version management
Improving Ruby, Patch by Patch

Rubyforge maintains a bug tracker for submitting patches and bug reports to Matz and the gang. These reports also get submitted to the Ruby-Core mailing list for discussion, so you can be sure your request won’t go unnoticed. You can also send your patches straight to the mailing list. Either way, you are encouraged to take part in the discussion that ensues.

To summarize, the steps for building a patch are:

1. If you are fixing a bug in Ruby 1.8, check out a copy of Ruby 1.8 from Subversion using the ruby_1_8 branch.

   $ svn co http://svn.ruby-lang.org/repos/ruby/branches/ruby_1_8

   If you wish to add a feature to Ruby, check out the trunk of Ruby’s source. Even if you wish to add a feature to Ruby 1.8, it has to be proven in the trunk first.

   $ svn co http://svn.ruby-lang.org/repos/ruby/trunk ruby

2. Add your improvements to the code.
3. Create a patch.
   $ svn diff > ruby-changes.patch

4. Email your patch to the Ruby-Core mailing list with a CHANGELOG entry describing the patch.

5. If there are no issues raised about the patch, committers will be given the approval to apply it.
Improving Ruby, Patch by Patch

Rubyforge maintains a bug tracker for submitting patches and bug reports to Matz and the gang. These reports also get submitted to the Ruby-Core mailing list for discussion, so you can be sure your request won't go unnoticed. You can also send your patches straight to the mailing list. Either way, you are encouraged to take part in the discussion that ensues.

To summarize, the steps for building a patch are:

1. If you are fixing a bug in Ruby 1.8, check out a copy of Ruby 1.8 from Subversion using the ruby_1_8 branch.
   
   ```bash
   $ svn co http://svn.ruby-lang.org/repos/ruby/branches/ruby_1_8
   ```

   If you wish to add a feature to Ruby, check out the trunk of Ruby's source. Even if you wish to add a feature to Ruby 1.8, it has to be proven in the trunk first.

   ```bash
   $ svn co http://svn.ruby-lang.org/repos/ruby/trunk ruby
   ```

2. Add your improvements to the code.
3. Create a patch.
   
   ```bash
   $ svn diff > ruby-changes.patch
   ```

4. Email your patch to the Ruby-Core mailing list with a CHANGETLOG entry describing the patch.
5. If there are no issues raised about the patch, committers will be given the approval to apply it.
Improving Ruby, Patch by Patch

Rubyforge maintains a bug tracker for submitting patches and bug reports to Matz and the gang. These reports also get submitted to the Ruby-Core mailing list for discussion, so you can be sure your request won’t go unnoticed. You can also send your patches straight to the mailing list. Either way, you are encouraged to take part in the discussion that ensues.

To summarize, the steps for building a patch are:

1. If you are fixing a bug in Ruby 1.8, check out a copy of Ruby 1.8 from Subversion using the ruby_1_8 branch.

   $ svn co http://svn.ruby-lang.org/repos/ruby/branches/ruby_1_8

   If you wish to add a feature to Ruby, check out the trunk of Ruby’s source. Even if you wish to add a feature to Ruby 1.8, it has to be proven in the trunk first.

   $ svn co http://svn.ruby-lang.org/repos/ruby/trunk ruby

2. Add your improvements to the code.
3. Create a patch.
   
   $ svn diff > ruby-changes.patch

4. Email your patch to the Ruby-Core mailing list with a CHANGELOG entry describing the patch.
5. If there are no issues raised about the patch, committers will be given the approval to apply it.
Improving Ruby, Patch by Patch

Rubyforge maintains a bug tracker for submitting patches and bug reports to Matz and the gang. These reports also get submitted to the Ruby-Core mailing list for discussion, so you can be sure your request won’t go unnoticed. You can also send your patches straight to the mailing list. Either way, you are encouraged to take part in the discussion that ensues.

To summarize, the steps for building a patch are:

1. If you are fixing a bug in Ruby 1.8, check out a copy of Ruby 1.8 from Subversion using the ruby_1_8 branch.
   
   `$ svn co http://svn.ruby-lang.org/repos/ruby/branches/ruby_1_8`

   If you wish to add a feature to Ruby, check out the trunk of Ruby’s source. Even if you wish to add a feature to Ruby 1.8, it has to be proven in the trunk first.

   `$ svn co http://svn.ruby-lang.org/repos/ruby/trunk ruby`

2. Add your improvements to the code.
3. Create a patch.

   `$ svn diff > ruby-changes.patch`

4. Email your patch to the Ruby-Core mailing list with a CHANGETLOG entry describing the patch.
5. If there are no issues raised about the patch, committers will be given the approval to apply it.
Generic version management terms

1. **Repository** - the database storing the files or directory trees
2. **Working copy** - a developers own local directory of files
3. **Branch** - a separate copy of the files with its own history tracking
4. **Trunk** - the main branch in the repository
5. **Head** - the current revision of a branch
6. **Add** - tells repository to begin tracking a certain file or directory
7. **Revision** - a particular version of a file (or files...)
8. **Check out** - download a particular revision of one or more files (creates local working copy)
9. **Commit or Check in** - upload one or more changed files (from local working copy) to the repository
10. **Checkin message** - a message describing what was changed
Generic version management terms

11. **Diff** - a summary of the differences between two revisions (or the difference between a revision and the local working copy)

12. **History** - a list of changes made to one or more files since they were added

13. **Merge** - bring sets of changes together (ex. merging an experimental branch into the trunk)

14. **Conflict** - when two different sets of changes clash (and the system can’t sort it out)
Version management benefits

Version control is primarily to allow you to track files or directory trees over time. This is useful for an individual programmer, but it is essential for working in teams.

1. Backup and restore - you can jump to any point in time ("I need that file as it was last Thursday").
2. Team members can share code and stay up-to-date as changes are made.
3. For each change the versioning system can track who made the change, when it was made and (optionally) a message explaining the change (why).
4. An individual can make changes (one his/her working copy) without worrying about getting in the way of other changes.
5. Some team members can experiment with a new idea without worrying about breaking the code for everyone else (using a separate branch).
Revisions and commits

CVS uses per file revision numbers (views.py 1.1, 1.2, 1.3, ...)

Subversion uses per repository wide revision numbers
Revisions and commits

CVS uses per file revision numbers (views.py 1.1, 1.2, 1.3, ...)
Subversion uses per repository wide revision numbers
Revisions and commits

In this way users can: retrieve a revision of a specific file, check out all source code exactly as it was two months ago, tell you what changed in a particular file between revision 8 and 9, etc.
Tags

Tags are a great way of keeping track of significant events in the life of the project.
Branching and merging

Now suppose you wanted to tryout some experimental changes without interrupting the rest of the project team?
Branching and merging

Now suppose you wanted to tryout some experimental changes without interrupting the rest of the project team?

**Answer:** just work on your local working copy and don’t commit until it is ready
Branching and merging

Now suppose you and a few other team mates wanted to tryout some experimental changes without interrupting the rest of the project team?
Branching and merging

Now suppose you and a few other team mates wanted to tryout some experimental changes without interrupting the rest of the project team?
Branching and merging

What issues arise when getting ready for a release (say 2.0) of your software?
Branching and merging

What issues arise when getting ready for a release (say 2.0) of your software?
Branching and merging

What issues arise when getting ready for a release (say 2.0) of your software?
Heise Security reports how Leopard Mail is vulnerable to email attachments masquerading as the wrong type. For example, a shell script named “Foo.jpg”, but which has a resource fork item adding assigning the file to Terminal, will be displayed by Leopard Mail as a JPEG image, but will open and execute in Terminal — without any warning or prompt — if you double-click it from Mail. Oddly, you do get a warning on subsequent attempts to open the attachment within Mail — it only executes in Terminal without warning the first time. **Even worse, this same vulnerability was closed by Apple before, in Tiger, but has returned in Leopard.**

How might this have happen?
Centralized versus decentralized repositories
SVN uses a centralized repository model.
Git uses a distributed repository model
Distributed topologies

With a distributed version management system, any topology (i.e., any arrangement of connections between repositories) is possible. Examples follow ... For each of the examples, think about the consequences, the tradeoffs, etc.

Here is a legend for the diagram that follows:

- A repository
- A release manager's repository
- The flow of code updates
Distributed topologies

Assume a situation where there are subgroups but there is no central repository and no central repository for each subgroup. How could things (such as releases) be managed?
Distributed topologies

Assume a situation where there are subgroups but there is no central repository and no central repository for each subgroup. How could things (such as releases) be managed?

For each subgroup, assign one person to be the local release manager. At the top level assign one person to be the project wide release manager.

Arbitrary connections could exist within groups, but the top level release manager would communicate with each subgroup’s release manager.