there is an unknown file left behind, because it existed in the previous branch, this is not a problem. In any case, we should move it out of the way, otherwise it might lead to conflicts when we switch to another branch later. We can remove the file manually, or by using the `bzr clean-tree` command:

```
$ bzr clean-tree
hello.rb
Are you sure you wish to delete these? ([y]es, [n]o): yes
deleting paths:
  hello.rb
```

As usual, Bazaar does not make irreversible changes without asking for confirmation. To skip the confirmation, you can use the `--force` flag.

With this last move, we have successfully switched the working tree to another branch.

Switching to another branch by using switch

The `switch` command is not a core feature, but is included in the **loom** plugin. It makes switching between branches much easier:

```
$ bzr info
Repository checkout (format: 2a)
Location:
  repository checkout root: .
        checkout of branch: /sandbox/reusing/trunk
         shared repository: /sandbox/reusing

$ bzr switch ../feature1/
Updated to revision 4.
Switched to branch: /sandbox/reusing/feature1/
$ bzr info
Repository checkout (format: 2a)
Location:
  repository checkout root: .
        checkout of branch: /sandbox/reusing/feature1
         shared repository: /sandbox/reusing
```

The `switch` command hides most of the details involved in the process of switching branches.

An interesting difference from using the `bind-update-revert` method is that `bzr switch` will refuse to run if there are local commits in the branch. This is a good thing, because it is potentially unsafe to switch branches when there are local commits, as those commits would get lost. To switch anyway throwing away local commits, use the `--force` flag.

A very useful feature of the `switch` command is to create a new branch from the current one and switch to it immediately by using the `-b` or --`create-branch` option. For example:

```
$ bzr switch -b ../feature2
Tree is up to date at revision 3.
Switched to branch: /sandbox/reusing/feature2/
```

Using a lightweight checkout for switching branches

When using a shared repository, it may seem pointless at first to use a lightweight checkout locally—branches are already very cheap, as the revisions are not stored directly inside the branches but in the shared repository. Whether a checkout is lightweight or not, it makes no difference in terms of disk space, and since we are performing branch operations locally, there is also no network latency.

However, there is still a benefit of using a lightweight checkout instead of a regular one—by definition there cannot be local commits. Considering that you may inadvertently delete important local commits, there exists some amount of risk when using regular checkouts. By using lightweight checkouts, you can effectively eliminate this risk when switching branches.

You can convert your working tree by using `bzr reconfigure`. For example:

```
$ bzr reconfigure --lightweight-checkout
$ bzr info
Lightweight checkout (format: 2a)
Location:
  light checkout root: .
   checkout of branch: /sandbox/reusing/feature2
    shared repository: /sandbox/reusing
```

Using stacked branches

The concept of stacked branches is a space-saving technique, which allows multiple branches to re-use a common repository to access common revisions. A branch can be stacked on another branch, which is called the **stacked-on branch**. The stacked branch will store only the new revisions that are added to it directly, and it will re-use the repository of the stacked-on branch, whenever it needs to access the older revisions.

This setup removes the limitation of shared repositories that all the branches must be created within the same directory tree. A branch can be stacked on any other branch to re-use its repository, as long as the branch and the repository are at a location that can be accessed in read-only mode.

In practice, stacked branches make it possible to designate a read-only master branch, and let team members create their own branches at any other location in an efficient way, without unnecessary duplication of common revisions. Creating new branches stacked on the master branch is fast and efficient, because the revision history does not need to be copied.

Stacked branches are most useful in a server environment, where multiple users need an efficient way to share revision data, without the limitation of creating all branches within the same shared repository. Such an advanced setup is beyond the scope of this book; we only explain the basics of the concept because it is used on Launchpad, and occasionally you may see it mentioned in the output of certain commands. For example:

```
$ bzr push lp:~bzrbook/bzrbook-examples/dinner-party-new
Using default stacking branch /+branch-id/707963 at chroot-64745232:///~bzrbook/bzrbook-examples/
Created new stacked branch referring to /+branch-id/707963.
```

When working with multiple branches of a project in your local environment, it is easiest to use a shared repository. However, if you ever need to create branches outside the shared repository, then stacked branches can be useful to speed up branch operations and avoid wasting disk space.

For more information on using stacked branches, refer to the following URL:

`http://doc.bazaar.canonical.com/development/en/user-guide/stacked.html`

Signing revisions using GnuPG

By using cryptographic signing of commits, it is possible to verify the true identity of the committer. Revisions can be signed automatically at the time they are committed, or later manually. Signed commits are verified automatically when viewing the logs, or can be verified manually.

There are a few things to prepare in order to use signatures with Bazaar:

- Your digital signature key for signing
- The GnuPG tool to work with signatures
- The `gpgme` Python module for working with GnuPG
- The Bazaar configuration to use signatures with Bazaar commands

Getting a digital signature key for signing is beyond the scope of this book. Please refer to the following article for more information:

https://help.launchpad.net/YourAccount/ImportingYourPGPKey

GnuPG stands for **GNU Privacy Guard**. It is a free software alternative to the PGP suite of cryptographic software.
For more information, see the project's homepage at http://www.gnupg.org/.

Configuring the signing key used by Bazaar

By default, Bazaar uses the signing key that matches your identity as configured by the `bzr whoami` command or the `email` configuration in your `~/.bazaar/bazaar.conf` file. To use a different signing key, add a configuration entry as follows:

```
gpg_signing_key = 12345678
```

You can add this configuration either in `~/.bazaar/branch.conf` to be effective globally in all your projects, or in the `.bzr/branch/branch.conf` file of a branch to limit its use within that branch.

The value of the signing key comes from the `pub` line in the output of `gpg --list-keys`. For example:

```
$ gpg --list-keys
/home/janos/.gnupg/pubring.gpg
------------------------------
pub   2048R/12345678 2012-06-24
uid                  Janos Gyerik <janos@example.com>
sub   2048R/23456789 2012-06-24
```

Setting up a sample repository

Let's create a new shared repository to test the signing revisions:

```
$ bzr init-repo /sandbox/signing
Shared repository with trees (format: 2a)
Location:
  shared repository: /sandbox/signing
$ cd /sandbox/signing
```

Next, let's grab a sample branch with several committers:

```
$ bzr branch lp:~bzrbook/bzrbook-examples/unsigned --standalone --no-tree
Branched 3 revisions.
```

Verifying signatures

Verifying signatures will help in our examples to understand first how to verify signatures by using the `bzr verify-signatures` command:

```
$ bzr verify-signatures unsigned/
0 commits with valid signatures
0 commits with key now expired
0 commits with unknown keys
0 commits not valid
3 commits not signed
```

Since we didn't specify the revisions, this verified all the commits in the branch. You can specify revisions by using the `-r` flag as usual, for example, to verify only the latest revision:

```
$ bzr verify-signatures unsigned/ -rlast:1
0 commits with valid signatures
0 commits with key now expired
0 commits with unknown keys
0 commits not valid
1 commit not signed
```

As the output suggests, in addition to checking whether a commit is signed or not signed, the command also checks for expiration, validity, and whether the key has been imported into your key ring or not.

Signing existing revisions

First, let's create a test branch to work on:

```
$ bzr branch unsigned/ signed
Branched 3 revisions.
```

You can sign the existing revisions by using the `bzr sign-my-commits` command:

```
$ bzr sign-my-commits signed/
Signed 0 revisions
```

As the name of the command suggests, by default, it signs only the revisions committed by you; that is, revisions that match the value of your `email` configuration or the output of the `bzr whoami` command. To sign the revisions by other committers, you must specify the name of the committer as it appears in `bzr log`. For example:

```
$ bzr log signed/ | grep committer
committer: Anna <anna@example.com>
committer: mike@example.com
committer: jack@example.com
$ bzr sign-my-commits signed/ 'Anna <anna@example.com>'
anna@example.com-20121106205046-mlbm5jq4abxkyqr7

You need a passphrase to unlock the secret key for
user: "Janos Gyerik <janos@example.com>"
2048-bit RSA key, ID 12345678, created 2012-06-24

Signed 1 revisions
```

The preceding steps sign all the commits whose committer information matches precisely the one given on the command line. In this step, you must enter the passphrase of your signing key, unless you have already stored in memory by using `gpg-agent` or a similar key manager. You can confirm that the commit is now signed correctly by re-running the `bzr verify-signatures` command:

```
$ bzr verify-signatures signed/ -rlast:1
All commits signed with verifiable keys
```

To see more details about the signature, take a look at the revision by using `bzr log`, and specify the `--signatures` flag:

```
$ bzr log signed/ --signatures -rlast:1
------------------------------------------------------------
revno: 3
committer: Anna <anna@example.com>
branch nick: unsigned
timestamp: Tue 2012-11-06 21:50:46 +0100
signature: valid signature from Janos Gyerik <janos@example.com>
message:
  added shell implementation
```

Bazaar Explorer also shows the signature details when viewing the revision logs.

`bzr sign-my-commits` has some limitations:

- It cannot sign commits of specific revisions; only of specific committers
- It cannot sign commits that already have a signature

Signing a range of commits

There is a hidden command `bzr re-sign`, which can be used to sign a range of commits or commits that already have a signature:

```
$ bzr re-sign -rlast:2..last:1 -d signed/

You need a passphrase to unlock the secret key for
user: "Janos Gyerik <janos@example.com>"
2048-bit RSA key, ID 12345678, created 2012-06-24


You need a passphrase to unlock the secret key for
user: "Janos Gyerik <janos@example.com>"
2048-bit RSA key, ID 12345678, created 2012-06-24
```

Although this works, you must enter your passphrase for each revision to sign.

Signing new commits automatically

In order to sign all your new commits automatically, you need to add the following configuration:

```
create_signatures = always
```

You can either add this configuration in the `[DEFAULT]` section of the global configuration file `~/.bazaar/bazaar.conf`, or in a branch configuration file `.bzr/branch/branch.conf`. An easy way to set or reset this configuration is by using the `bzr config` command.

Use the following command to set and reset the configuration in the current branch:

```
$ bzr config create_signatures=always
$ bzr config create_signatures --remove
```

Use the following command to set and reset the configuration globally for all your commits:

```
$ bzr config create_signatures=always --scope=bazaar
$ bzr config create_signatures --remove --scope=bazaar
```

The only currently supported value for the configuration is `always`; other possible values may be added in the future. For more details, see the `create_signatures` section in `bzr help configuration`.

When this configuration is enabled, commit operations can only succeed after the revision is signed. If the signing fails for some reason, for example, if the entered passphrase is incorrect, then the commit itself will fail too:

```
$ bzr init temp
Created a repository branch (format: 2a)
Using shared repository: /sandbox/signing/
$ cd temp/

$ date > date.txt
$ bzr add
adding date.txt
$ bzr commit -m 'just a test'
Committing to: /sandbox/signing/temp/
added date.txt

You need a passphrase to unlock the secret key for
user: "Janos Gyerik <janos@example.com>"
2048-bit RSA key, ID 12345678, created 2012-06-24

gpg: gpg-agent is not available in this session
Enter passphrase:
gpg: Interrupt caught ... exiting
bzr: interrupted
$ bzr status
added:
  date.txt
```

Configuring a hook to send an e-mail on commit

Hooks provide a great way to customize the behavior of Bazaar. They can be programmed to perform some action before or after certain Bazaar operations.

We will cover the details of hooks in *Chapter 10*, *Programming Bazaar*. Here, we only explain how to use a very commonly used hook to send an e-mail report on every commit. Such e-mail reports are very practical in a team, to let all the team members know of the latest changes, and to facilitate the good practice of peer reviews within the team.

In Bazaar, hooks are implemented as plugins. There is a list of defined Bazaar operations named **hook points** that we can hook into and perform some action. To send e-mails with the summary of changes, we will use the **email** plugin, triggered by the `post_commit` hook point.

Setting up the example

For configuring and testing the e-mail sending, we just need a very simple repository with a few files for making simple changes and dummy commits. You can create a new repository with `bzr init` and make some random commits in it, or grab the following sample repository:

```
$ bzr branch lp:~bzrbook/bzrbook-examples/common-two-features /tmp/
emailing
Branched 3 revisions.
```

Installing the email plugin

Depending upon your operating system and mode of installation, the email plugin may already be installed. We can confirm this by looking at the list of plugins:

```
$ bzr plugins | grep email
email
  Sending emails for commits and branch changes.
```

If the plugin is not installed, try to get it by using the package manager of your operating system. Alternatively, it is quite easy to install from source:

```
$ bzr branch lp:bzr-email
$ cd bzr-email
$ python setup.py install --user
```

Enabling commit emails

The email plugin is pre-configured to respond to two hook points:

- `post_commit`: This hook point is triggered on every commit
- `post_change_branch_tip`: This hook point is triggered by push and pull operations

By default, the email plugin doesn't do anything. To enable sending e-mail messages, you must set the configuration `post_commit_to` to an e-mail address. Although you can add this to your global configuration file `~/.bazaar/bazaar.conf`, it probably makes more sense to add it in each branch for which you want to receive e-mail notifications. Let's try this in our sample branch:

```
$ bzr config post_commit_to=janos@example.com
```

Testing the configuration

Once `post_commit_to` is set to an e-mail address, all commits will trigger an e-mail report. Let's make some changes and commit:

```
$ bzr rm hello.py
deleted hello.py
$ bzr commit -m 'deleted a file'
Committing to: /tmp/emailing/
deleted hello.py
Committed revision 4.
```

If you check your e-mails, you should receive an e-mail with the subject as `Rev 4: deleted a file in file:///sandbox/emailing/`. That is, the subject includes the revision number, the message of the commit, and the URL of the repository. The message body includes more details, such as the summary of changes, and the diffs of content changes in plaintext files.

```
At file:///sandbox/emailing/


------------------------------------------------------------
revno: 4
revision-id: janos@example.com-20121108063451-e3ch135atoehp6x9
parent: janos@example.com-20120805100525-n8pdfboqgjji3kui
committer: Janos Gyerik <janos@example.com>
branch nick: emailing
```

```
timestamp: Wed 2012-11-07 22:34:51 -0800
message:
  deleted a file
=== removed file 'hello.py'
--- a/hello.py  2012-08-05 09:59:39 +0000
+++ b/hello.py  1970-01-01 00:00:00 +0000
@@ -1,3 +0,0 @@
-#!/usr/bin/env python
-
-print 'hello world!'
```

Customizing the plugin

The email plugin can be customized by setting more branch configuration options.

The branch URL used in the subject and message body of the e-mail is the `public_branch` URL or the path of the branch in the `file:///` format. If the branch has a public URL, you should make sure to configure it. For example:

```
$ bzr config public_branch=https://repos.example.com/project1
```

By default, the plugin sends e-mails only on commits, not on push and pull operations. To enable e-mails on push and pull too, set the `post_commit_push_pull` option:

```
$ bzr config post_commit_push_pull=1
```

Finally, you may also want to customize the sender address. By default, it is the same as defined with `bzr whoami`, but you can override it with the `post_commit_sender` option:

```
$ bzr config post_commit_sender='Janos <janos@example.com>'
```

If you need even more customization, then you might want to write your own hook. See *Chapter 10*, *Programming Bazaar*, for more details.

Summary

In this chapter, we have covered a few interesting advanced features of Bazaar. Although these features are not essential in most cases and might not apply for everyone, they can help you work more efficiently and enrich your workflows:

- Shelving changes helps you keep your commits clean
- Aliases help you do things faster
- Lightweight checkouts are a fast way to get the tip of a repository without downloading the entire history
- Re-using a working tree is useful in most projects, especially in projects with lots of local environment-specific configuration files
- Stacked branches are a space-saving solution for advanced repository layouts in a server environment
- Signing commits using digital signatures makes it possible to verify the identity of committers
- Committing e-mails is useful to track the progress of a project and to facilitate code reviews

The next chapter will explain how Bazaar can work together with other version control systems such as Subversion and Git.

9

Using Bazaar Together with Other VCS

Bazaar is very flexible and can work well with other version control systems almost completely transparently. In this way, even if your team is using a different VCS, you may still have the option to use Bazaar and take advantage of its unique features.

In this chapter, we will explain how to use Bazaar to interact with other version control systems, along with important practical tips and limitations that are good to be aware of. We will cover, in detail, how to work with Subversion and Git directly. As an alternative solution, we will explain fast-import, an indirect method that can work with any VCS, at least in theory.

The following topics will be covered in this chapter:

- Working with other VCS in general
- Using Bazaar with Subversion
- Using Bazaar with Git
- Migrating between VCS using fast-import

Working with other VCS in general

Bazaar can interact with other version control systems through plugins. An appropriate plugin can intercept version control operations between Bazaar and the foreign system, and perform the necessary translation between the two protocols. In this way, in theory, Bazaar can work with any other VCS.

Throughout this chapter, we will refer to other version control systems as **foreign repositories**, and branches or branch-like concepts in such systems as **foreign branches**. In this section, we will cover the important general subjects when working with foreign repositories and branches, such as how to install the required plugins, and common good practices, limitations, and known issues.

Working with foreign branches

In order to interact with foreign branches, the plugin must intercept all the operations between Bazaar and the foreign branch, and perform the necessary translation between the different protocols. This translation is an overhead, and inevitably results in some slowness, especially in operations that potentially fetch large number of revisions, such as branch, checkout, push, pull, and merge.

A very important key point to understand is that when creating a local branch by using `bzr branch` or `bzr checkout` from a foreign branch, the new local branch will be a native Bazaar branch. You can use the branch in your local workflows in the same way as any of your other local native Bazaar branches; all your operations will be native Bazaar operations.

Another key point is that in order to ensure integrity in interactions with multiple foreign branches, Bazaar identifies unique revisions of the foreign repository even across multiple branches, and it preserves all the necessary metadata to avoid loss of information.

Installing plugins

Depending upon your operating system, the plugins required to work with foreign branches may be installed by default, or you may have to install them as separate packages.

The relevant plugins used in this chapter are as follows:

- `bzr-git`: This provides support to work with Git branches
- `bzr-svn`: This provides support to work with Subversion branches
- `bzr-fastimport`: This provides support to import and export VCS-agnostic version control data

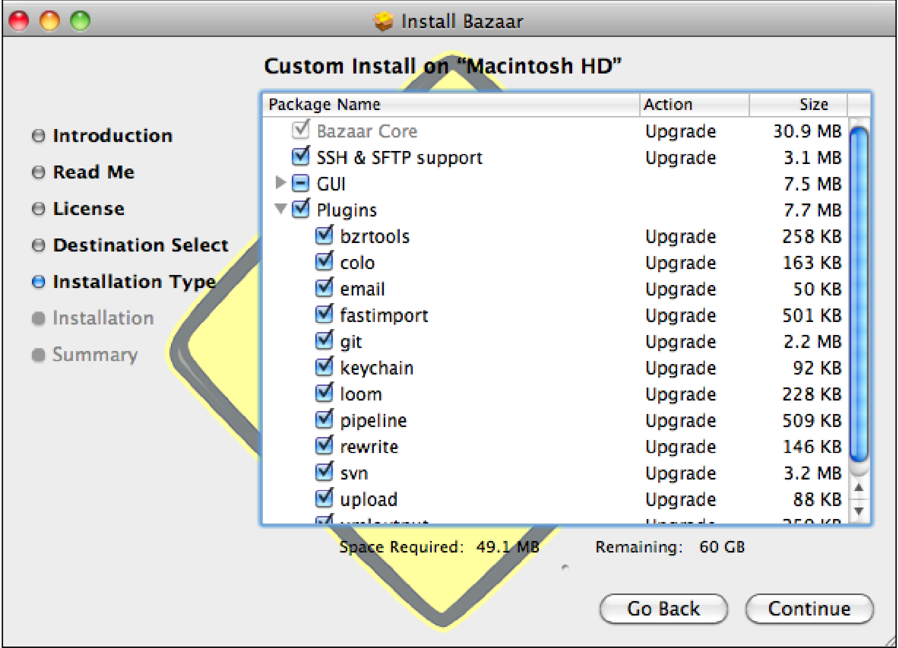
You can confirm whether these plugins are already installed by using the `bzr plugins` command:

```
$ bzr plugins
fastimport 0.14.0dev
  FastImport Plugin

git 0.6.7
  A GIT branch and repository format implementation for bzr.

svn 1.2.1
  Support for Subversion branches
```

For basic information about each plugin, you can use the `bzr help` command by specifying the name of the plugin. For example, `bzr help svn`.

Installing plugins in Windows or Mac OS X

In Windows and Mac OS X, the installer includes the plugins to work with Subversion and Git by default. If you deselected them during installation, then you can install them later by running the installer again and selecting the plugins you need. For example in Mac OS X:



Installing plugins in Linux

In Linux, all Bazaar plugins are in separate packages; you can install them by using the software management tool of your distribution. The package names of the plugins are typically prefixed with `bzr-`; for example, `bzr-svn`, `bzr-git`, and `bzr-fastimport`.

Installing plugins using Pip

Installing plugins using Pip, the Python package manager tool, can be a good option if you don't have administrator access or if you prefer to isolate the plugins in a **virtualenv** environment. The package names of Bazaar plugins are typically prefixed with `bzr-`; for example, `bzr-svn`, `bzr-git`, and `bzr-fastimport`. Install the plugins you need by using `pip install`. For example:

```
$ pip install bzr-svn
```

Installing additional requirements

Some plugins may have additional requirements; for example, `bzr-svn` requires the `subvertpy` Python package, and `bzr-fastimport` requires `fastimport`. These requirements are mentioned in the relevant plugin-specific sections. To install these requirements, follow the same steps as when installing the plugins themselves.

Understanding the protocol overhead

Keep in mind that when working with foreign branches, Bazaar must translate between the remote protocol and its own native format. For this reason, all the operations involving foreign branches are slower as compared to the same operations with native Bazaar branches.

Bazaar is a distributed VCS, therefore when branching from a remote branch, it downloads the full revision history, even if the remote branch is not a distributed VCS. Considering that the revision information must be translated to the native Bazaar format, this can take a very long time.

Using shared repositories

When working with multiple remote branches, using a shared repository helps reducing the overhead of network transfers, as the common revisions in the remote branches only need to be downloaded once.

Similarly, in the case of fetching foreign branches, using a shared repository helps reducing the overhead of translation between Bazaar and the foreign protocol, as the common revisions in the foreign branches only need to be translated into Bazaar's native format once. Thus, downloading a second branch from the same foreign repository should be significantly faster than the first.

Limitations

Although a plugin can do a great job at translating the core version control operations, some features may not be possible to implement perfectly due to fundamental architectural differences between Bazaar and the foreign system. This is an inevitable limitation; you should always keep this in mind when interacting with another VCS through Bazaar.

Before you begin to use Bazaar with a foreign repository, make sure to read the *Limitations* section of the plugin documentation carefully, in order to understand the key differences. It also helps to have a good working knowledge of the given foreign repository.

Issues and crashes

Working with foreign branches is very difficult, and unfortunately sometimes even the most basic operations, such as the initial branching or checkout, may fail. In this case, Bazaar dumps a stack trace, which you can use to report it as a bug, but, of course, that's little consolation when you need to work on something immediately.

A possible last resort in such cases may be using fast-import to convert the version control information of the foreign branch into Bazaar's native format. We cover fast-import later in this chapter. However, this method is designed for migrating repositories from one version control system to another, and thus is really not the same thing as performing Bazaar operations on foreign branches.

Using Bazaar with Subversion

Bazaar works transparently with Subversion—the `checkout`, `update`, `commit`, `branch`, `pull`, `push`, and `merge` commands, all work directly with Subversion repositories. There are several advantages in working with Subversion through Bazaar:

- **The benefit of being distributed is that branches carry the complete revision history**: Since Subversion is a centralized VCS; a checkout contains only the latest revision, not the full history. Operations that involve the history require access to the central repository, making these operations slower, or even unusable when the server is down. If you check out from Subversion by using Bazaar, you will have the full history locally.
- **You can create local branches**: In Subversion, all branches must be created and stored on a central server. If you checkout using Bazaar, you can create local branches and enjoy more freedom to experiment.
- **Merging branches is easier**: Bazaar is very good at merging branches, and you also have the option to try different merging strategies.
- **Backup, mirroring, migration**: Since a checkout using Bazaar will have the complete history, it is close to a backup of the original Subversion location.

Installing bzr-svn

The plugin to work with Subversion branches is named `svn`, typically in a package named `bzr-svn`. Confirm whether or not it is already installed in your system by using `bzr plugins`. If it is not in the list, see the *Installing plugins* section, and follow the steps to install it.

In addition to `bzr-svn`, you also need to install `subvertpy`, a Python library to interface with Subversion. Install it using your operating system's package manager or `pip`. A simple way to verify that both `bzr-svn` and `subvertpy` are correctly installed is by creating a dummy Subversion repository using Bazaar:

```
$ bzr init --subversion /tmp/empty-svn-repo
Initialising Subversion metadata cache in /Users/janos/.bazaar/svn-cache/2f4f2ffe-08e1-45a5-847b-c4ce217833be.
Created a repository branch (format: subversion)
Using shared repository: /tmp/empty-svn-repo/
```

The `--subversion` flag is provided by the `bzr-svn` plugin. Normally, the `bzr init` command does not have such options.

Supported protocols and URL schemes

Bazaar supports all the standard Subversion URL formats, such as the following:

- `http://server/path/to/branch`
- `https://server/path/to/branch`
- `svn://server/path/to/branch`
- `svn+ssh://user@server/path/to/branch`
- `file:///path/to/repo/branch`

Although the generic URL formats, such as `http://`, `https://`, or `file:///`, can be Bazaar or even other VCS, Bazaar correctly detects Subversion repositories.

Note that when using the `svn+ssh://` protocol, the path to the Subversion repository must be the full absolute path, even if the repository is in the home directory of the user. And, unlike with `bzr+ssh://`, the `/~/` notation to indicate the user's home directory does not work; you must use the full path always.

Using the example Subversion repository

The best way to explore how Bazaar works with Subversion branches is by playing with a local Subversion repository. If you are familiar with Subversion, you can create a repository by using `svnadmin`, or you can download and unpack the following sample repository:

`https://launchpad.net/bzrbook-examples/trunk/examples/+download/svn-repo.tar.gz`

The `svn-repo` directory inside the ZIP file contains a small Subversion repository, which will be accessible via `file:///` URLs. For example, if you move the `svn-repo` directory into `/tmp`, it will be accessible via `file:///tmp/svn-repo`. You can try some other commands, such as `bzr info` or `bzr log`, on it:

```
$ bzr info file:///tmp/svn-repo
Repository branch (format: subversion)
Location:
  shared repository: /tmp/svn-repo
  repository branch: /tmp/svn-repo
$ bzr log file:///tmp/svn-repo/trunk --line -n0
4: gatekeeper 2012-12-02 [merge] merged from anna
```

```
  3.1.3: anna 2012-12-02 added r impl
  3.1.2: anna 2012-12-02 added python impl
  3.1.1: janos 2012-12-02 created branch for anna
3: janos 2012-11-26 merged from mike
2: janos 2012-11-26 added readme
1: janos 2012-11-26 created standard svn layout
```

The examples in this section will use this repository, assuming it at the path `/sandbox/svn-repo`.

Understanding branches in Subversion

Before we begin to branch or check out from Subversion, it is good to be aware of some important differences between how branches and tags work in Subversion.

A typical Subversion repository contains the following subdirectories at the repository root:

- `trunk`: This is the master branch, where the main development happens
- `branches`: This is a container directory for branches
- `tags`: This is a container directory for tags

In Subversion, a **branch** is a lightweight copy of another directory at some specific revision. Typically, it is created in the `branches` directory, and it behaves like a directory. Tags are created in exactly the same way, that is, as lightweight copies of specific directories at specific revisions.

Although internally Subversion stores branches and tags efficiently, when you checkout the repository, the layout is expanded into the filesystem as regular directories, which can take a lot of disk space. In order to avoid expanding a very large directory tree with all the branches, make sure to use the right Subversion URL in the checkout and branch operations.

Keep in mind that the preceding layout is a commonly used standard, not a hard rule. It is up to the managers of the project how they organize branches and tags inside their Subversion repository. Make sure to confirm the right URL before you begin to checkout or branch.

Another important point to note is that although in Bazaar the smallest logical unit that you can checkout is a branch, in Subversion it is a subdirectory.

Branching or checkout from Subversion

You can branch or checkout from Subversion with the `bzr branch` and `bzr checkout` commands, or by using Bazaar Explorer as usual. But first let's create a shared repository:

```
$ bzr init-repo /sandbox/svn-examples
Shared repository with trees (format: 2a)
Location:
  shared repository: svn-examples
$ cd /sandbox/svn-examples/
```

It is always a good idea to use a shared repository when working with multiple remote branches. This is especially true when working with foreign branches, due to the added overhead of translation between Bazaar and the foreign protocol.

Next, let's checkout a few branches by using `bzr checkout`:

```
$ bzr checkout file:///sandbox/svn-repo/trunk
Initialising Subversion metadata cache in /Users/janos/.bazaar/svn-cache/5260eff9-c761-4657-805e-432c3fbc2731.
$ bzr checkout file:///sandbox/svn-repo/branches/mike
```

Notice the message after the first checkout—`Initializing Subversion metadata cache`. This is an optimization by Bazaar—the Subversion plugin builds a cache of Subversion metadata in order to speed up operations when working with the same Subversion repository again in the future. A new directory is created to store the cache of each distinct Subversion repository you use, and the name of the directory corresponds to the UUID of the Subversion repository, a universally unique ID generated by Subversion when the repository was created.

Branching from Subversion works exactly in the same way as with Bazaar branches. Regardless of the manner in which you fetch a Subversion branch, the end result will be a native Bazaar branch.

Keep in mind that Bazaar branches store the complete revision history, and therefore the first checkout or branch from a Subversion repository may take a long time depending upon the size of the project and the number of commits.

Preserving Subversion metadata

When fetching a Subversion location, Bazaar preserves various metadata about the revisions, such as the following:

- Basic revision information—committer, timestamp, and log message
- Original revision number in Subversion
- Versioned properties, such as `svn:mergeinfo` and `svn:mime-type`
- Revision IDs and file IDs

Preserving original revision numbers

Revision numbers in Subversion are not per-branch but per-repository. If you recall that branches in Subversion behave much like directories, this makes sense.

Although both Bazaar and Subversion increments the revision number on each commit, when fetching a subdirectory of a Subversion repository, Bazaar downloads only the revisions that affected the specified path, which is only a subset of all the revisions inside the entire Subversion repository. As a result, the revision numbers in the Bazaar branch will not be the same as in the original Subversion repository. We can confirm this by using `bzr log`:

```
$ bzr log -r3 trunk/
------------------------------------------------------------
revno: 3
svn revno: 6 (on /trunk)
committer: janos
timestamp: Mon 2012-11-26 20:50:53 +0000
message:
  merged from mike
```

The Bazaar revision number is `3`, but the Subversion revision number is `6`, because only three of the commits in Subversion affected `/trunk`.

You can reference the past revisions by their original Subversion revision number by prefixing with `svn:`. For example:

```
$ bzr log -r svn:6 trunk/
------------------------------------------------------------
revno: 3
svn revno: 6 (on /trunk)
committer: janos
timestamp: Mon 2012-11-26 20:50:53 +0000
message:
  merged from mike
```

Preserving versioned properties

Versioned properties are a feature of Subversion that is not supported by Bazaar. In Subversion, these are used for storing all kinds of metadata associated with files, directories, and merging, such as mime-types, keywords, line-ending character, and various other purposes, and all these metadata are versioned. Bazaar will preserve versioned properties and write them back when pushing to a Subversion repository, but in general, they will be ignored.

Preserving revision and file IDs

Most importantly, Bazaar preserves revision and file IDs, which we can see by using the `--show-ids` flag of `bzr log`. For example:

```
$ bzr log -r3 trunk/ --show-ids
------------------------------------------------------------
revno: 3
revision-id: svn-v4:5260eff9-c761-4657-805e-432c3fbc2731:trunk:6
parent: svn-v4:5260eff9-c761-4657-805e-432c3fbc2731:trunk:2
svn revno: 6 (on /trunk)
committer: janos
timestamp: Mon 2012-11-26 20:50:53 +0000
message:
  merged from mike
```

The revision ID is derived from the Subversion repository's UUID, the path inside the Subversion repository, and the original Subversion revision number. In this way, Bazaar can uniquely identify revisions that originated from a Subversion repository, and it also keeps track of the parent-child relationships of the revisions.

The great benefit of this is that two independent Bazaar branches created from the same Subversion location will be identical:

```
$ bzr branch file:///sandbox/svn-repo/trunk tmp
Branched 4 revisions.
$ bzr missing -d trunk/ tmp/
Branches are up to date.
```

This is especially important when merging branches, as it effectively prevents from merging the same revisions twice.

Pulling or updating from Subversion

Updating a Bazaar branch from a Subversion parent branch works in the same way as it does with native Bazaar branches. Let's test this by branching from an older revision of a Subversion branch, and then pulling from it to bring the local branch up-to-date:

```
$ bzr branch -rlast:2 file:///sandbox/svn-repo/trunk feature1
Branched 3 revisions.
```

Our local Bazaar branch is out of date, precisely one revision behind, so let's bring it up-to-date by using `bzr pull`:

```
$ cd feature1
$ bzr pull
Using saved parent location: /sandbox/svn-repo/trunk/
+N  hello.r
+N  hello.tcl
All changes applied successfully.
Now on revision 4 (svn revno: 10).
```

Keep in mind that the `pull` command may rearrange the revisions in the local branch in the same way as it works with native Bazaar branches.

Similarly, if we had used `bzr checkout` to get a Subversion branch, then we can download new revisions from the Subversion server with `bzr update`, exactly as if we were working with native Bazaar branches:

```
$ bzr checkout -rlast:2 file:///sandbox/svn-repo/trunk checkout1
$ cd checkout1/
$ bzr up
+N  hello.r
+N  hello.tcl
All changes applied successfully.
Updated to revision 4 of branch /sandbox/svn-repo/trunk
```

Committing to Subversion

Committing in a branch bound to a Subversion repository works exactly in the same way as with native Bazaar branches—changes are first committed at the remote location, and only if that is successful, the changes are committed in the local Bazaar branch. If the local branch is out of date, the commit will be rejected, and you will have to update the local branch first by using `bzr update`.

When committing to a Subversion repository, Bazaar sets some metadata about the revisions to re-use later when working with those revisions again, such as the author information and other details. Bazaar stores these metadata as revision properties named with a prefix `bzr:` to make it easy to filter them out if necessary. These revision properties have no effect on Subversion operations, and are only visible in the most detailed logs. If for some reason you want to prevent Bazaar from saving such data, then you cannot use `bzr commit`; the only way is to unbind the branch, commit your changes locally only, and use the `bzr dpush` command to push them to Subversion. In this way, the extra `bzr:` properties won't be saved; the revisions will be applied on the server as if they were native Subversion commits.

> To see all the revision properties, including the ones set by Bazaar operations, you can use the Subversion command `svn log --with-all-revprops --xml`.

Pushing to Subversion

Pushing to a Subversion repository works in the same way as it does with native Bazaar branches:

```
$ bzr branch file:///sandbox/svn-repo/trunk feature2
Branched 4 revisions.
$ cd feature2
$ bzr push file:///sandbox/svn-repo/branches/feature2
Created new branch at /branches/feature2.
```

Be careful when using `bzr push`, as it can reorder revisions to match the local history layout at the remote repository location. This is especially important when the Subversion repository is publicly available. To prevent such issues, Bazaar will refuse to perform the push if that would reorder revisions.

As with committing, Bazaar saves additional metadata to Subversion when pushing revisions. If for some reason you prefer to not save such metadata, then use `bzr dpush` to push revisions instead of `bzr push`. If the pushed revisions do not contain a merge from other Subversion branches, then the end result will be exactly the same.

Merging Subversion branches

Bazaar is very good at merging Subversion branches, thanks to a few notable key factors:

- `bzr-svn` correctly tracks the relationships between Subversion revisions across different branches
- Since `bzr-svn` converts Subversion branches to native Bazaar branches, ultimately the merge operation is performed between Bazaar branches
- Bazaar is great at merging, thanks to proper tracking of rename operations, and the many user-friendly and powerful features, such as `bzr remerge`

Let's try it out by merging `/branches/jack` into `/trunk`. We already fetched the trunk with `bzr checkout` earlier; now, let's get `/branches/jack` and perform the merge:

```
$ bzr branch file:///sandbox/svn-repo/branches/jack --no-tree
Branched 6 revisions.
$ cd trunk/
$ bzr merge ../jack/
+N  hello.py
+N  hello.rb
All changes applied successfully.
```

In this example, we fetched `/branches/jack` to a local Bazaar branch before merging from it, but we could have used the remote branch URL directly. However, it is always a good idea to fetch the branch first, as in this way you can re-use the branch multiple times without redownloading it again from the source repository.

Also notice that we used the `--no-tree` flag to create the branch. In this way, we can save disk space, as we don't need the working tree of a branch we just want to merge from it.

Let's commit the merge:

```
$ bzr commit -m 'merged from jack' --author gatekeeper
Committing to: /sandbox/svn-repo/trunk
added hello.rb
added hello.tcl
Committed revision 5.
```

Since we fetched trunk with `bzr checkout`, it is bound to the Subversion repository, and as we can see in the `Committed to` line in the output, the revision is committed to the Subversion repository itself. If our trunk directory had been an unbound branch, then the merge would be committed only locally, and we could push it to Subversion with `bzr push :parent`.

When committing the merge or pushing it later to Subversion, Bazaar sets the `svn:mergeinfo` property used by Subversion to track merges. This is essential to let Subversion understand correctly that the commit was in fact a merge, and appear as if performed by using a native Subversion client. We can confirm this by using the `svn` program, the command-line interface of Subversion:

```
$ svn pget svn:mergeinfo file:///sandbox/svn-repo/trunk
/branches/anna:7-9
/branches/mike:3-5
/branches/jack:11-13
```

Indeed, the property contains the information about the merge from `/branches/jack` we performed earlier.

As mentioned in the previous sections, `bzr push` will write the additional revision properties to the Subversion repository. Although this behavior can be prevented by using `bzr dpush` instead, in that case Bazaar will also not write `svn:mergeinfo`. In this situation, you would have to add `svn:mergeinfo` manually later, by using a Subversion client.

Remember that with Bazaar, you can choose from different merge algorithms, which sometimes yield better results with fewer conflicts. The `bzr remerge` command is especially useful to try a different algorithm on selected files, and the `--reprocess` flag may help reducing the size of conflicted areas. See *Chapter 3*, *Using Branches*, on branching and merging for a detailed explanation with examples.

Merging local branches into Subversion

Although you can merge local Bazaar branches into Subversion, there is a very important difference between Bazaar and Subversion—in Bazaar, the merged revisions are preserved and propagated to the remote location when committing or pushing the merge, but in Subversion, this is not the case.

When merging a local Bazaar branch that does not exist in the Subversion repository, the merge will be just a single commit in Subversion; the unique revisions of the branch will not be copied. If you want to preserve the revisions of the local branch in Subversion, then first you have to push that branch to a suitable location in the Subversion repository.

Let's illustrate this with an example, through the following steps:

1. Create a local branch from the trunk.
2. Commit to the branch a few times.
3. Push the branch to Subversion.
4. Merge the branch to the trunk and commit it to Subversion.

Let's create a local branch from the trunk called `feature3` and do some commits in it:

```
$ bzr branch trunk feature3
Branched 4 revisions.
$ cd feature3/
$ echo >> hello.pl
$ bzr commit -m 'meaningless change'
Committing to: /sandbox/svn-examples/feature3/
modified hello.pl
Committed revision 5.
$ echo >> hello.sh
$ bzr commit -m 'another meaningless change'
Committing to: /sandbox/svn-examples/feature3/
modified hello.sh
Committed revision 6.
```

At this point, the branch exists only locally, not in Subversion. Let's push this to Subversion before we merge it into the trunk. We can push it after we merge too, but in this way, the metadata will make more sense, as you will see later:

```
$ bzr push file:///sandbox/svn-repo/branches/feature3
Created new branch at /branches/feature3.
```

Next, let's update our trunk and merge the branch into it. In our example, the updating is unnecessary, as we know for a fact that nobody else has committed to the trunk; this is just a reminder so that you don't forget to do in a real-life situation.

```
$ bzr update
Tree is up to date at revision 4 of branch /sandbox/svn-repo/trunk
$ bzr merge ../feature3/
 M  hello.pl
 M  hello.sh
All changes applied successfully.
```

Finally, let's commit the merge. Since our trunk is bound to the Subversion repository, because we created it using `bzr checkout`, the commit will be written back to Subversion immediately:

```
$ bzr commit -m 'merged from feature3 branch'
Committing to: /sandbox/svn-repo/trunk
modified hello.pl
modified hello.sh
Committed revision 5.
```

This commit, like all commits in Subversion, is a single revision including all the changes we did in our local `feature3` branch. If we hadn't pushed the `feature3` branch to Subversion earlier, the commits in that branch would not be preserved anywhere in Subversion. Since we did push it, the revisions are preserved, and Bazaar also created the necessary metadata to let Subversion know about the relationship between the trunk and the `feature3` branch we pushed, as we can confirm with `svn pget`:

```
$ svn pget svn:mergeinfo file:///sandbox/svn-repo/trunk /branches/mike:3-
5
/branches/anna:7-9
/branches/feature3:14-15
```

If you want to preserve the individual commits in your local branches, remember to push the branch to Subversion before merging it into the trunk. That way, Bazaar will record the necessary metadata correctly. If you forget to perform the operations in this order, then you would have to create that metadata yourself.

Binding and unbinding to Subversion locations

You can bind to and unbind from a Subversion repository in the same way as with native Bazaar branches. The result of `bzr checkout` is a bound branch:

```
$ bzr info trunk/
Repository checkout (format: 2a)
Location:
  repository checkout root: trunk
        checkout of branch: /sandbox/svn-repo/trunk
         shared repository: .
```

You can unbind from the Subversion location, which can be useful, for example, if the repository is temporarily inaccessible due to network problems, or if you want to commit locally only:

```
$ cd trunk/
$ bzr unbind
$ bzr info
Repository tree (format: 2a)
Location:
  shared repository: /sandbox/svn-examples
  repository branch: .
```

When the connection is restored, or you are ready to merge your local commits into Subversion, you can bind to it again:

```
$ bzr bind file:///sandbox/svn-repo/trunk
$ bzr update
Tree is up to date at revision 4 of branch /sandbox/svn-repo/trunk
```

Similarly, you can bind to a Subversion location that you branched from using bzr branch:

```
$ bzr branch file:///sandbox/svn-repo/trunk tmp
Branched 4 revisions.
$ cd tmp
$ bzr bind :parent
```

Using lightweight checkouts

An interesting feature of `bzr-svn` is that a lightweight checkout is actually a native Subversion working copy:

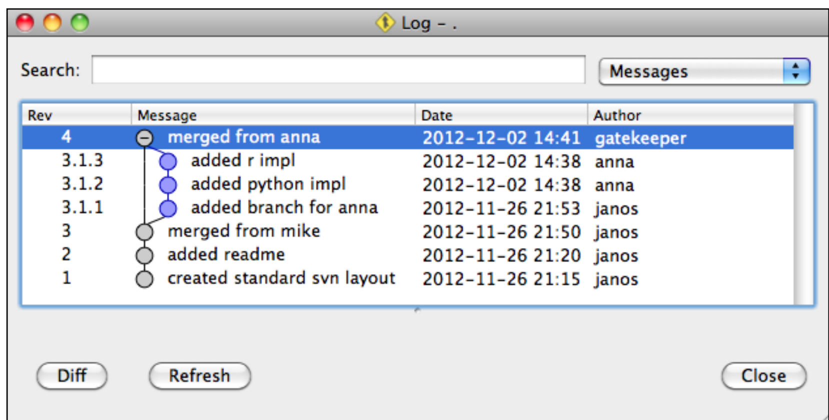
```
$ bzr checkout --lightweight file:///sandbox/svn-repo/trunk light
$ ls -a light/
.
..
.svn
README.md
hello.pl
hello.py
hello.r
hello.sh
```

As you can see, instead of a `.bzr` directory, there is a `.svn` directory. In fact, Bazaar can work with native Subversion working copies, as if using a Subversion client.

In general, the same limitations apply to lightweight checkouts as when using native Bazaar branches—all the operations that work with the revision history will be slow. In addition, due to the overhead of the conversion between protocols, these operations will be even slower than usual.

Browsing the logs

An interesting benefit of using Bazaar to merge Subversion branches is that the branch history is easier to view in the logs, thanks to the additional metadata added by Bazaar:



Notice that we can see the merged revisions from Anna's branch—merged using Bazaar, but we cannot see the merged revisions in Mike's branch—merged using Subversion.

Limitations of bzr-svn

The plugin has the following limitations:

- Creating the first branch from a Subversion repository can be extremely or even intolerably slow, depending upon the size of the repository.
- Creating a branch from a Subversion repository is sometimes not possible at all, due to bugs and crashes.
- Some Subversion properties have no effect in Bazaar—`svn:ignore`, `svn:mime-type`, `svn:eol-style`, `svn:keywords`, and `svn:externals`. These properties are ignored in general, but carried over correctly in branch operations.
- Subversion merges are not shown in Bazaar, unless the `svk:merge` property (used by SVK) is also set in addition to `svn:mergeinfo`.

For a more complete and up-to-date list of limitations, refer to the plugin's homepage:

`http://wiki.bazaar.canonical.com/ForeignBranches/Subversion`

Final remarks on bzr-svn

For the most part, Bazaar works transparently with Subversion repositories. The result of a checkout or branch operation from Subversion is a native Bazaar branch, thus all your local operations will be native Bazaar operations that you are used to. However, all the interactions with the remote Subversion repository will be inevitably slower than usual, due to the translation between the protocols of these systems.

As long as you are aware of the limitations and the differences of the two systems, you can greatly benefit from the added features of Bazaar, such as the ability to create local branches and advanced merging capabilities.

Quick tips and cheat sheet:

- Be aware of the limitations before you start using Bazaar with Subversion
- Always use a shared repository locally when working with Subversion
- Double-check the correct Subversion URL before you checkout or branch, in order to avoid downloading too many branches at once
- The first checkout or branch operation will take a long time; subsequent operations should be much faster

- Avoid operations directly on remote Subversion branches, check out or branch from them first, and use the local Bazaar branch instead
- Feel free to merge Subversion branches in Bazaar to benefit from Bazaar's advanced merging features, such as `bzr remerge` and the various merging strategies
- Remember to push local branches to Subversion if you want to preserve their revisions in the Subversion repository
- Use a lightweight checkout if you just want to view the latest version of the project without working on it

The up-to-date details and limitations of the plugin are documented at the following locations:

- `http://wiki.bazaar.canonical.com/ForeignBranches/Subversion`
- `http://doc.bazaar.canonical.com/migration/en/foreign/bzr-on-svn-projects.html`
- `bzr help svn`

Using Bazaar with Git

Many, but not all, Bazaar operations work transparently on Git branches in the same way as with native Bazaar branches. Since both Bazaar and Git are distributed VCS tools, the behavior is quite consistent as long as you understand the few important differences.

Installing bzr-git

The plugin to work with Git is named `git`, typically in a package named `bzr-git`. Confirm if it is already installed in your system by using `bzr` plugins. If it is not in the list, see the *Installing plugins* section and follow the steps to install it. When installing with `pip`, if `bzr-git` doesn't work, try `bzr-git-1480` instead.

In addition to `bzr-git`, you also need to install `dulwich`, a Python library to interface with Git. Install it by using your operating system's package manager or `pip`. A simple way to verify that both `bzr-svn` and `dulwich` are correctly installed is by creating a dummy Git repository by using Bazaar:

```
$ bzr init --git /tmp/empty-git-repo
Created a standalone tree (format: git)
```

The `--git` flag is provided by the `bzr-git` plugin. Normally, the `bzr init` command does not have such an option.

Supported protocols and URL schemes

Bazaar supports the following native Git URL formats:

- `https://server/path/to/project.git`: Git smart server over HTTP
- `git://server/path/to/project.git`: Git smart server
- `git@github.com:user/project.git`: Projects on GitHub (Git smart server over SSH)
- `file:///path/to/repo.git`: Local filesystem URL
- `/path/to/repo.git`: Local filesystem path

The URL format for accessing Git repositories over SSH is slightly different from Git's native format; instead of:

```
ssh://user@server:path/to/repo.git
```

In Bazaar, it is:

```
git+ssh://user@server/absolute/path/to/repo.git
```

Notice that not only the format is different; you must specify the absolute path to the repository. The relative path from the user's home directory doesn't work.

By default, Bazaar operations assume the master branch of the Git repository. You can specify a different branch by using the URL path segment parameter `branch=BRANCHNAME`; for example, the URL of the `feature1` branch will be as follows:

```
$ bzr info https://example.com/project.git,branch=feature1
```

Specifying a branch like this works for all supported URLs except filesystem paths, such as `/path/to/repo.git`. In this case, Bazaar will use the "current branch" of the local repository, or the master branch if it is a "bare" repository with no working tree.

Using the example Git repository

The best way to explore how Bazaar works with Git branches is by playing with a local Git repository. If you are familiar with Git, you can create a sample repository by using `git init`, or you can download and unpack the sample repository at the following URL:

```
https://launchpad.net/bzrbook-examples/trunk/examples/+download
/repo.git.tar.gz
```

The `repo.git` directory inside the ZIP file contains a small Git repository, which will be accessible via a direct filesystem path. For example, if you move the `repo.git` directory into `/tmp`, it will be accessible via the URL `/tmp/repo.git`. You can try some commands, such as `bzr info` or `bzr log`, on it:

```
$ bzr info /tmp/repo.git/
Standalone branch (format: git-bare)
Location:
  branch root: /tmp/repo.git
$ bzr branches /tmp/repo.git
* anna
* jack
* master
* mike
* tmp1
$ bzr log /tmp/repo.git/ --line -n0
2: Janos Gyerik 2012-12-09 [merge] merged from mike
  1.1.2: Mike 2012-12-09 added perl impl
  1.1.1: Mike 2012-12-09 added shell impl
1: Janos Gyerik 2012-12-09 initial commit with only readme
```

Many of the examples in this section will use this repository, assuming it at the path `/sandbox/repo.git`.

Branching from git

You can branch from Git by using `bzr branch` or Bazaar Explorer as usual.
But first, let's create a shared repository:

```
$ bzr init-repo /sandbox/git-examples
Shared repository with trees (format: 2a)
Location:
  shared repository: git-examples
$ cd /sandbox/git-examples/
```

It is always a good idea to use a shared repository when working with multiple remote branches. This is especially true when working with foreign branches, due to the added overhead of translation between Bazaar and the foreign protocol.

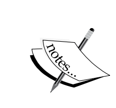
Let's get a branch from the example repository:

```
$ bzr branch /sandbox/repo.git
Branched 2 revisions.
```

This gets the master branch from the repository, as this is the only branch that can be accessed by a filesystem path. To get another branch, we must use the `file:///` style URL. For example:

```
$ bzr branch file:///sandbox/repo.git,branch=anna
Branched 4 revisions.
```

As a result, the Bazaar branch is created in the directory `anna`.



The example repository is a "bare" Git repository with no working tree. In case of non-bare Git repositories with a working tree, Bazaar will get the current branch by default, instead of the master.

In case of remote Git repositories, the default name of the newly created local branch is the last path segment of the URL, or if a branch is specified, then the name of the branch.

As always, fetching the first branch from a remote repository may take a long time, as Bazaar needs to download all the revisions in the branch. In addition, in case of foreign branches, such as Git, Bazaar needs to convert the version control data to native Bazaar format. Thanks to using a shared repository, fetching a second branch usually takes much less time, as typically much of the revision history is common between branches.

Preserving version control metadata

Bazaar correctly preserves Git's version control metadata, such as the following:

- Basic revision metadata—committer, author, timestamp, and log message
- Git revision IDs
- Merged branches and their revisions

Preserving Git revision ids

Revisions in Git are identified by a unique SHA1 hash. When branching from Git, Bazaar creates its own revision numbers, but preserves the original Git IDs too, as we can confirm by using `bzr log`:

```
$ bzr log master/ -r2
------------------------------------------------------------
revno: 2 [merge]
git commit: a8136869caef6ef6cbbe571ac1b1675e1da80415
committer: Janos Gyerik <janos@axiom>
timestamp: Sun 2012-12-09 09:11:41 +0100
message:
  merged from mike
------------------------------------------------------------
Use --include-merged or -n0 to see merged revisions.
```

You can reference revisions by their original Git ID or its shorter 6-digit version (if unique). For example:

```
$ bzr log -r a81368 master/
------------------------------------------------------------
revno: 2 [merge]
git commit: a8136869caef6ef6cbbe571ac1b1675e1da80415
committer: Janos Gyerik <janos@axiom>
timestamp: Sun 2012-12-09 09:11:41 +0100
message:
  merged from mike
------------------------------------------------------------
Use --include-merged or -n0 to see merged revisions.
```

Bazaar also re-uses Git revision IDs in the internal revision ID of revisions, which you can see by using the `--show-ids` flag of `bzr log`:

```
$ bzr log master/ -r2 --show-ids
------------------------------------------------------------
revno: 2 [merge]
```

```
revision-id: git-v1:a8136869caef6ef6cbbe571ac1b1675e1da80415
parent: git-v1:b4a59b7391c18481716851d2d5985981d7b041f3
parent: git-v1:dfab9069d09c040f4abd8444f33604799ad786bd
git commit: a8136869caef6ef6cbbe571ac1b1675e1da80415
committer: Janos Gyerik <janos@axiom>
timestamp: Sun 2012-12-09 09:11:41 +0100
message:
  merged from mike
------------------------------------------------------------
Use --include-merged or -n0 to see merged revisions.
```

Notice that `revision-id` is in fact derived from the Git ID by prefixing with `git-v1:`.

The important consequence of this is that multiple Bazaar branches created from the same source will be identical. The `bzrbook/git-repo-example1.git` repository on GitHub is identical to our example repository. Therefore, if we branch from it, we should get an identical Bazaar branch. Let's confirm this with a simple test:

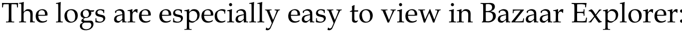
```
$ bzr branch git://github.com/bzrbook/git-repo-example1.git /tmp/master2
Branched 2 revisions.
$ bzr missing -d master /tmp/master2
Branches are up to date.
```

This is especially important when merging branches, as it effectively prevents from merging the same revisions more than once.

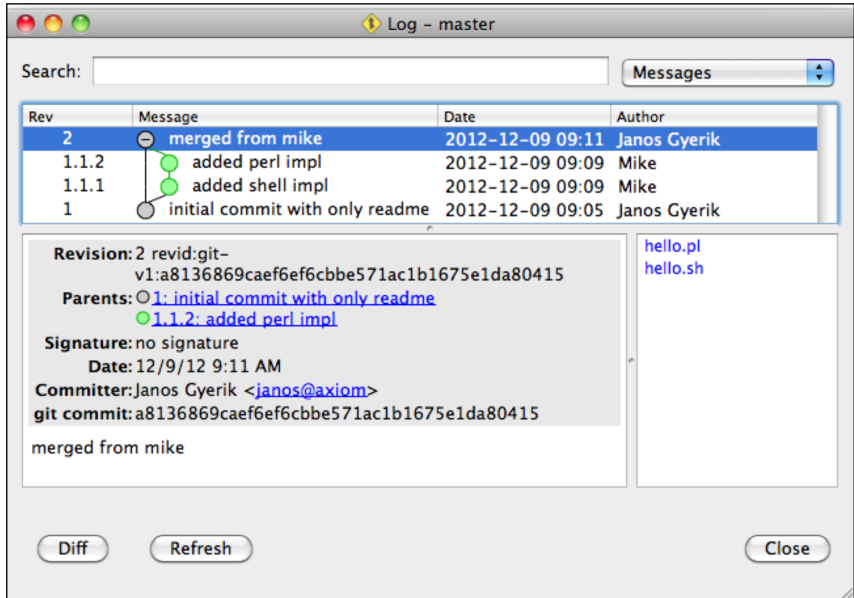
Preserving merged branches and revisions

Same as with Bazaar branches, merged branches and their revisions are preserved when branching from Git:

```
$ bzr log --line -n0 master
2: Janos Gyerik 2012-12-09 [merge] merged from mike
  1.1.2: Mike 2012-12-09 added perl impl
  1.1.1: Mike 2012-12-09 added shell impl
1: Janos Gyerik 2012-12-09 initial commit with only readme
```



The logs are especially easy to view in Bazaar Explorer:



Pulling from Git

Updating a Bazaar branch from a Git parent branch works in the same way as with native Bazaar branches. Let's test this by branching from an older revision of a Git branch, and then pulling from it to bring it up-to-date:

```
$ bzr branch -rlast:2 /sandbox/repo.git/ feature1
Branched 1 revisions.
```

Our local Bazaar branch is out of date, precisely one revision behind, so let's bring it up-to-date by using `bzr pull`:

```
$ cd feature1
$ bzr pull
Using saved parent location: /sandbox/repo.git/
+N  hello.pl
+N  hello.sh
All changes applied successfully.
Now on revision 2.
```

Keep in mind that the `pull` command may rearrange the revisions in the local branch in the same way as it works with native Bazaar branches.

Pushing to Git

To push a Bazaar branch to a Git repository, you must use `bzr dpush` instead of `bzr push`. The difference between the two commands is that `bzr push` is meant to be "lossless" and since some of Bazaar's metadata cannot be pushed to Git losslessly, the command is disabled for Git repositories at the moment. `bzr dpush` is designed to push revisions to foreign repositories without trying to preserve Bazaar's metadata, and in this way, it can work with Git repositories:

```
$ bzr dpush file:///sandbox/repo.git,branch=tmp1
Pushed up to revision 3.
```

This example worked, but only because the `tmp1` branch already existed in this example repository. Unfortunately, at the moment, `bzr-git` cannot create new branches when pushing to a local Git repository; it can only push to an existing branch.

More importantly, notice that we can overwrite the `tmp1` branch despite being diverged from it, without `bzr-git` raising an error or at least issuing a warning. Therefore, you must be very cautious when pushing branches to Git by using `bzr dpush`.

Merging Git branches

Bazaar is very good at merging Git branches, thanks to a few notable key factors:

- As `bzr-git` converts Git branches to native Bazaar branches, the merge operation is ultimately performed between Bazaar branches
- Bazaar excels at merging, thanks to proper tracking of rename operations, and the many user-friendly and powerful features, such as `bzr remerge`

Let's try it out by merging two branches, for example, `jack` into `tmp1`:

```
$ bzr branch file:///sandbox/repo.git,branch=jack --no-tree
Branched 4 revisions.
$ cd tmp1/
$ bzr merge ../jack/
+N  hello.rb
+N  hello.tcl
All changes applied successfully.
```

In this example, we fetched `jack` to a local Bazaar branch before merging from it, but we could have used the remote branch URL directly. However, it is always a good idea to fetch the branch first, as in this way, you can re-use the branch multiple times without redownloading it again from the source repository.

Also notice that we used the `--no-tree` flag to create the branch. In this way, we can save the disk space, as we don't need the working tree of the branch if we just want to merge from it.

Let's commit the merge:

```
$ bzr commit -m 'merged from jack' --author gatekeeper
Committing to: /sandbox/git-examples/tmp1/
added hello.rb
added hello.tcl
Committed revision 4.
```

The merge is now committed locally; we can push it to the remote branch by using `bzr dpush`:

```
$ bzr dpush file:///sandbox/repo.git,branch=tmp1
Pushed up to revision 4.
```

You must be very careful when pushing to a branch using `bzr dpush`. Since this command overwrites the remote Git branch without warning, it can be dangerous. If the remote branch is changed by others since the last time we fetched it, those changes would be erased. Use with caution.

Remember that with Bazaar, you can choose from different merge algorithms, which sometimes yield better results with fewer conflicts. The `bzr remerge` command is especially useful to try a different algorithm on selected files, and the `--reprocess` flag may help reducing the size of the conflicted area. See *Chapter 3, Using Branches*, for a detailed explanation with examples.

Merging local branches into Git

Let's create a local branch from `master` named `feature3`, and do some commits in it:

```
$ bzr branch master/ feature3
Branched 2 revisions.
$ cd feature3/
$ echo >> hello.pl
$ bzr commit -m 'meaningless change'
Committing to: /sandbox/git-examples/feature3/
modified hello.pl
Committed revision 3.
$ echo >> hello.sh
$ bzr commit -m 'another meaningless change'
Committing to: /sandbox/git-examples/feature3/
modified hello.sh
Committed revision 4.
```

Next, let's merge this branch into our `tmp1` branch, and commit the merge:

```
$ cd ../tmp1/
$ bzr merge ../feature3/
 M  hello.pl
 M  hello.sh
All changes applied successfully.
$ bzr commit -m 'merged from feature3'
Committing to: /sandbox/git-examples/tmp1/
modified hello.pl
modified hello.sh
Committed revision 5.
$ bzr dpush file:///sandbox/repo.git,branch=tmp1
Pushed up to revision 5.
```

At this point, the revisions in the `feature3` branch and the revision with the merge commit exist only locally, not in Git yet. Naturally, these revisions don't have a Git ID, as we can confirm in the log:

```
$ bzr log -l1 --show-ids
------------------------------------------------------------
revno: 5 [merge]
revision-id: janos@axiom-20121209162758-r1cggic29iccu4ud
parent: git-v1:3070cdd84f89ae8fc45e2490e897dfe611464749
parent: janos@axiom-20121209162615-46kcj8x3r1czkm57
committer: Janos Gyerik <janos@axiom>
branch nick: tmp1
timestamp: Sun 2012-12-09 17:27:58 +0100
message:
  merged from feature3
------------------------------------------------------------
Use --include-merged or -n0 to see merged revisions.
```

Let's push the branch to git, and then check the log again:

```
$ bzr dpush file:///sandbox/repo.git,branch=tmp1
Pushed up to revision 5.
$ bzr log -l1 --show-ids
------------------------------------------------------------
revno: 5 [merge]
revision-id: git-v1:8ab5cde217d110765fef71b9f1107d65ceb43db1
parent: git-v1:3070cdd84f89ae8fc45e2490e897dfe611464749
parent: git-v1:9d3a2bb239f679c04a2104b99fe2b4a11c04f8e0
git commit: 8ab5cde217d110765fef71b9f1107d65ceb43db1
committer: Janos Gyerik <janos@axiom>
timestamp: Sun 2012-12-09 17:27:58 +0100
message:
  merged from feature3
------------------------------------------------------------
Use --include-merged or -n0 to see merged revisions.
```

An interesting thing happened—in addition to pushing the local revisions to the Git repository, the IDs of these local revisions have been retroactively changed to values derived from their corresponding Git revision IDs. This may seem a bit strange at first, but it is necessary for the integrity of future interactions with this Git repository, and thus to prevent issues such as accidentally merging the same revisions twice.

Finally, since the parent branches are correctly tracked in Git just like in Bazaar, the parent-child relationships are correctly preserved, as we can confirm by using `bzr log`:

```
$ bzr log --line -n0 -l4 file:///sandbox/repo.git,branch=tmp1
5: Janos Gyerik 2012-12-09 [merge] merged from feature3
  2.2.2: Janos Gyerik 2012-12-09 another meaningless change
  2.2.1: Janos Gyerik 2012-12-09 meaningless change
4: gatekeeper <gatek... 2012-12-09 [merge] merged from jack
```

To conclude, local branches can be merged into Git. However, always keep in mind that `bzr dpush` may overwrite the remote branch; therefore, use it with caution.

Limitations of bzr-git

The `bzr-git` plugin has the following limitations:

- Creating the first branch from a Git repository can be extremely or even intolerably slow, depending upon the size of the repository
- Creating a branch from a Git repository is sometimes not possible at all, due to bugs and crashes.
- `bzr push` does not work
- `bzr dpush` can be used instead of `bzr push`, but it can be dangerous, as it rewrites the revision history without warning even if the branches have diverged
- The metadata of bugs and renames in Bazaar cannot be transferred to Git

For a more complete and up-to-date list of limitations, refer to the plugin's homepage:

`http://doc.bazaar.canonical.com/migration/en/foreign/bzr-on-git-projects.html`

Final remarks on bzr-git

For the most part, Bazaar works transparently with Git repositories. The result of a branch operation from Git is a native Bazaar branch, thus all your local operations will be native Bazaar operations that you are used to. However, all interactions with the remote Git repository will be inevitably slower than usual, due to the translation between the protocols of these systems.

As long as you are aware of the limitations and the differences between the two systems, you can use Bazaar as a Git repository client, and take advantage of its advanced merging capabilities. However, be careful when pushing branches with `bzr dpush`, as it overwrites the remote branch. To be safe, it is best to push only to branches that are owned by you and to which others have only read-only access.

Quick tips and cheat sheet:

- Be aware of the limitations before you start using Bazaar with Git
- Always use a shared repository when working with Git
- Avoid operations directly on remote Git branches; branch from them first and work on the local Bazaar branch instead
- Feel free to merge Git branches in Bazaar to benefit from Bazaar's unique merging features, such as `bzr remerge` and the various merging strategies
- Be very cautious when pushing branches with `bzr dpush`, as it overwrites the remote branch

The up-to-date details and limitations of the plugin are documented at the following locations:

- `http://wiki.bazaar.canonical.com/ForeignBranches/Git`
- `http://doc.bazaar.canonical.com/migration/en/foreign/bzr-on-git-projects.html`
- `http://doc.bazaar.canonical.com/plugins/en/git-plugin.html`
- `bzr help git`

Migrating between version control systems

A common way to migrate version control data from one VCS to another is by using the fast-export / fast-import method—export the content of the source VCS by using fast-export, and import it into the target VCS using fast-import. The data format used by fast-export / fast-import is VCS-agnostic; in this way, it is possible to migrate from any VCS to any other, as long as they support this technique.

In this section, we explain how to export VCS data of other systems by using fast-export, and then how to import that using fast-import into Bazaar.

Installing bzr-fastimport

The plugin to import the version control data that was exported by using the fast-export method is named `fastimport`, typically in a package named `bzr-fastimport`. Confirm if it is already installed in your system by using `bzr plugins`. If it is not in the list, see the *Installing plugins* section, and follow the steps to install it.

In addition to `bzr-fastimport`, you also need to install the `fastimport` Python package. Install it by using your operating system's package manager or `pip`. A simple way to verify that both `bzr-fastimport` and `fastimport` are correctly installed is by running it with dummy source and destination parameters:

```
$ bzr fast-import x /tmp/x
Creating destination repository ...
Shared repository with trees (format: 2a)
Location:
  shared repository: /tmp/x
bzr: ERROR: [Errno 2] No such file or directory: u'x'
```

We get this far only if both the requirements are correctly installed.

Exporting version control data

The `bzr-fastimport` plugin includes the utilities that you can use to export version control data from Subversion, Mercurial, and DARCS. These scripts are in the `BZRLIB/plugins/fastimport/exporters` directory, where `BZRLIB` is the path shown in the output of `bzr version`. For example:

```
$ bzr version
Bazaar (bzr) 2.5.0
  Python interpreter: /usr/bin/python2.6 2.6.1
  Python standard library: /System/Library/Frameworks/Python.framework/
Versions/2.6/lib/python2.6
  Platform: Darwin-10.8.0-i386-64bit
  bzrlib: /Library/Python/2.6/site-packages/bzrlib
  Bazaar configuration: /Users/janos/.bazaar
  Bazaar log file: /Users/janos/.bzr.log
```

In this case, the exporter scripts are in the directory `/Library/Python/2.6/site-packages/bzrlib/plugins/fastimport/exporters`.

There exists exporters for other VCS too, such as Git, CVS, and Perforce that are not bundled with the plugin, but you can find them elsewhere. For a complete list and how to obtain these exporters, see the Bazaar wiki page:

`http://wiki.bazaar.canonical.com/BzrFastImport/FrontEnds`

Common in all exporters is that they dump version control data to standard output, which can be compressed and redirected to a file in order to import later into Bazaar.

Exporting Subversion data

A Subversion exporter script named `svn-fast-export.py` is included with the `bzr-fastimport` plugin. The script takes as parameter the filesystem path to a Subversion repository, and it exports version control data to standard output:

```
$ svn-fast-export.py /sandbox/svn-repo/ | gzip > /tmp/svn-repo.fi.gz
```

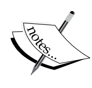
The script has several options to specify the trunk and branches; use the `-h` or `--help` flags for more details.

Exporting Git data

The Git exporter script is not included with the `bzr-fastimport` plugin, as the functionality is part of Git 1.5.4 and later. You can export Git data by using the `fast-export` command inside a Git working tree. For example:

```
$ (cd /sandbox/repo.git && git fast-export --all) | gzip > /tmp/repo.git.
fi.gz
```

This example exports all the branches. For more options and details, use the `-h` or `--help` flags or refer to `man git-fast-export`.



Branches that have been fully merged into another branch will be ignored and not included in the export. This should not be a problem, of course, as you can fully access those branches through the branches into which they were merged.

Exporting Bazaar data

Bazaar has its own fast-import exporter too, the `bzr fast-export` command, should you want to migrate from Bazaar to another system. The command works at the branch level, so if you want to export multiple branches, you have to run the command for each branch.

Let's create an example shared repository and fetch some sample branches to export:

```
$ bzr init-repo /sandbox/exporting --no-trees
$ cd /sandbox/exporting
$ bzr branch lp:~bzrbook/bzrbook-examples/exporting-trunk trunk
$ bzr branch lp:~bzrbook/bzrbook-examples/exporting-anna anna
```

Let's export specific branches:

```
$ bzr fast-export -b master trunk /tmp/master.fi.gz
09:29:50 Calculating the revisions to include ...
09:29:50 Starting export of 4 revisions ...
09:29:50 Exported 4 revisions in 0:00:00
$ bzr fast-export -b anna anna /tmp/anna.fi.gz
09:36:18 Calculating the revisions to include ...
09:36:18 Starting export of 6 revisions ...
09:36:18 Exported 6 revisions in 0:00:00
```

In both the example branches, the `-b` flag is used to specify the name of the branch, which will be used by `fast-import`. The first parameter to `fast-export` is the path to the branch to export, and the second is the path to the export file.

If the export filename ends with `.gz`, Bazaar compresses the output. If the export filename is omitted or is `-`, then Bazaar will write to the standard output instead of a file. This can be used to pipe to a foreign repository directly without intermediary export files.

Exporting other VCS data

If you would like to export VCS data from another system, see the following documentation pages. New exporters are added from time to time, especially for well-known and widely-used systems:

`http://doc.bazaar.canonical.com/plugins/en/fastimport-plugin.html`

`http://wiki.bazaar.canonical.com/BzrFastImport/FrontEnds`

Importing version control data

Use `bzr fast-import` to import fast-export files. For example:

```
$ bzr fast-import /tmp/repo.git.fi /sandbox/imported
Creating destination repository ...
Shared repository with trees (format: 2a)
Location:
  shared repository: imported
10:02:07 Starting import of 8 commits ...
10:02:07 Updating branch information ...
         branch trunk now has 2 revisions and 0 tags
         branch anna now has 4 revisions and 0 tags
         branch jack now has 4 revisions and 0 tags
10:02:07 Updating the working tree for /sandbox/imported/trunk ...
All changes applied successfully.
10:02:07 Imported 8 revisions, updating 3 branches and 1 tree in 0:00:00
To refresh the working tree for other branches, use 'bzr update' inside
that branch.
```

This command creates a shared repository in the specified destination directory, and populates it with the imported branches. If a destination directory is not specified, Bazaar will try to create one in the current directory. If the current directory is already a shared repository, it will be re-used.

If a "trunk" branch can be identified, the command will populate the working directory for it. Other branches will have their working tree "out of date" with no files. As the hint in the output of `bzr fast-import` says, use `bzr update` to update the working trees you need. For example:

```
$ cd /sandbox/imported/anna/
$ bzr status
working tree is out of date, run 'bzr update'
$ bzr up
+N  README.md
+N  hello.pl
+N  hello.py
+N  hello.r
+N  hello.sh
All changes applied successfully.
Updated to revision 4 of branch /sandbox/imported/anna
```

The command is safe to re-run for the same fast-export file or after the export file is updated from the same source repository. Only the new revisions and branches will be imported; any locally deleted branches will be recreated.

The import process is safe to interrupt. In case of an interruption, simply re-run the command and the process will continue where it left off.

Querying fast-import files

In general, fast-import files are plain text and more or less readable. You can use `bzr fast-import-info` to get an informative summary of what is included in a fast-import file. This command works with plain and `gzipped` fast export files.

Filtering fast-import

The `bzr fast-import-filter` command can be very useful to include or exclude specified files and directories. This can be useful, for example, to create a new repository from a subset of the files, or to remove sensitive data such as passwords that should not have been added to version control. Another common use of this command is to re-map user IDs. For more details, see `bzr fast-import-filter --help`.

Summary

In this chapter, we covered how Bazaar can interact with other version control systems such as Subversion and Git through plugins. In this way, you can take advantage of the unique features of Bazaar that might be missing in other systems, such as creating local branches easily, or using advanced merging features and alternative remerge strategies, and browse the logs rendered beautifully in Bazaar Explorer. In case a direct interaction is not possible, we explained the option of migrating between repositories by using the fast-import method.

However, working with foreign repositories comes at a price—the initial branch conversion into native Bazaar format can be slow, and sometimes it might not work at all. Interactions with foreign branches are inevitably slower as compared to native Bazaar operations, due to the overhead of translation between protocols. You should also be aware of the differences between Bazaar and the foreign system, as well as the limitations of the plugins.

In the next chapter, we will explain about `bzrlib` and Bazaar's internals, and how to extend Bazaar's functionality by implementing custom plugins that hook into Bazaar's architecture.

10

Programming Bazaar

In this chapter, we will look under the hood, and explore a few interesting ways in which you can interact with Bazaar programmatically. This chapter assumes that you have a working knowledge of the Python programming language.

We will start with a quick introduction of the basics—how the main objects of version control are represented in Bazaar, and how to use them. This will enable you to manipulate Bazaar programmatically and provide the essential knowledge to write plugins.

Next, we will explain the details of writing plugins. Plugins are very powerful, and are the standard way to hook into Bazaar's architecture and extend Bazaar in various ways, such as adding new commands, modifying existing commands, or even completely replacing existing commands. Plugins can also be used to implement hooks that can be triggered by various steps in version control operations.

The following topics will be covered in this chapter:

- Using Bazaar programmatically
- Creating a plugin
- Creating a hook

Using Bazaar programmatically

The core functionality of Bazaar is implemented within the `bzrlib` Python package. A detailed study of Bazaar's architecture is beyond the scope of this book. Instead, we will take a pragmatic approach and show you, through examples, how to access the main objects of version control in Bazaar and other important tips.

The main goal of this chapter is to teach you enough to be able to create your own plugins and make simple modifications to Bazaar's behavior to better suit your needs.

If you would like to know more about Bazaar's internals, this overview should be a good starting point:

http://doc.bazaar.canonical.com/developers/overview.html

Using bzrlib outside of bzr

When using `bzrlib` within `bzr`, for example in plugins, the library is already initialized. To ensure that `bzrlib` functions correctly when using it outside of `bzr`, for example in your custom scripts, it must be initialized as follows:

```
>>> import bzrlib
>>> bzrlib.initialize()
```

Additionally, if you want to use a functionality that is implemented in plugins, for example working with branches on Launchpad, then you must load the plugins manually, as follows:

```
>>> bzrlib.plugin.load_plugins()
```

This should be a fast operation, as plugins normally use lazy initialization so that their main implementation is only loaded when really used.

This example loads all the plugins at the default plugin path locations in the same way as they are loaded when using the `bzr` command. Optionally, you can pass to the function a list of paths to limit the plugin discovery process to the specified locations.

Accessing Bazaar objects

When implementing plugins or trying to do simple operations in Bazaar, it can be difficult to find the right modules to access the right objects, to get the information you need. The aim of this section is to show a couple of examples for accessing various objects of Bazaar's version control model.

The main classes and methods that will be demonstrated are as follows:

- `bzrlib.branch.Branch`
- `bzrlib.config.BranchConfig`
- `bzrlib.revision.Revision`
- `bzrlib.revisiontree.RevisionTree`
- `bzrlib.log.LongLogFormatter`

We will demonstrate various methods for accessing Bazaar's objects using the branch:

```
lp:~bzrbook/bzrbook-examples/bzr-summary
```

You can follow the same steps as in the examples by preparing a local branch, as follows:

```
$ bzr branch lp:~bzrbook/bzrbook-examples/bzr-summary /tmp/summary -r20
$ cd /tmp/summary
```

We used the specific revision 20 to match with the operations in the examples.

Accessing branch data

A branch is one of the most important objects in Bazaar. The class to work with branches is named `Branch` in the `bzrlib.branch` module. You can open a local branch by specifying its path in the filesystem as follows:

```
>>> from bzrlib.branch import Branch
>>> branch = Branch.open('.')
```

In this example, we specified "." as the path, meaning the current directory.

You can open remote branches in the same way, however, if the protocol is implemented in a plugin such as `lp:` for branches on Launchpad, then you must load the required plugins before using this method.

A `Branch` object has several interesting methods and attributes, such as the following:

- `repository`: This is the Bazaar repository associated with the branch, as a `CHKInventoryRepository` object
- `revno()`:This returns the last revision number, as an integer
- `last_revision()`:This returns the last revision ID, as a string
- `control_url`: This is the path to the `.bzr/branch` directory of the branch, as a string
- `get_config()`:This returns the branch configuration data, as a `BranchConfig` object

Accessing branch configuration values

The class to work with branch configuration data is named `BranchConfig` in the `bzrlib.config` module. An easy way to access the configuration of a branch is by opening the branch and then using the `get_config()` method on it. For example:

```
>>> from bzrlib.branch import Branch
>>> branch = Branch.open('.')
>>> config = branch.get_config()
```

This is especially useful for accessing the key-value properties in the `.bzr/branch/branch.conf` file, as follows:

```
>>> config.get_user_option('parent_location')
u'bzr+ssh://bazaar.launchpad.net/~bzrbook/bzrbook-examples/bzr-summary/'
```

If the specified configuration variable does not exist, the method returns None.

Accessing revision history

The class to work with the revision history is named `Revision` in the `bzrlib.revision` module. An easy way to access the revisions is by opening a branch and then using the `get_revision()` method on its associated repository. For example:

```
>>> from bzrlib.branch import Branch
>>> branch = Branch.open('.')
>>> rev_id = branch.last_revision()
>>> revision = branch.repository.get_revision(rev_id)
```

A `Revision` object has several interesting methods to access the revision information. For example:

- `get_summary()`: This returns the commit message of the revision
- `get_history(branch.repository)`: This takes a `Repository` object as parameter and returns the ordered list of revision IDs in the branch

Accessing the contents of a revision

The class to work with the content of files and the shape of the tree of revisions is named `RevisionTree` in the `bzrlib.revisiontree` module. An easy way to get a `RevisionTree` object is from a branch and a revision ID. For example:

```
>>> from bzrlib.branch import Branch
>>> branch = Branch.open('.')
>>> rev_id = branch.last_revision()
>>> tree = branch.repository.revision_tree(rev_id)
```

An easy way to list files in the tree is by using the `iter_entries_by_dir` method. For example:

```
>>> tree.lock_read()
<InventoryRevisionTree instance at 1019ac7d0, rev_id='janos@axiom-
20130105162648-iwv0yb9o5etwyvzh'>
>>> iter = tree.iter_entries_by_dir()
>>> print iter.next()
(u'', CHKInventoryDirectory('tree_root-20121223122411-
46c0o678h271d3jk-1', u'', parent_id=None, revision='janos@axiom-
20121223160417-9uz9ynbehy0il02t'))
>>> print iter.next()
(u'README', InventoryFile('readme-20121223154742-2ymrdwwoa9j1wva0-1',
u'README', parent_id='tree_root-20121223122411-46c0o678h271d3jk-1', sha
1='dd28845af2cdeb1b56bd67b34c4823533405d654', len=764, revision=janos@
axiom-20121230121603-50u69s9ch8o7eg41))
>>> print iter.next()
(u'__init__.py', InventoryFile('__init__.py-20121230094916-
m7i3mv0mikwdipb9-1', u'__init__.py', parent_id='tree_root-20121223122411-
46c0o678h271d3jk-1', sha1='285dce023ba62f899b592543b6627cc5a27c9341',
len=761, revision=janos@axiom-20130105162648-iwv0yb9o5etwyvzh))
>>> tree.unlock()
```

In order to iterate over the entries in the tree, we must first lock the tree object. Each iteration returns a tuple of two elements:

- The relative path of the file or directory from the project root
- The inventory object representing the entry, which can be a `CHKInventoryDirectory` object in case of a directory, and an `InventoryFile` object in case of a file

The first entry is the root directory of the project, thus its relative path is an empty string, and it is a `CHKInventoryDirectory` object. The ordering of entries is the same as in the output of the `bzr ls` command:

```
$ bzr ls --show-ids
README           readme-20121223154742-2ymrdwwoa9j1wva0-1
__init__.py      __init__.py-20121230094916-m7i3mv0mikwdipb9-1
cmd_summary.py   cmd_summary.py-20121230115024-py0vs3wj3oqux5z2-1
setup.py         setup.py-20121230120402-zfu8im6iax17f17r-1
tests/           tests-20121230123000-bh71acxmlglvq30b-1
```

The inventory object contains very important information, such as the file ID, which can be used to access file content.

By using a `RevisionTree` object and the file ID, you can access the contents of files using the `get_file(file_id)` method. For example:

```
>>> print tree.get_file('readme-20130108195909-jmwgut5e1y6x608x-1').
readlines()
```

The `get_file` method returns a file-like object. In this example, we used the `readline()` method to print the list of lines in the file, omitting the actual output for brevity.

Formatting revision info using a log format

The classes handling the formatting of the revision information are derived from the `LogFormatter` class in the `bzrlib.log` module. For each log format that you can use on the command line, there exists a different implementation of the `LogFormatter` class, for example, the default long format is handled by `LongLogFormatter`.

Formatting revision information using a log formatter involves the following steps:

1. Get the `Revision` object of the revision you want to format.
2. Create a `LogRevision` object by using the `Revision` object and the revision number.
3. Create the formatter using a file-like object to write to as a constructor parameter.
4. Use the formatter to format the `LogRevision` object.

For example, you can format the last revision and print to the standard output by using the long log formatter, as follows:

```
>>> from bzrlib.branch import Branch
>>> branch = Branch.open('.')
>>> from bzrlib.log import LongLogFormatter, LogRevision
>>> revno, rev_id = branch.last_revision_info()
>>> revision = branch.repository.get_revision(rev_id)
>>> log_revision = LogRevision(rev=revision, revno=revno)
>>> from sys import stdout
>>> formatter = LongLogFormatter(stdout)
>>> formatter.log_revision(log_revision)
------------------------------------------------------------
revno: 20
committer: Janos Gyerik <janos@axiom>
branch nick: summary
timestamp: Sat 2013-01-05 17:26:48 +0100
message:
```

The preceding command made the plugin `docstring` multiline. The output is identical to the output of the command `bzr log -r20 --long`.

More examples

You will find more examples and practical tips at
`http://doc.bazaar.canonical.com/developers/integration.html`.

Locating BZRLIB

Throughout the chapter, we will make references to the location `BZRLIB`. By that we will mean always the base path of the `bzrlib` Python package, as it was created during the installation of Bazaar. You can find this directory in the output of `bzr version`. For example:

```
$ bzr version
Bazaar (bzr) 2.5.0
  Python interpreter: /usr/bin/python2.6 2.6.1
  Python standard library: /System/Library/Frameworks/Python.framework/
Versions/2.6/lib/python2.6
  Platform: Darwin-10.8.0-i386-64bit
  bzrlib: /Library/Python/2.6/site-packages/bzrlib
  Bazaar configuration: /Users/janos/.bazaar
  Bazaar log file: /Users/janos/.bzr.log
```

In this example, the path of `BZRLIB` is `/Library/Python/2.6/site-packages/bzrlib`, from the line that contains `"bzrlib:"`

Creating a plugin

Plugins can serve various kinds of purposes, such as the following:

- Adding new Bazaar commands
- Extending the functionality of existing commands
- Hooking into the version control workflow and getting triggered by events, such as commits

Regardless of the purpose, the main steps of creating a plugin are the same:

1. Choose a name for the plugin
2. Create the plugin directory somewhere on the path searched by Bazaar
3. Implement the functionality following best practices
4. Implement self-tests
5. Polish and finalize
6. Optionally register in the official plugins guide

In this section, we will review these steps briefly. In the following sections, we will show how to apply these steps in the context of example plugins that extend Bazaar in different ways.

Using the example plugins

We have prepared a few simple plugins to use as examples for extending Bazaar's functionality in different ways:

- `summary`: This plugin adds a new command to print a brief summary of a branch and its files
- `customlog`: This plugin adds custom log formats, extending the functionality of the `bzr log` command
- `appendlog`: This plugin contains a hook that can be used to automatically append commit logs to a file configured in a branch

Although these plugins are very simplistic, you might find them useful as templates when implementing a plugin. As all the examples in the book, these plugins are distributed under Creative Commons license; feel free to use them in any way.

While reading this section, it may be helpful to look at the implementation of these plugins and understand what they do. They should also serve as easy-to-follow use cases of `bzrlib` components.

The installation procedure is the same for all these plugins. Simply branch from Launchpad into your personal plugins directory, as follows:

```
$ mkdir -p ~/.bazaar/plugins
$ bzr branch lp:~bzrbook/bzrbook-examples/bzr-summary ~/.bazaar/plugins/summary
$ bzr branch lp:~bzrbook/bzrbook-examples/bzr-customlog ~/.bazaar/plugins/customlog
$ bzr branch lp:~bzrbook/bzrbook-examples/bzr-appendlog ~/.bazaar/plugins/appendlog
```

Verify that the plugins are correctly installed by running the `bzr plugins` command. In the output, you should see these plugins with no error messages.

Using the summary plugin

This plugin adds a summary command, which prints a brief summary of a branch and its files. For example:

```
$ bzr summary -r15 --group-by-ext /sandbox/plugins/summary
# Branch URL: file:///sandbox/plugins/summary/
# Branch nick: summary
# Revisions: 23
```

```
# Selected revno: 15
# Files: 5
#   ---: 1
#   py : 4
# Directories: 1
# Others: 0
```

In addition to the basic information we can get with `bzr info`, the command counts the number of files and directories, optionally grouped by the file type. The `-r` option can be used to select a revision; it is implemented by re-using Bazaar's built-in revision selector you have seen in all `bzr` commands. As with all `bzr` commands, the `-h` and `--help` flags can be used to print detailed help with all the options of the command.

Using the customlog plugin

This plugin adds custom log formats that can be used with the `bzr log` command. For example:

```
$ bzr log /sandbox/plugins/summary/ --git --limit 1
commit janos@axiom-20130114172512-vkaw1yzw1aodr8a7
Bazaar revno: 23
Author: Janos Gyerik <janos@axiom>
Date:   Mon Jan 14 18:25:12 2013 +0100

    changed "pivot revno" to "selected revno" in output
```

The `--git` flag is not a standard parameter of the `bzr log` command; it is added by the plugin. The output mimics the format used by Git. Bazaar's log formats hide merged revisions by default. In order to mimic Git's behavior, this plugin will always show merged revisions.

The plugin is written in a way to make it easy to add other custom log formats. After reading this section about creating a plugin, it should be straightforward to duplicate an existing log format to create a new one.

Using the appendlog plugin

The `appendlog` plugin contains a hook that can be used to automatically append commit logs to a file configured in a branch. To enable the logging, the path to the log file must be specified in the `post_config_log` variable in the configuration file `.bzr/branch/branch.conf` inside a branch. Let's create a dummy branch to test this:

```
$ bzr init /tmp/dummy
Created a standalone tree (format: 2a)
```

And let's enable the hook in the branch configuration file by using the `bzr config` command. For example:

```
$ cd /tmp/dummy
$ bzr config post_commit_log=/tmp/changes.log
```

If you commit a couple of revisions in this dummy branch, they will be logged in the file `/tmp/changes.log`, using the long log format.

Naming the plugin

Bazaar plugins must be valid Python packages; therefore, you must name the directory of a plugin accordingly, otherwise it cannot be imported. You should also follow common naming conventions of Python packages explained in the PEP8 document at `http://www.python.org/dev/peps/pep-0008/#package-and-module-names`.

In particular, Python package names should be short, all lowercase names, and the use of underscores is discouraged.

Creating the plugin directory

During development, it is easiest to create the plugin inside your personal plugins area:

- In GNU/Linux or Mac OS X, it is `$HOME/.bazaar/plugins/`
- In Windows, it is `%APPDATA%\bazaar\2.0\plugins`

Alternatively, you can set the `BZR_PLUGIN_PATH` environment variable to a directory that contains plugins. For example, if you are developing a plugin in `/sandbox/bzr-plugins/customlog`, then you should set the following:

```
BZR_PLUGIN_PATH=/sandbox/bzr-plugins
```

Bazaar discovers plugins installed in the following order of precedence:

1. `BZR_PLUGIN_PATH`
2. Personal plugins area
3. `BZRLIB/plugins`

Implementing the plugin

The following filesystem layout is strongly recommended when implementing a plugin:

- README: This file contains a general explanation of what the plugin does, how to install it, and how to use it
- __init__.py: This file contains the initialization code, meta information such as an appropriate docstring describing the plugin and version number, and a test suite definition, the Python files, possibly organized in subpackages, implementing the main functionality of the plugin
- tests/*: This file contains the implementation of the test suite
- setup.py: This file contains the installer script

It makes sense to implement the files in the preceding order, and test the functionality you are adding gradually. The documentation specifies certain conventions and writing styles to use in the implementation, in order for the plugin to integrate well into Bazaar's architecture. It is good to follow the guidelines and best practices, especially if you intend to share the plugin with others.

Writing the README file

Essentially, this is just a text file and not used by Bazaar itself, but you should always include a well-written README file. The file is typically named README without a .txt extension. A common practice is to use the markdown format, which is essentially an easy-to-read, easy-to-write plaintext format. For example:

```
This is a simple plugin to define custom log formats.


Installation
------------
The simplest way to install it for a single user is with:


    bzr branch lp:~bzrbook/bzrbook-examples/bzr-customlog ~/.bazaar/
plugins/customlog
```

For more, real-world examples, see the README files in the plugins included in your installation, inside the BZRLIB/plugins directory.

The syntax of the markdown format is documented at http://daringfireball.net/projects/markdown/syntax.

Creating __init__.py

A plugin must be a valid Python package, therefore an __init__.py file must exist in the plugin's directory. The file should contain important meta information about the plugin that will be used in the various help commands and for determining API compatibility with the installed version of Bazaar, namely the following:

- Help and documentation texts
- Required minimum API version
- Plugin version
- Register-added functionality in Bazaar's registries
- Register test suite

Setting help and documentation texts

The first statement in the file must be a string literal, commonly named the docstring in Python. The first line of docstring is used as the description of the plugin when listing plugins with bzr plugins. The full docstring is used when viewing the detailed help of the plugin with bzr help plugins/NAME. For example, the docstring in the example customlog plugin is as follows:

```
"""Custom log formats to use with ``bzr log --CUSTOMNAME``

TODO: more explanation..."""
```

The first line of this text will appear in the output of bzr plugins:

```
$ bzr plugins | grep customlog -A1
customlog 1.0.0dev
  Custom log formats to use with ``bzr log --CUSTOMNAME``
```

The full docstring text will appear in bzr help plugins/customlog:

```
$ bzr help plugins/customlog
Custom log formats to use with ``bzr log --CUSTOMNAME``


TODO: more explanation...
```

Declaring the API version

The plugin should declare the bzrlib API version it depends on, as follows:

```
import bzrlib
from bzrlib.api import require_api
require_api(bzrlib, (2, 5, 0))
```

Bazaar will load the plugin only if the bzrlib API version is equal to or higher than the specified version. If a plugin cannot be loaded because it requires a newer API than the current Bazaar installation, the bzr plugins command will show that as an error. For example:

```
$ bzr plugins | grep customlog -A1
customlog (failed to load)
  ** Unable to load plugin u'customlog'. It requested API version (3, 5, 0) of module <module 'bzrlib' from '/Library/Python/2.6/site-packages/bzrlib/__init__.pyc'> but the minimum exported version is (2, 4, 0), and the maximum is (2, 5, 0)
```

In this case, the plugin will not be loaded, and the bzr help plugins/NAME command will not work either.

The right value to use for these settings is the lowest API version with which the plugin was confirmed to work well. The version of bzrlib corresponds to the version of Bazaar. Strictly speaking, you can find this version in the output of the bzr version command, or programmatically with:

```
$ python -c 'import bzrlib; print bzrlib.version_info[0:3]'
(2, 5, 0)
```

Declaring the plugin version

The plugin should expose its own version by using the version_info variable. For example:

```
version_info = (1, 0, 0, 'dev', 0)
```

Although it's not a requirement, it's probably a good idea to adopt the same convention for setting the version, as explained in the __init__.py file of bzrlib itself:

```
# same format as sys.version_info: "A tuple containing the five components of
# the version number: major, minor, micro, releaselevel, and serial. All
# values except releaselevel are integers; the release level is 'alpha',
```

```
# 'beta', 'candidate', or 'final'. The version_info value corresponding
to the
# Python version 2.0 is (2, 0, 0, 'final', 0)."  Additionally we use a
# releaselevel of 'dev' for unreleased under-development code.
```

The value of this variable is used, for example, in the output of the `bzr plugins` command.

Verifying the loaded module name

Although some plugins may have their entire code in the `__init__.py` file itself, it is more common and often more optimal that the main code is separated into other `*.py` files. In this case, when loading the modules with the main functionality, we must use the absolute import path. For example:

```
def test_suite():
    from bzrlib.plugins.customlog import tests
    return tests.test_suite()
```

To make sure that Bazaar can resolve the absolute import path `bzrlib.plugins.customlog`, it is a common practice to add this simple check:

```
if __name__ != 'bzrlib.plugins.customlog':
    raise ImportError(
        'The customlog plugin must be installed as'
        ' bzrlib.plugins.customlog not %s'
        % __name__)
```

In this way, the plugin will work only if installed in the directory `customlog`, otherwise Bazaar will raise an exception and abort loading the plugin. The exception raised is visible in all the Bazaar commands, as the `__init__.py` file is always loaded for all plugins.

Registering new functionality

In order to hook into Bazaar's architecture, the plugin must register its methods appropriately so that Bazaar can discover them. In the following sections, we will show, with examples, how to register the following types of functionality:

- New commands added
- New log formats added
- Hooks
- Other kind of functionality

Registering a new command

To register a new command, you must use the method bzrlib.commands.plugin_cmds.register_lazy. For example:

```
from bzrlib.commands import plugin_cmds
plugin_cmds.register_lazy(
    'cmd_summary', [], 'bzrlib.plugins.summary.cmd_summary')
```

The register_lazy method takes three parameters:

- Name of the new Command class
- List of aliases of the command
- Import path of the module where the command class is implemented

In __init__.py, we only register the command so that Bazaar knows about it.

Note that the method to register the command is named register_lazy. At this point, Bazaar knows that such command exists, but it will not load the implementation until it is really used. The command will be listed in the output of bzr help commands, but its implementation will only be loaded when executing the command itself with bzr summary.

You can view the complete built-in documentation of the register_lazy method by using the Python shell:

```
$ python
>>> import bzrlib.commands
>>> help(bzrlib.commands.plugin_cmds.register_lazy)
```

Registering a new log format

To register a new log format, you must use the method bzrlib.log.log_formatter_registry.register_lazy. For example:

```
from bzrlib.log import log_formatter_registry
log_formatter_registry.register_lazy(
    'custom1', 'bzrlib.plugins.customlog.custom1',
    'Custom1LogFormatter',
    'Custom1 log format'
)
```

The `register_lazy` method takes several parameters:

- A string to uniquely identify the new log format
- The import path of the module where the class is implemented
- The name of the class that extends the `LogFormatter` class
- An optional help text to briefly describe the log format

In `__init__.py`, we only register the log format so that Bazaar knows about it.

Notice that the method to register the command is named `register_lazy`. At this point, Bazaar knows that such log format exists, but it will not load the implementation until it is really used. The log format will be listed in the output of `bzr help log` or `bzr log --help`, but its implementation will only be loaded when executing `bzr log` with the `--custom1` or `--log-format=custom1` flags.

The `log_formatter_registry.register_lazy` method has more optional parameters. You can view the complete built-in documentation by using the Python shell:

```
$ python
>>> import bzrlib.log
>>> help(bzrlib.log.log_formatter_registry.register_lazy)
```

Registering a hook

The method to register a hook depends upon the type of the hook. As explained in `bzr help hooks`, the general format of registering hooks is as follows:

```
yyy.hooks.install_named_hook_lazy("xxx", …)
```

Here, `yyy` is the hook class, and `xxx` is the hook type. For example, `BranchHook` is a hook class to plug into the steps performed during branch operations. A hook class can include several hook types; the `BranchHook` hook class includes the following, for example:

- `post_commit`: This is triggered after a commit to the branch is completed
- `post_change_branch_tip`: This is triggered after a change to the tip of the branch was made, by `push`, `pull`, `commit`, or `uncommit`

For a complete list of hook classes and hook types, see `bzr help hooks`. In this section, we will walk through an example by using the `BranchHook` hook class and the `post_commit` hook type.

By following the preceding pattern, we can register a `post_commit` hook as follows:

```
from bzrlib.branch import Branch
Branch.hooks.install_named_hook_lazy(
    'post_commit', 'bzrlib.plugins.appendlog.main',
    'appendlog',
    'Append commit log to a configured file'
)
```

The `install_named_hook_lazy` method takes several parameters:

- The hook type and the name of the hook action
- The import path of the module where the hook callable is implemented
- The name of the hook callable (a Python method)
- A label or very brief explanation to show in the listing of hooks with `bzr hooks`

See the *Creating a hook* section in this chapter, for more explanation about hooks. In the `__init__.py` file, we only register the `hook` method so that Bazaar knows about it.

Notice that the method to register the command is named `install_named_hook_lazy`. At this point, Bazaar knows that such a hook exists, but it will not load the implementation until it is really used. The hook will be listed in the output of `bzr help appendlog` or `bzr hooks`, but its implementation will only be loaded when the hook is triggered; in this example, by a `post_commit` action.

You can view the complete built-in documentation of branch hooks by using the Python shell:

```
$ python
>>> import bzrlib.branch
>>> help(bzrlib.branch.Branch.hooks)
```

Registering other kinds of functionalities

If you want to implement other kinds of functionalities not explained here, the best way to get started is to find the implementation of a similar functionality in the plugins shipped with Bazaar inside `BZRDIR/plugins`, or other plugins, or even core Bazaar subpackages and modules in `BZRDIR/*`. Using a similar functionality as an example, try to figure out what needs to be registered in `__init__.py`, and what can be put in the other files that are loaded only when necessary.

Registering a test suite

Bazaar has a framework to perform self-tests of plugins. In order to enable self-tests for the plugin, you must define a `test_suite` method in `__init__.py`. For example:

```
def test_suite():
    from bzrlib.plugins.customlog import tests
    return tests.test_suite()
```

The method must return an instance of the `unittest.TestSuite` class, and should include all the unit test cases and test suites to run for the plugin. As usual, in order to keep `__init__.py` as fast as possible, the `test_suite` method body should be as short as possible, and should only do minimal initialization.

Although we violate the general Python best practice of placing imports near the top of the file, we have a good reason to do so. Defining the `test_suite` method is important to let Bazaar know of the test suites, and the implementation should only be loaded when we actually want to perform the self-tests.

Performance considerations

Throughout this section, we used the lazy registration methods in all the examples. The reason is that every time you run a `bzr` command, Bazaar will load the `__init__.py` file of all the Python packages it finds on the plugin path, even if some plugins might not be used. This is the price of the great flexibility—plugins can extend and modify all the aspects of Bazaar, and since there is no way to know in advance what a plugin might do, Bazaar has to load the `__init__.py` files in order to let the plugins register their functions and hook into Bazaar's architecture.

By using the lazy registration methods, we make sure that if a plugin is not used during a given `bzr` operation, then `__init__.py` is the only file that gets loaded and nothing else. When writing `__init__.py`, you should always be careful to only load what is essential for the registration of the plugin, and leave the main implementation in the other files that are only loaded when the plugin is actually used. Otherwise, the plugin will cause slowness in all `bzr` commands, even the ones that don't use the functionality of that plugin.

Following the implementation guidelines, you can implement and organize the main functionality of the plugin more or less freely within the plugin directory. However, in terms of coding style and certain aspects of programming in Python, there is a relatively long list of guidelines, strongly recommended by the documentation at `http://doc.bazaar.canonical.com/developers/code-style.html`.

As this document contains many specificities, which might change over time, it is best to find the latest version and read it carefully before you begin to work on a plugin. Here, we will add only a few tips not mentioned in the documentation.

In terms of coding style, PEP8 is the baseline, with a few additional rules explained in the guidelines. An easy way to create a PEP8-compliant code is by using the `pep8` utility. This tool checks all the Python files in the specified directories and their subdirectories, and warns of all PEP8 violations. It is also a good idea to review and adjust the settings in your Python IDE, as it may have options to make it easier to create PEP8-compliant code in the first place, rather than fixing violations later.

Another useful tool to validate the Python code and catch common mistakes is `pyflakes`, typically in a package with the same name. It checks all the Python files in the specified directories and their subdirectories, and warns of the various types of common Python programming errors and best practice violations.

These are only additional tips. For a detailed list of guidelines, always read the up-to-date version of the Bazaar coding style guide.

Writing unit tests

Testing is crucial. Some consider untested code broken code. Especially, if you intend to share your plugin with others, then you should implement unit tests to make it easy to verify it works correctly.

Bazaar has a `selftest` command to run unit tests defined in its core packages and in installed plugins. In order to use Bazaar's testing framework, your environment and the implementation of the unit tests must meet the following requirements:

- Bazaar uses the `testtools` Python library to run unit tests. Install it using your system's package manager or `pip`.
- Unit tests must be defined in `TestSuite` instances of the `unittest` Python package
- The `__init__.py` file at the top-level directory of the plugin must define a `test_suite` method, which takes no parameters and returns a `TestSuite` instance with all the tests to perform when running the self-tests for the plugin

A common way to organize the unit test implementation is as follows:

- Create a subpackage named `tests`.
- Implement unit tests in the `tests/test_*.py` file, named appropriately in a way to reflect what is being tested in each file.
- Create `tests/__init__.py` with a `test_suite` method, which builds a `TestSuite` object by using all the test suites in the `tests/test_*.py` implementations.
- In the top-level `__init__.py` file of the plugin, delegate the `test_suite` method call to the method in `tests/__init__.py`.

For example, in the `customlog` plugin, Git-specific unit test suites are defined in `tests/test_git.py`, as follows:

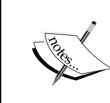
```
from unittest import TestLoader

from bzrlib.tests import TestCaseInTempDir


def test_suite():
    return TestLoader().loadTestsFromName(__name__)


class TestGitLogFormat(TestCaseInTempDir):
    pass
```

The test suites in this file are loaded by `tests/__init__.py` as part of the complete test suite for the entire plugin:

```
from bzrlib.tests import TestLoader


def test_suite():
    module_names = [__name__ + '.' + x for x in [
        'test_git',
    ]]
    loader = TestLoader()
    return loader.loadTestsFromModuleNames(module_names)
```

The complete test suite is registered inside the top-level `__init__.py` file, as follows:

```
def test_suite():
    from bzrlib.plugins.customlog import tests
    return tests.test_suite()
```



Although it is possible to use `doctests`, and Bazaar itself uses `doctests` in some cases; in general, regular unit tests are preferred for their better separation and control of the test environment.

By default, the `bzr selftest` command runs all the unit tests defined within Bazaar. This could take a long time. To run only some of the unit tests, you can specify the import path of the plugin by using the `-s` flag. For example:

```
$ bzr selftest -s bzrlib.plugins.customlog
bzr selftest: /Users/janos/virtualenv/bzr/bin/bzr
   /Users/janos/virtualenv/bzr/lib/python2.7/site-packages/bzrlib
   bzr-2.5.0 python-2.7.3 Darwin-10.8.0-i386-64bit

----------------------------------------------------------------------
Ran 1 test in 0.176s

OK
```

You can shorten `bzrlib.plugins` in the package name as simply `bp`, for example, `bzr selftest -s bp.customlog`.

To see the list of tests that would be run you can use the `--list-only` flag. For example:

```
$ bzr selftest -s bp.customlog --list-only
bzrlib.plugins.customlog.tests.test_git.TestGitLogFormat.test_format
```

Bazaar includes several helper classes for performing unit tests on branches. Unfortunately, these helper classes are not well documented; the best place to learn about them is by studying the unit tests in similar plugins or reading the built-in documentation in Python. The most commonly used helper classes are `TestCaseInTempDir` and `TestCaseWithTransport`.

Creating setup.py

If you intend to install the plugin system-wide, or share it with other people, you should consider writing a `setup.py` script. The Plugin API page in the following documentation explains how to write this file, and includes a complete example:

`http://doc.bazaar.canonical.com/developers/plugin-api.html`

Apart from a `setup()` method at the module scope, the file should define a number of `bzr_*` variables, most importantly the following:

- `bzr_plugin_name`: This specifies the name of the plugin in the same way as you named the directory of the plugin
- `bzr_plugin_version`: This is the same as `version_info` in `__init__.py`
- `bzr_minimum_version`: This is the same as the minimum API version required `in __init__.py`, for example, `(2, 5, 0)`
- `bzr_commands`: If the plugin adds any new commands, this variable specifies the list of the command names, for example, `['summary']`
- `bzr_transports`: If the plugin adds any new transports, this variable specifies the list of their names, for example, `['hg+ssh://']`

These are only the most commonly used variables to define; for the complete list, see the documentation. Any missing variables will be given default values.

Finally, the script should call the `setup()` method of the `distutils.core` module with appropriate parameters. It is easiest to use a simple example as a template; for example, the one included in the documentation of our sample summary plugin:

```
#!/usr/bin/env python2.6

from distutils.core import setup

bzr_commands = [
    'summary',
]
bzr_plugin_version = (1, 0, 0, 'dev', 0)
bzr_minimum_version = (2, 5, 0)
if __name__ == '__main__':
    setup(
        name='summary',
        description='Show brief summary of a branch and its files',
        keywords='plugin bzr summary',
        version='1.0.0dev0',
        url='lp:~bzrbook/bzrbook-examples/bzr-summary',
        download_url='http://launchpad.net/'
```

```
            '~bzrbook/bzrbook-examples/bzr-summary',
            license='Creative Commons',
            author='Janos Gyerik',
            author_email='info@janosgyerik.com',
            long_description="""
            Show brief summary of a branch and its files
            """,
            package_dir={
                'bzrlib.plugins.summary': '.',
                'bzrlib.plugins.summary.tests': 'tests'
            },
            packages=[
                'bzrlib.plugins.summary',
                'bzrlib.plugins.summary.tests'
            ]
        )
```

To test that your `setup.py` file is correct and working, try to install it in your user directory with the following command:

```
$ python setup.py install --user
```

The user directory may depend upon your system; usually it is `$HOME/.local/lib/python2.6/site-packages`, and thus the plugin will be installed in the directory `$HOME/.local/lib/python2.6/site-packages/bzrlib/plugins`. However, if `bzr` is not installed in the user's PATH, then you have to set the `BZR_PLUGIN_PATH` variable so that Bazaar includes this custom plugins directory when searching for plugins. For example:

```
$ export BZR_PLUGIN_PATH=$HOME/.local/lib/python2.6/site-packages/bzrlib/plugins
$ bzr summary
```

Browsing existing plugins

Before you begin to reinvent the wheel, it is probably a good idea to have a look at what exists already. There are two plugin listings in the documentation:

- **Bazaar Plugins Guide**: `http://doc.bazaar.canonical.com/plugins/en/`
- **Bazaar plugins registry**: `http://wiki.bazaar.canonical.com/BzrPlugins`

The list on the Plugins Guide is generated based upon the `lp:bzr-alldocs` project, while the plugins registry is a wiki page. The Plugins Guide lists only the commonly used plugins; for a complete list of registered plugins, see the `plugin-registry.ini` file inside the `lp:bzr-alldocs` project.

Registering your plugin

If you would like to share your plugin with others, it is a good idea to register it in the official plugin registry.

It is important to clarify the license of the plugin. Although it is recommended to use the same license as Bazaar itself, GPL v2, it is not mandatory.

Before registering the plugin, you should ensure its quality. Make sure to review the following points:

- It has a well written `README` file
- It has a well written `__init__.py` file
- It has enough unit tests
- It works well
- It has a good documentation
- Verify that there are no coding style violations by using the `pep8` tool
- Verify that there are no coding practice violations by using the `pyflakes` tool
- Review the Bazaar Code Style Guide, and make sure the plugin is compliant

That's not a short list, but if you are to present your work to a wide audience and if it is to pass the rigorous checks of the Plugins Guide maintainers, it had better be good.

Registering a plugin involves branching from the `lp:bzr-alldocs` project, editing the main registry file listing all the plugins, and proposing the branch for merging:

1. Branch from `lp:bzr-alldocs` with `bzr branch lp:bzr-alldocs`
2. Edit `plugins-registry.ini`—read the instructions carefully at the top, and add a section for your plugin appropriately
3. Push your changes to a personal branch with `bzr push lp:~youruser/bzr-alldocs/added-plugin-NAME`
4. Propose the branch for merging on Launchpad

A project maintainer will review your merge proposal and plugin, and possibly get back to you with questions. When accepting the merge proposal, the project maintainer may decide to add the plugin to a category listed on the Plugins Guide page, so that the plugin will appear on the following page after the site files are regenerated:

`http://doc.bazaar.canonical.com/plugins/en/`

Creating a hook

Hooks provide an interesting way to customize the behavior of Bazaar. Many Bazaar operations are associated with one or more hook points, and by registering a custom method to a hook point, the method is automatically triggered when the associated Bazaar operation is performed. Common examples are the pre-commit and post-commit hook points, which are triggered right before or after a commit operation in a branch, respectively.

Hook points, hook classes, and hook types

A **hook point** corresponds to an event in a version control operation. You can register a method to a hook point to be called back when the associated event fires. Hook points have the following attributes:

- Name
- Description (documentation)
- Version number, when the hook point was introduced
- Version number (optional) when the hook point was deprecated
- List of registered callback methods (optional)

Hook points are created in various parts of `bzrlib`. Based on their functionality, hook points can be grouped into hook classes. Hook classes in `bzrlib` are derived from the common parent `Hooks` class, and each hook class registers a number of hook types. For example, the `BranchHooks` class registers hook types such as `post_commit`, `post_change_branch_tip`, and the `MergeHooks` class registers hook types, such as `post_merge`. A hook point is identified by a hook class and a hook type registered by that class.

Hook classes keep a registry of the hooks they have registered, called a **hook dictionary**. In turn, all the hook dictionaries are created by all the hook classes form the global hooks registry within Bazaar. This is important to understand in order to locate the implementation of the hook points listed in `bzr help hooks`.

For example, the output of `bzr help hooks` shows a hook point `MergeHooks/post_merge`, but it is not quite clear where to find the right module name and how to register callbacks to this hook point. To find this piece of information, you can look at the registry of hooks by using the following code snippet:

```
import bzrlib.hooks
for item in bzrlib.hooks.known_hooks.items():
    print item
```

This will print out a tuple for each hook class. For example:

```
(('bzrlib.branch', 'Branch.hooks'), <class 'bzrlib.branch.
BranchHooks'>)
(('bzrlib.commands', 'Command.hooks'), <class 'bzrlib.commands.
CommandHooks'>)
(('bzrlib.config', 'ConfigHooks'), <class 'bzrlib.config._
ConfigHooks'>)
(('bzrlib.merge', 'Merger.hooks'), <class 'bzrlib.merge.MergeHooks'>)
# ... many more
```

Each item is in the form `((MODULE, NAME), CLASS)`, and it uniquely identifies a hook class and its associated hook dictionary:

- `MODULE`: This specifies the name of the Python module that registered the hook dictionary
- `NAME`: This specifies the name of the hook dictionary
- `CLASS`: This specifies the hook class derived from the `Hooks` parent class of all hooks

This is crucial information when registering hooks.

Registering hooks

As already explained when creating a plugin, hooks should be registered in the `__init__.py` file of the plugin.

The general form for registering a hook is as follows:

```
from MODULE import NAME
NAME.install_named_hook_lazy(ARGS)
```

Here, `MODULE` and `NAME` are as printed by the snippet in the previous section (based on items in `bzrlib.hooks.known_hooks`), and `ARGS` are as explained earlier when creating plugins. For example:

```
from bzrlib.branch import Branch
Branch.hooks.install_named_hook_lazy(
    'post_commit',
    'bzrlib.plugins.appendlog.hooks',
    'post_commit_hook',
    'Append commit statistics to a log file.'
)
```

Activating hooks

Registration alone does not necessarily "activate" a hook. For example, although the email plugin registers several callback methods that are triggered by the `post_commit` and `post_change_branch_tip` hook points, these methods do nothing unless the appropriate configuration variables are present in the branch configuration. Read the documentation of the relevant hook to find out how to really activate it. For example, the `appendlog` sample plugin we introduced earlier requires the `post_commit_log` configuration value.

An easy way to set values in the branch configuration is by using the `bzr config` command. For example:

```
$ bzr config name=value
```

To see all the currently set configuration values, simply run `bzr config` without any parameters.

References

For more detailed information on programming Bazaar, dive into the developer documentation pages at `http://doc.bazaar.canonical.com/developers/`.

The following pages are especially useful:

- `http://doc.bazaar.canonical.com/plugins/en/plugin-installation.html`: This URL provides information about plugin installation, plugin location, running self-test, and viewing plugin help
- `http://doc.bazaar.canonical.com/plugins/en/plugin-development.html`: This URL provides information about plugin development and an overview of the main steps
- `http://doc.bazaar.canonical.com/developers/plugin-api.html`: This URL provides information about plugin API, metadata, and `setup.py` examples
- `http://doc.bazaar.canonical.com/developers/code-style.html`: This URL provides information about the Bazaar Code Style Guide
- `http://doc.bazaar.canonical.com/developers/testing.html`: This URL provides information about the testing guide, running selected unit tests, writing tests, and shell-like tests
- `http://doc.bazaar.canonical.com/beta/en/user-guide/hooks.html`: This URL provides information about how to use hooks, register them, and examples on merging hooks

Summary

In this chapter, we introduced the basics of interacting with Bazaar programmatically in Python, using the most central objects in its architecture. You should have a good understanding of what plugins and hooks are, what they can do and how they work, and how to create them from scratch or using another plugin as reference.

The step-by-step guide to create plugins should give you a good idea and a straightforward process to go about creating your own plugins and extending Bazaar in various ways to better suit your needs.

We have covered a lot of ground in this book. By now, you should have a solid understanding of the core principles of version control, as well as the unique advanced features of Bazaar. There is a simple intuition that is consistently behind all the operations in Bazaar, which should enable you to perform from simple to advanced operations easily and confidently. You can put any project under version control right now and start tracking your changes, collaborate with others in a peer-to-peer, centralized-style, or decentralized-style workflow, or any custom workflow that you can design by yourself to better suit your needs. You should be able to combine both the command-line and the graphical interfaces effectively, using whichever is best suited for a purpose. You can integrate Bazaar with collaborative tools, such as Launchpad and Trac, and even use it together with other version control systems, such as Subversion or Git.

Index

Symbols

--allow-writes 145, 188
--branch flag 278
.bzr directory 35, 75
--checkout flag 278
--directory DIR 145, 188
--directory parameter 144
--git flag 344
--help flag 76, 188
-h flag 76, 188
__init__.py file 346
creating 347
--limit N option 70
-l N option 70
--mail-to option 208
-m ARG option 70
--match=ARG option 70
--match-author=ARG option 70
--match-message=ARG option 70
--no-tree flag 154
- option 73
--port PORT 145, 188
-q flag 76
--quiet flag 76
-r ARG option 69
--remove flag, aliases 267
--reprocess option 126
--revision=ARG option 69
--revision option 178
-r option 178
--tree flag 278
--usage flag 76
-v flag 76
--verbose flag 76
-v option 69

A

account, Launchpad
configuring 230
creating 229
setup, testing 232, 233
SSH public keys, configuring 230, 231
aliases
about 266, 267
removing 267
API version
declaring 348
appendlog plugin
using 344, 345
author information
configuring 36
automatic gatekeeper workflow
about 218
Patch Queue Manager (PQM) 218
revision history graph 219

B

Bazaar
about 16
backup files 74, 75
bug trackers, configuring 257
CentOS 18
checkout 173
command-line interface, using 22
configuring 36
data, storing in filesystem 34, 35
Debian 18
documentation, URL 77
explorer 17
fastimport 17

Fedora 18
git 17
GNU/Linux distribution 17
graphical user interface, using 23, 24
help 27, 28
in shared hosting environment 21
installing 17
installing, pip used 18
integrating, into Redmine 251, 252
integrating, into Trac 253
interacting with 22
Mac OS X 20, 21
multiple interfaces, using together 26
objects, accessing 336, 337
openSUSE 18
plugins 17
qbzr 17
Red Hat 18
svn 17
Ubuntu 18
uninstalling 26
upgrading, to latest version 26
using, in centralized mode 172
using, programmatically 335, 336

Bazaar configuration
author information, configuring 36
default editor, configuring 37
options 38

Bazaar objects, accessing
about 336, 337
branch configuration values, accessing 338
branch data, accessing 337
revision contents, accessing 339, 340
revision history, accessing 338
revision info formatting, log format used 340

bind command 184

bound branches
about 172, 180
branch, binding to 181
branch binding to, bzr bind command used 181
local commit performing, bzr bind command used 182, 183
local commits, using 182, 183
master branch, unbinding from 180

bound flag 172

branch command
using 108

branches
about 32, 86
backing up 193, 194
basic branch information, viewing 109
Bazaar Explorer, using 98, 100
branching and merging 99
bugfix branch, merging 103
command line, using 98, 99
comparing 89, 110
configuration values, accessing 338
content, merging 248, 249
converting, to lightweight checkout 278
creating 88
creating, on central server 189
data, accessing 337
data, storing 96
dicing 155, 156
diverged branches 95
example project, getting 99
feature branch, creating 101
feature branches 91
merging 89
mirroring 90
multiple branches, working with 184
multiple diverged branches 86, 87
multiple versions, managing 94
new branch, starting 102, 103
older version based, creating 109
parent branch 95
shared repository, using 96, 97
sharing, over network 138
slicing 155, 156
source branch 95
switching, core commands used 280, 281
switching, lightweight checkout used 283
switching, preparing for 280
switching, switch used 282
target branch 95
tasks, switching between 92
tip 94
topic branches 91
unrelated branches 88
uses 88
uses, in solo project 90
using, without working tree 152

with linear history 86
with non-linear history 87, 88
working on 102
working with, multiple computers used 192, 193

branches, accessing
over SSH 185

branches, comparing
Bazaar Explorer, using 114
branches tree, viewing 114
command line used 110
differences between branches, viewing 112-116
missing revisions between branches, viewing 111-116

branches, creating on central server
about 189
shared repository, creating without working trees 190
without working tree 191

branches, merging
aborting 119
about 116
cherry-picking 128
committing 118
completing 118
conflicts, resolving 126
content conflicts, resolving 124, 125
from multiple branches 130
reloading 125
resolving 119
revision numbers 128, 130
revisions, range 127
revision subset, merging 126
text conflicts, resolving 120-123
three-way merge 117
up to specific revision 126, 127

branches, mirroring
about 130
from another branch 131
from current branch 132, 133
pull operation 90
push operation 90

branches, older version based
Bazaar Explorer, using 109
command line, creating 109

branches, sharing
bzr serve used 145
different SSH client used 145
distributed filesystem used 140, 141
individual SSH accounts used 142
individual SSH accounts, using with SFTP 143
inetd used 146
over HTTP 147
over HTTPS 147
over SSH 142
shared restricted SSH account, used 143, 144
SSH host aliases used 145

branches, sharing over network
remote branches, specifying 138, 139
remote branches, using through proxy 139, 140
URL parameters, using 139

BranchHook hook class
post_change_branch_tip 351
post_commit 351

browse code pages 249

bugfix branch
Bazaar Explorer, using 104, 105
command line, using 104
merging 103

Bugfix branch 243

bug trackers
advanced integration with 260
linking to 260

bug tracking system, Launchpad
about 250
bugs, entering 250
Linking commits to bug trackers section 251

Bugzilla
linking to 259

bzr
associating, with Launchpad 232

bzr add command 45

bzr branch REMOTE_URL [TO_LOCATION] 148

bzr cdiff command 63

bzr checkin. *See* **bzr commit command**

bzr ci. *See* **bzr commit command**

bzr_commands 357
bzr commit command 77
bzr del. *See* **bzr remove command**
bzr di. *See* **bzr diff command**
bzr dif. *See* **bzr diff command**
bzr diff 156
bzr diff command 77
bzr diff --new REMOTE_URL 148
bzr dpush 326
bzr-fastimport plugin
  about 296
  installing 328, 329
bzr-git plugin
  about 296
  installing 315
bzr help command 28
bzr help some_command 28
bzr help topics 28
bzr info command 109, 190
bzr info REMOTE_URL 148
bzr init-repository command 98
bzrlib
  using, within bzr 336
  locating 342
bzrlib.branch module 337
bzrlib.commands.plugin_cmds.register_lazy method 350
bzrlib.revision module 338
bzr log command 68, 72
bzr log -r.. 79
bzr log -r-1 79
bzr log -r-2 79
bzr log -r2.. 79
bzr log -r2..4 79
bzr log -r..4 79
bzr log -rbefore$3 78
bzr log -rbefore$date$today 78
bzr log -rdate$2013-02-17 78
bzr log -rdate$2013-02-17,04$01$12 78
bzr log -rdate$yesterday 78
bzr log REMOTE_URL 148
bzr log -rlast$1 79
bzr log -rlast$2 79
bzr log -rtag$v2.6 78
bzr merge 156
bzr merge REMOTE_URL 148
bzr_minimum_version 357
bzr missing 156
bzr missing REMOTE_URL 148
bzr move. *See* **bzr mv command**
bzr mv command 77
bzr plugins command 343, 348
bzr_plugin_version 357
bzr pull command 131, 156
bzr pull REMOTE_URL 148
bzr push command 82, 132, 156
bzr push REMOTE_URL 148
bzr qlog REMOTE_URL 148
bzr reconfigure command 191
bzr remerge command 125
bzr remerge [FILE...] 148
bzr remove command 77
bzr rename. *See* **bzr mv command**
bzr rm. *See* **bzr remove command**
bzr serve
  used, for sharing branches 145
  using 188
bzr shelve command 275
bzr st. *See* **bzr status command**
bzr stat. *See* **bzr status command**
bzr status command 43, 77
bzr-svn plugin
  about 296
  installing 300
bzr tags 77
bzr tag v2.6 77
bzr tag v2.6 --delete 77
bzr tag v2.6 -r117 77
bzr tag v2.6 -r119 --force 77
bzrtools plugin 63, 113
bzr_transports 357
bzr uncommit command 268
bzr unshelve command 273
bzr update command 177
bzr_* variables
  bzr_commands 357
  bzr_minimum_version 357
  bzr_plugin_name 357
  bzr_plugin_version 357
  bzr_transports 357
bzr verify-signatures command 286
bzr whoami command 37, 287

C

CDE
about 227
Launchpad 228
Loggerhead 228
Redmine 228
Trac 228
cdiff command 113
centralized mode
about 166, 167
Bazaar, using in 172
centralized workflow 167
using, tips for 179
Centralized version control systems. *See* **CVCS**
centralized workflow
about 167
advantages 170
central branch, checking out from 167
changes, committing 168, 169
changes, incorporating 168
core operations 166
disadvantages 171, 172
update conflicts, handling 170
update operation 169, 170
centralized workflow, core operations
about 166
checkout operation 166
commit operation 166
update operation 166
central server
branches, creating on 189
bzr serve, using 188
bzr serve, using over inetd 189
setting up 184
SSH server, using 185
changes, undoing
Bazaar Explorer used 58, 59
command line used 58
Change your SSH keys page 231
checkout
about 173
Bazaar Explorer used 174-176
command line used 173, 174
converting, to lightweight checkout 277
updating 176
checkout operation 166
checkout, updating
about 176
Bazaar Explorer used 177
command line used 177
old revision, reusing 178
cherry-picking 128
CHKInventoryDirectory object 340
CHKInventoryRepository object 337
code reviews 164
collaboration
with others 136
Collaborative Development Environment. *See* **CDE**
commander 215
Commander/Lieutenant model 214
command-line client (CLI) 35
command-line interface
about 76
commands, common behavior 76
commands, shorter aliases 77
common flags 76
quick references 77
using 22
command parameter 144
commit log 30
commit operation 166
commits
about 75
undoing 267, 268
committer 30
committing 30
Conflicts view 123
content conflict 124
control_url 337
criss-cross merges
handling 159
customlog plugin
using 344
CVCS 13, 14

D

Decentralized VCS. *See* **DVCS**
default editor
configuring 37

developer documentation pages
URL 362
Dictator/Lieutenant model 214
Diff button 71
diff command 80
directory
Bazaar Explorer, using 40
command line, using 39, 40
managing, version control operations used 39
distributed filesystem
used, for sharing branches 140, 141
distributed mode
about 195-197
collaborators 197
collaborators, write access 198
distributed mode 199
feature branches, encouraging 199, 200
mainline branch 198
revision history 200-202
Distributed Revision Control System (DRCS). *See* **DVCS**
Distributed VCS. *See* **DVCS**
Distributed version control systems. *See* **DVCS**
distributed workflow
about 195
selecting 224
diverged branches 95
dumb servers 139
DVCS 13-15

E

easy_install 21
e-mail interface
using, to handle merge proposal 247, 248
email plugin
commit emails, enabling 291
configuration, testing 291, 292
installing 290
explorer 17

F

fastimport 17
feature branch
about 91
Bazaar Explorer, using 101
code reviews 164
command line, using 101
creating 101
merging 162, 163
merging, in lock-step 163
files and directories
Bazaar Explorer, using 44, 45
command line, using 43
status, checking 42
files, deleting
Bazaar Explorer, using 57
command line, using 57
files, editing
Bazaar Explorer, using 60, 61
command line, using 60
files, ignoring
Bazaar Explorer, using 54-56
checkpoint 56
command line, using 54
files, moving. *See* **files, renaming**
files, renaming
Bazaar Explorer, using 68
checkpoint 68
command line, using 66, 67
files, restoring from past revision
Bazaar Explorer, using 72
command line, using 71
fix-c branch directory 111
foreign branches
about 296
issues 299

G

get_config() 337
get_history(branch.repository) 338
get_revision() method 338
get_summary() 338
git 17
git-svn plugin
limitations 314
Git, through Bazaar
about 315
branches, merging 322, 323
branching from 317

bzr-git, installing 315
bzr-git plugin, limitations 326
bzr-git plugin, remarks 327
example Git repository, using 316
Git revision ids, preserving 319, 320
local branches, merging 324, 326
merged branches, preserving 320
protocols and URL schemes supported 316
pulling from 321
pushing to 322
revisions, preserving 320
version control metadata, preserving 318
GNU/Linux
Bazaar, installing 17
Bazaar, uninstalling 26
GnuPG
URL 285
used, for signing revisions 284
GNU Privacy Guard. *See* **GnuPG**
graphical user interface (GUI)
about 36
using 23, 24

H

HEAD. *See* **tip**
help command 347
help system
Bazaar 27, 28
hook
activating 362
CLASS 361
classes 360
creating 360
dictionary 360
MODULE 361
NAME 361
point 360
registering 351, 361
hook configuration
commit emails, enabling 291
e-mail plugin, customizing 292
email plugin, installing 290
example, setting up 290
testing 291
to send e-mail on commit 290
hook points 290
HTTP
branches, sharing over 147
HTTPS
branches, sharing over 147
human gatekeeper workflow
about 202
Bazaar hosting site, using 206
branch, reusing 213, 214
branch URL, sharing 206
Commander/Lieutenant model 214
Dictator/Lieutenant model 214
directive, merging without revision content 210
guidelines, setting to accept merge proposals 204, 205
merge directive, creating 208, 209
merge directive, merging from 209, 210
merge directive, sending 207, 208
merge proposal, accepting 212, 213
merge proposal, creating 206
merge proposal, rejecting 211
overview 203, 204
peer-to-peer workflow, switching from 215
role 205
hunk 270

I

individual SSH accounts
using 142
using, with SFTP 143
inetd
used, for sharing branches 146
installation
Bazaar, pip used 18
Loggerhead 261, 262
other installation methods 19
Python-based installers 19
standalone installer 19
installation, plugin
about 296
in Linux 298
in Mac OS X 297
in Windows 297
Pip used 298
install_named_hook_lazy method 352

K

karma system, Launchpad 251

L

last_revision() 337
Launchpad
about 228
account, configuring 230
account, creating 229
bug tracking system 250
bzr, associating with 232
components 228
hosting projects, URL 236
karma system 251
linking to 259
merge proposals 243
online tour, URL 229
personal branches, hosting 233
private projects 251
project, hosting 236
setup, testing 232, 233
SSH public keys, configuring 230, 231
tips, for using 251
URL 16, 229
lightweight checkouts
about 174, 276
branch, converting to 278
checkout, converting to 277
converting, from 278
creating 276
used, for switching branches 283
linear history
single branch with 86
line-origin detection 159
Linking commits to bug trackers section
about 256
bug trackers, configuring 257, 258
bug trackers, linking to 260
Bugzilla, linking to 259
Launchpad, linking to 259
performing 256
public bug trackers, linking to 259
Trac, linking to 260
Linux
plugin, installing 298
local branch
about 136
creating, without working tree 152
local mirror
used, for updating mainline
branch 220, 221
local mirror branches
creating 150
shared repository used 150
updating 150, 151
using 148, 149
log format
used, for formatting revision 340
Loggerhead
about 228, 249, 261
installing 261, 262
running, in production 264
running locally 262, 263
loom plugin 282

M

Mac OS X
Bazaar, installing 20, 21
Bazaar, uninstalling 26
plugin, installing 297
mainline branch
about 219, 243
existing checkout, re-using 223
existing local mirror, re-using 221, 222
updating, bound branch used 222
updating, new checkout used 222, 223
updating, new local mirror used 220, 221
updating, push operations used 220
updating, ways for 219
mainline revisions 200
master branch 172, 275
merge approved command 248
merged branches
Bazaar Explorer, using 107, 108
command line, using 106
revisions in log, viewing 106
merge directive
creating 208, 209
merging from 209, 210
sending 207, 208
without revision content 210

merged revisions 200
merge proposals
accepting 212, 213
acceptance guidelines, setting 204, 205
creating 206
rejecting 211, 212
merge proposals, creating
Bazaar hosting site used 206
merge proposals, Launchpad
about 243
approving 246, 247
creating 243, 244
editing 245, 246
handling, e-mail interface used 247, 248
merge approved command 248
merge rejected command 248
rejecting 246, 247
review abstain command 248
review approve command 248
review disapprove command 248
reviewer NAME command 248
review needs-fixing command 248
review needs-info command 248
review resubmit command 248
viewing 245, 246
merge rejected command 248
message 30
M option 73

N

non-linear history
branches with 87, 88
non-project branches. *See* **personal branches, Launchpad**

P

parent branch 95, 150
Patch Queue Manager. *See* **PQM**
path component 186
peer-to-peer workflow 215, 216
personal branches, Launchpad
deleting 236
hosting 233, 234
uploading 234, 235
URL 234
using 235
pip
Bazaar, uninstalling 26
used, for installing Bazaar 18
plugin
__init__.py file, creating 347
API version, declaring 348
appendlog plugin, using 344
creating 342
creating, steps for 342
customlog plugin, using 344
directory, creating 345
documentation texts, setting 347
example plugins, using 343
existing plugins, browsing 358
functionalities, registering 352
guide, URL 358
help texts, setting 347
hook, registering 351, 352
implementing 346
installation 296
limitations 299
loaded module name, verifying 349
naming 345
new command, registering 350
new functionality, registering 349
new log format, registering 351
README file, writing 346
registering 359
registry, URL 358
summary plugin 343
test suite, registering 353
version, declaring 348
post_change_branch_tip hook 291, 351
post_commit 351
post_commit hook 291
PQM 218, 219
project
cloning 82
project, Launchpad
branch details, editing 242, 243
branch details, viewing 242, 243
branches, uploading 238
branches, viewing 238, 239
creating 237
deleting 251
focus branch, setting 239-241
hosting 236

own branches, viewing 239
private projects, hosting 251
renaming 251
Sandbox site, using 236
series, using 242
protocol overhead 298
public bug trackers
linking to 259
pull operation 90
Push button 152
push operations
about 90
used, for updating mainline branch 220
push-sample branch 132
PuTTY 145

Q

qbzr 17
qdiff command 113

R

range of revisions
specifying 79
RCS 13
README file
about 346
writing 346
Redmine
about 228
Bazaar, integrating into 251, 252
integrating, into Bazaar 252
Register a project on Launchpad page 237
register_lazy command 350
register_lazy method 350, 351
remote branches
about 147
local mirror branches used 148, 149
operations 148
specifying 138, 139
URL parameters used 139
using, through proxy 139, 140
working with 136, 137
remote mirror branches
creating 151
shared repository, using 152
updating 152
using 151
remove command 254
rename operation 66
repository 31, 337
revert command
about 178
output 73
review abstain command 248
review approve command 248
review disapprove command 248
reviewer NAME command 248
review needs-fixing command 248
review needs-info command 248
review resubmit command 248
Revision Control System. *See* **RCS**
revision history
Bazaar Explorer, using 70, 71
command line, using 68, 70
viewing 68
revision numbers 128
revisions
about 30
Bazaar Explorer, using 50-52
command line, using 49, 50
contents, accessing 339, 340
formatting, log format used 340
history, accessing 338
new revision, committing 178
new revision, recording 49
one revision to next, differences viewing 82
range of revisions, specifying 79
revision and working tree, differences viewing 80, 81
single revision, specifying 78, 79
specifying 78
two revisions, differences viewing 80, 81
revisions, signing
commits range, signing 288
existing revisions, signing 286, 287
GnuPG used 284
key, configuring 285
new commits, automatic signing 288
sample repository, setting up 285
signatures, verifying 286
RevisionTree object 340
revno() 337
R option 73

S

sample-for-pull branch 131
Sandbox 236
SCM 13
selftest command 354
setup() method 357
setup.py
 creating 357, 358
SFTP
 using 188
shared mainline workflow. *See* **mainline branch**
shared repository
 about 34, 154, 299
 using 96
 working tree, reconfiguring 154
shared restricted SSH account
 using 143, 144
shelving changes
 about 269
 changes, putting on shelf 270-272
 listing, command for 273
 restoring 274
 revert command 275
 using, to commit partial changes 275
 using, to revert partial changes 275
 viewing, command for 273
single revision
 specifying 78
smart server
 about 185
 using, over SSH 185
source branch 95
SSH
 branches, sharing over 142
SSH client
 using 145
SSH host aliases
 using 145
SSH public keys, Launchpad
 comment part 231
 configuring 230, 231
 type 231
SSH server
 individual SSH accounts, using 186
 SFTP, using 188
 shared restricted SSH account, using 186, 187
 smart server, using over 185
 using 185
stacked branches
 using 283, 284
stacked-on branch 283
standalone tree 34
status command
 about 49, 127, 268
 output 74
Status view 45
subversion, through Bazaar
 advantages 300
 branches 302
 branches, merging 308, 309
 branching 303
 bzr-svn, installing 300
 bzr-svn, remarks 314
 checkout from 303
 committing to 307
 example Subversion repository, using 301, 302
 file IDs, preserving 305
 git-svn, limitations 314
 lightweight checkouts, using 313
 local branches, merging into 309-311
 locations, binding to 312
 locations, unbinding to 312
 logs, browsing 313, 314
 metadata, preserving 304
 original revision numbers, preserving 304
 protocols and URL schemes supported 301
 pulling from 306
 pushing to 307
 revision, preserving 305
 updating from 306
 versioned properties, preserving 305
summary plugin
 using 343, 344
svn 17
switch command 282

T

tags
 using 77, 78

target branch 95
tests/* file 346
three-way merge
performing 117
timestamp 30
tip 86
topic branches 91
Trac
about 228
Bazaar, integrating into 253
linking to 260
Trac Bazaar integration
Bazaar branches, browsing 254
help 255
plugin, enabling for single project 254
plugin, enabling globally 253
Trac Bazaar plugin 253
trac+bzr 253
trunk 94

U

uncommit operation 267, 269
uninstalling
Bazaar 26
unit tests
writing 354-356
unrelated branches 88
update command 178
update operation 166
user interfaces
about 35
Bazaar Explorer, using 36
command-line client (CLI) 35

V

VCS
about 7, 13
branching 10-12
changes log, viewing 8, 9
merging 10-12
project, reverting to previous state 8
revisions differences, viewing 9, 10
VCS, migrating between
about 328
Bazaar data, exporting 330
bzr-fastimport, installing 328
fast-import files, querying 332
fast-import, filtering 332
Git data, exporting 329
other VCS data, exporting 331
subversion data, exporting 329
version control data, exporting 328, 329
version control data, importing 331, 332
version control
Bazaar Explorer, using 46-48
command line, using 45, 46
version control operations
about 38
Bazaar Explorer, using 41, 44
command line, using 40-46
differences in changed files, viewing 61, 62
directories status, checking 42
directory, managing 39
files, adding 45
files, deleting 56
files, editing 60
files from past revision, restoring 71
files, ignoring 53
files, moving 66
files, renaming 66
files status, checking 42
new revision, recording 49
revert operation 58
revision history, viewing 68
version control system. *See* **VCS**
viewing differences
Bazaar Explorer, using 64, 65
between any two two revisions 81
between revision and working tree 80, 81
between two revisions 80
checkpoint 65
command line, using 62, 63
from one revision to next 82
in changed files 61, 62
virtualenv environment 298

W

weave algorithm 159
Windows
Bazaar, installing 19
Bazaar, uninstalling 26
plugin, installing 297

workflows
branches, merging from 158, 159
common trunk, using 161
criss-cross merges, handling 159
feature branches, using 161
features 162
history, viewing 160, 161
implementing 156
independent personal branches used 157
working tree
about 32, 33
core commands, used for switching branches 280, 281
creating 153
example, setting up 279
existing working tree, removing 191
in shared repository, reconfiguring 154
lightweight checkout, used for switching to branches 283
local branch, creating without 152
remote branches, creating without 154
removing 153
reusing 278
shared repository creation, without 190
shared repository reconfiguring, without 190
switch branches, preparing for 280
switch, used for switching branches 282

Thank you for buying

Bazaar Version Control

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

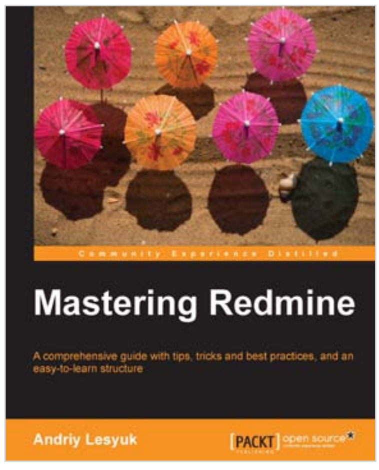
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



Mastering Redmine

ISBN: 978-1-849519-14-4 Paperback: 366 pages

A comprehensive guide with tips, tricks and best practices, and an easy-to-learn structure

1. Use Redmine in the most effective manner and learn to master it
2. Become an expert in the look and feel with behavior and workflow customization
3. Utilize the natural flow of chapters, from initial and simple topics to advanced ones

---

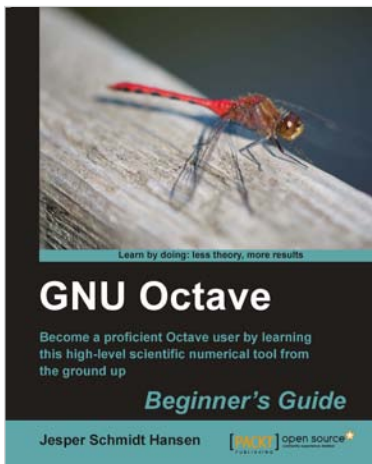
gnuplot Cookbook

ISBN: 978-1-849517-24-9 Paperback: 220 pages

Over 80 recipes to visually explore the full range of features of the world's preeminent open source graphing system

1. See a picture of the graph you want to make and find a ready-to-run script to produce it
2. Working examples of using gnuplot in your own programming language... C, Python, and more
3. Find a problem-solution approach with practical examples enriched with good pictorial illustrations and code

Please check **www.PacktPub.com** for information on our titles

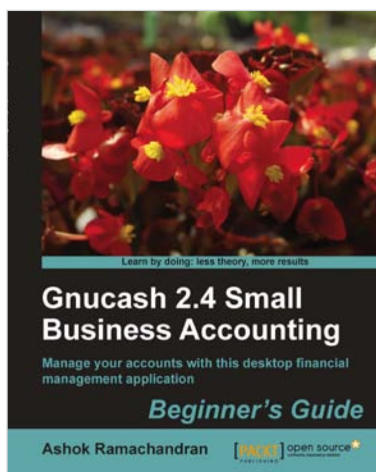


GNU Octave Beginner's Guide

ISBN: 978-1-849513-32-6 Paperback: 280 pages

Become a proficient Octave user by learning this high-level scientific numerical tool from the group up

1. The easiest way to use GNU Octave's power and flexibility for data analysis
2. Work with GNU Octave's interpreter – declare and control mathematical objects like vectors and matrices
3. Rationalize your scripts and control program flow
4. Extend GNU Octave and implement your own functionality



Gnucash 2.4 Small Business Accounting: Beginner's Guide

ISBN: 978-1-849513-86-9 Paperback: 324 pages

Manage your accounts with this desktop financial management application

1. Help small businesses maintain their books of accounts using feature-packed, easy-to-use GnuCash
2. Written by an author who has "been there, done that": who ran two small businesses, understands the needs of small businesses, and guides you to effectively use GnuCash
4. Instead of simply describing the features of the software, this book focuses on how to use this software to plan, get the right numbers, and make the decisions to reach your business goals

Please check **www.PacktPub.com** for information on our titles