Patcher

Automatic maintenance of RCS-based projects from within XEmacs
Version 4.0

Didier Verna <didier@xemacs.org>

Table of Contents

Copying

1 Introduction

When a project becomes important in size, or when the development is performed cooperatively by several people across the Internet, it is a common practice to help maintaining it by using a revision control system. Such tools (Git, Mercurial, Subversion, Darcs, CVS, PRCS to name a few) usually work by maintaining a centralized or distributed project archive (also called a repository) that keeps track of the history of the changes, lets you develop different "branches" at the same time and perform operations like merges between these different project branches.

In such "RCS-based" maintenance models, making the project evolve usually involves repeatedly the same few steps, some of which can be tedious: you work on your local copy of the project; once you're satisfied with your changes, you create a patch by diffing your local copy against the project's archive; then (or progressively), you construct the ChangeLog entries. Finally, you propose your changes by sending a mail to the developers list with your patch and the ChangeLog entries included, hoping that your proposition will be accepted. If you're one of the maintainers, you will still probably send the message to the list, simply announcing the modification, and immediately commit the patch with an informative log message.

Patcher is an XEmacs package designed to automate this process. Patcher can't work on the project for you. However, as soon as you give it some basic knowledge on the project structure and repository, it can automatically build a patch by comparing your local copy with the repository, create ChangeLog entries, prepare a mail announcing the changes, and even commit the patch for you with a informative log message. All of this is done in just a few keystrokes. Additionally, Patcher can perform some sanity checks, like verifying that your local copy is up-to-date, that you did not forget some ChangeLog entries, that the commit operation went well and so on.

If you're brave and impatient, and want to start using the basics of Patcher as soon as possible, see Chapter 3 [Quick Start], page 7. It is recommended to read it anyway, since it gives an overview of how Patcher works. If you know the basics and want a more detailed guide, see Chapter 4 [User Manual], page 11.

Enjoy using Patcher!

2 Installation

2.1 Distribution

Patcher is a standard XEmacs package. As such, you can download and install it directly from a running XEmacs session. The packages interface is accessible from the '`Tools`' menu or via *`M-x list-packages`*). You may also manually download a tarball or use the Mercurial repository. See `http://www.xemacs.org/Develop/packages.html` for more information.

Otherwise, Patcher is also distributed as a standalone package directly from my website (a Git repository and tarballs are available), at `http://www.lrde.epita.fr/~didier/software/elisp/misc.php`. You will also find different inlined versions of this documentation at that place. For standalone installation instructions, please read the '`INSTALL`' file in the distribution.

2.2 Requirements

Patcher currently works with XEmacs 21.4 or later. Patcher might also have some other requirements, depending on how you use it:

- If you let Patcher create ChangeLogs for you (see Section 4.4 [ChangeLogs Handling], page 23), you will need the '`add-log`' library from the '`xemacs-base`' package, version 2.21 or later, installed on your system.
- If you want to send mails from Patcher (see Section 4.2.1 [Mail Methods], page 18), you will need a mail user agent. Patcher currently supports '`sendmail`', '`message`' and '`Gnus`' natively and through the '`compose-mail`' interface. Other MUA might be partly supported when used with `compose-mail`. Patcher will probably suffer from non critical deficiencies in that case however (it will issue warnings).

2.3 Insinuation

With a proper installation of Patcher (either way), you don't need any special trickery in your '`.emacs`' file because all entry points to the library should be autoloaded.

However, Patcher has the ability to hook into external libraries, but won't do so unless requested. Currently, Patcher has hooks for Gnus only. If you're using Gnus as your MUA, you might want to add the following line to your '`gnusrc`' file:

```
(patcher-insinuate-gnus)
```

This will add some facilities described along the text.

3 Quick Start

This chapter demonstrates the use of Patcher through a quick and basic setup. Adapt the example as you wish. See also Appendix A [XEmacs Development], page 33 for an XEmacs specific sample setup. Let's make some assumptions first:

- You own a computer.
- You have the '`add-log`' library from the '`xemacs-base`' package, version 2.21 or later, installed on your system.
- You're working on a Git project called SuperProj.
- Your local clone of the Git repository is located in '`/home/me/superproj`'.
- You have commit access to this project.
- There is a mailing list for developers at `superproj-devel@superproj.org`.
- Your repository is up-to-date, but you've done some hacking in the sources that you'd like to commit.
- Since you're lazy, you didn't write the ChangeLog entries yet.

3.1 Setting up Patcher

The first thing to do is to make patcher aware of your "SuperProj" project. Put this in your '`.emacs`' file:

```
(setq patcher-projects
        '(("SuperProj" "/home/me/superproj"
        :to-address "superproj-devel@superproj.org"
        :commit-privilege t
        :themes (git))))
```

As you can imagine, `patcher-projects` is a user option in which you store information about the projects you want to manage with Patcher. It is actually a list of what's called *project descriptors*. Here's the meaning of the only project descriptor we have in the example above: we have a project named "SuperProj", located in '`/home/me/superproj`' and for which emails should be sent to `superproj-devel@superproj.org`. In addition to that, this project is handled by Git.

Note the particular syntax for specifying the mailing address. This is what's called a *project option*. Contrary to the project's name and directory, which are mandatory and always appear as the first and second elements of a project descriptor, project options are optional and can appear in any order. Note also that we have used a `:themes` option for specifying the revision control system in use. A "theme" is a set of options with particular values. Patcher happens to come with some predefined themes, including one for Git.

3.2 Calling Patcher

Now you want to build a patch with your changes, and prepare a message to submit them. The way Patcher works is currently to setup the message first, and then to control all subsequent operations from there. In other words, to create a patch, you actually ask Patcher to prepare a mail. Type this:

```
M-x patcher-mail
```

First, you're prompted (with completion) for a project name (the first element of each project descriptor, remember?). We currently only have a "SuperProj" project, so hitting *TAB* will directly fill the minibuffer in with this only choice. Then, you're prompted for a subject line that will be used in the mail. Say something sensible here.

Three operations are now executed in turn:

1. Patcher prepares a mail buffer. The message will be sent to the address you specified with the `:to-address` project option, and the subject line now reads "[PATCH] something sensible here".
2. Patcher now builds the patch. The command used to do this is specified in the Git theme, but it is a project option so it can be changed. Upon successful completion of this command (we assume that's indeed the case), the patch is inserted into the mail buffer. Some information about the patch is provided just above it (the command used, the files affected and so on).
3. Finally, Patcher generates ChangeLog skeletons from what it understands of the patch. This involves visiting the appropriate ChangeLog files, and creating initial entries.

3.3 Filling the ChangeLogs

Patcher has just created initial ChangeLog entries for you. You must now browse through the ChangeLog file(s) and fill the entries as you see fit. From the mail buffer type *C-c C-p n* (`patcher-mail-first-change-log`). This command will bring you to the first ChangeLog file that you need to fill in. From a ChangeLog buffer, the same keyboard sequence will will bring you to the next one, and so on (`patcher-change-log-next`).

Once you're done, you can very well save the ChangeLog buffers. However, don't kill them! Don't even think about it. Patcher still needs them. From any of the ChangeLog buffers you just filled in, type *C-c C-p m* (`patcher-change-log-mail`). This will bring you back to the mail buffer.

3.4 Filling the message

Now that you're satisfied with your ChangeLog entries and you've returned to the mail buffer, you want to write some explanation text in the message. I'll let you do that. You also want to insert the ChangeLog entries corresponding to your patch, since they are usually much more readable than the patch itself.

Inserting your ChangeLog entries in the mail buffer is as simple as typing *C-c C-p l* (`patcher-mail-insert-change-logs`). This command places them just above the patch, with a short information line (per ChangeLog file) on top.

3.5 Committing the Patch

If you have commit access to your project, you should read this. Otherwise, you may directly jump to Section 3.6 [Sending the Message], page 9.

Committing your changes involves three steps: preparing the commit command, preparing the commit log message, and actually committing the changes. Although Patcher can do all of this in one shot, it gives you control each step by default.

In order to start the commit process, simply type *C-c C-p c* (`patcher-mail-commit`). Congratulations. You've just been transported to a new buffer, the "log message" buffer. This buffer lets you edit the log message that will accompany your commit. Note that the message is initialized with the subject line of your mail. This is also a project option.

Once you're satisfied with the log message, type *C-c C-p c* or *C-c C-c* (`patcher-logmsg-commit`). This command computes the commit command to use, and while you think that you're done this time, you're not quite there yet. Indeed, patcher transports you to yet another buffer called the "commit command" buffer. This buffer lets you modify, or at least check the commit command to use.

The default commit command is specified in the Git theme, but it is of course a project option so it can be changed. Note that Patcher stores the log message in a temporary file and uses the '`-F`' option of the Git '`commit`' command. Finally, note that Patcher has automatically appended the affected ChangeLog files to the commit command.

If the commit command suits you, type *C-c C-p c* or *C-c C-c* (`patcher-cmtcmd-commit`). This time, you're done. If you had not previously saved the ChangeLog files, Patcher will do it for you just before committing.

As Patcher doesn't do pushing (neither pulling) yet, you may now want to push your changes to the remote repository by hand.

3.6 Sending the Message

Sending the message has actually nothing to do with Patcher. It depends on the method you use for sending mails, but will usually be done via a *C-c C-c* command of some sort. On thing to note however: if you've committed your changes via Patcher, the message has been slightly modified: the subject line now reads "[COMMIT] something sensible here" instead of "[PATCH] something sensible here", and a short commit notice has been inserted just at the beginning of the message's body.

That's it. That was easy. Congratulations on your first shot at Patcher, anyway! Of course, Patcher is much more powerful and customizable than what has been described in this chapter. For a complete documentation on how to use and customize Patcher, please refer to Chapter 4 [User Manual], page 11.

4 User Manual

This chapter provides a step-by-step guide to using Patcher. Everything there is to know about Patcher is here, though the features are introduced progressively.

All user options that are going to be presented in this manual can be found in the `patcher` customization group, or a subgroup of it.

At any time, and in any buffer related to a Patcher project (mail, ChangeLog *etc.*), you can query the current version of Patcher by calling the function `patcher-version`, bound to *C-c C-p v*.

4.1 Starting Up

Starting up Patcher implies first defining a project, and then calling one of the entry point functions. This section describes how to do that.

4.1.1 Project Descriptors

Projects specifications are stored in `patcher-projects`. This user option is actually a list of *project descriptors*. Each project descriptor has the following form: '`(`*NAME DIR :OPTION VALUE* `...)`'

- *NAME* is a string naming your project.
- *DIR* is a string specifying the directory in which to find the project. It can also be set to `nil` in which case Patcher will prompt you for the project's location every time it is needed (such projects are called "floating" projects). This feature may be useful when you maintain several clones of the same repository but want to define it only once in Patcher. Another potential use of this is when several independent projects happen to share exactly the same set of options.
- The remainder of a project descriptor is a sequence of zero or more option/value pairs that we call *project options*. All option names start with a colon. The type of a value depends on the corresponding option. For example, there is a project option named `:to-address`, whose value should be a string giving the email address to which you want to send Patcher messages.

When Patcher needs the value for a particular project option, it looks for it directly in the project descriptor, but also in other places. This process is described below.

4.1.1.1 Themes

If you have several projects sharing the same option set, you might want to setup a theme. Themes are named collections of project options.

Themes are stored in the `patcher-themes` user option. This option is a list of themes. Each theme has the following form: '`(`*NAME :OPTION VALUE* `...)`'.

NAME is the theme's name (a symbol). The remainder of the list is a sequence of zero or more option/value pairs, just like in project descriptors.

In order to use a theme in a given project, a `:themes` project option is provided. It is a list of theme names (symbols). Use this option in your project descriptor, and the project will implicitly inherit all options from the corresponding theme.

One important note: as `:themes` is a project option, it can appear in a theme. In other words, themes can inherit from other themes. When Patcher tries to retrieve an option from a theme (an that option is not directly available), the themes tree is traversed depth first.

Because themes can contain themes, a bogus setting might lead to an infinite loop (a cycle in a theme graph). To prevent this, the `patcher-max-theme-depth` user option is provided. It represents the expected maximum theme nesting level and defaults to 8.

Patcher comes with a set of built-in themes for several revision control systems. These are Git, Mercurial (Hg), Darcs, Subversion (Svn), CVS and PRCS. Look at the value of `patcher-built-in-themes` to see what's in them. Each of these themes have a `-ws` counterpart which eliminates white-space differences in diff outputs. This comes in handy if you are a committer (see Section 4.5 [Project Check In], page 27) and you perform some kind of automatic white-space cleanup in the files you edit, especially when you let Patcher generate the ChangeLog entries (see Section 4.4 [ChangeLogs Handling], page 23).

While you can't modify the value of `patcher-built-in-themes`, you're free to do whatever you want in `patcher-themes`, including creating a theme with the same name as a built-in one. This new theme will take precedence over the other. Having this built-in variable (a constant, actually) lets me modify its value from release to release without risking to smash your own adjustments.

4.1.1.2 Project inheritance

When two projects are very similar, you might prefer to use the project inheritance mechanism described below over themes.

There is a special project option called `:inheritance`. This option must be a list of project names (strings). The inheritance of a project defines a list of projects from which to inherit options.

One important note: inherited projects might have their own `:inheritance` option set to other projects in turn. In other words, the project inheritance can be more than one level deep. Just as for themes traversal, when Patcher tries to retrieve an option and this option is not available directly, the inheritance tree is traversed depth first.

Because inherited projects can inherit from projects, a bogus setting might lead to an infinite loop (a cycle in a project graph). To prevent this, the `patcher-max-inheritance-depth` user option is provided. It represents the expected maximum project inheritance level and defaults to 8.

The `:inheritance` project option is somewhat special in the sense that it can't appear in a theme. We will encounter other exceptions later in this manual.

4.1.1.3 Fallbacks

For each existing project option, Patcher also has a *fallback* user option with a default value that would be shared among all projects not setting the option explicitly. The name of the fallback is obtained by replacing the colon in the project option's name with the prefix `patcher-default-`. For example, the fallback corresponding to the `:to-address` project option is named `patcher-default-to-address`.

The `:inheritance` project option is also special in the sense that it doesn't have a corresponding fallback. We will encounter other exceptions later in this manual.

In the remainder of this manual, we will rarely mention the fallbacks again. When we introduce a new project option, just remember that it always has a corresponding fallback (well, not always, as you just discovered).

4.1.1.4 Retrieval

When Patcher needs the value of a particular project option, it looks for it in the following manner:

- First, it looks directly in the project descriptor to see if the option is given.
- If that fails, it next tries the given themes, if any. This involves recursively traversing the project's themes tree. Options successfully retrieved in themes are said to be *themed*.
- If that still fails, it then tries the inherited projects, if any. This involves recursively traversing the project's inheritance tree. Options successfully retrieved in inherited projects are said to be *inherited*. Note that in turn, such options could have been actually themed in the inherited project.
- If that fails again, it finally falls back to the value given in the corresponding fallback (if it exists). In such a case, the option is said to be *fallbacked*.

Note that a value of `nil` for a project option **is an actual value**. It is not equivalent to an option being unset. As a consequence, if Patcher finds a project option with a value of `nil` somewhere, it will use it and stop the search, even if a non `nil` value could be retrieved later from a theme, an inherited project or a fallback. This provides you with a way to annihilate themed, inherited or fallbacked options.

The retrieval process is completely dynamic. In particular, this means that even if you already have a running Patcher instance, you can still modify the project's options, and these modifications will be taken into account in your running instance. In fact, the only thing you can't do with a running Patcher instance is modify the project's name.

Beware however that modifying an option while a corresponding project has been instantiated is not very safe, and should be avoided as much as possible.

4.1.1.5 Inheritance or theme?

Let us summarize the four available ways to provide an option for a project: a direct setting in the project descriptor, a global default value in the fallback user option, plus themes and inherited projects.

At that point, you might be wondering why the themes and inheritance concepts were both designed, since they actually perform very similar tasks. Good question. Here is a good answer.

Projects might share options for different reasons. For example, my "XEmacs" (source) and "XEmacs Packages" projects share many options (`To:` address, `From:` address, diff and commit commands and so on) because they both relate to XEmacs. On the other hand I have personal but totally unrelated projects that share the same commands because they are all handled through a common system: Git.

In other words, you should rather use the inheritance mechanism when projects relate to each other, and the theme mechanism for settings that are orthogonal the projects they apply to.

4.1.2 Entry Points

Patcher currently uses the mail buffers as "master" buffers for controlling all operations: building a patch, creating the ChangeLog entries, committing... all is done from the mail buffer. Note however that you don't need to actually send mails to use Patcher (see Section 4.2.1.2 [Fake Mail Method], page 19).

To use Patcher on a certain project, you start by preparing a (possibly fake) mail. There are several ways to do so: you could start a brand new message, "adapt" a message already in preparation to Patcher, or even compose some sort of a Patcher reply to another message.

At any time from a mail buffer, you may change your mind and decide that starting Patcher was a mistake. You can then call the function `patcher-mail-kill`, bound to *`C-c C-p k`*, and Patcher will "kill" the current project, cleaning up the place like Patcher had never existed before.

4.1.2.1 Mail Creation

Creating a message is done with the following function.

`patcher-mail` [Function]

Start composing a brand new Patcher message. This function interactively prompts you for the name of the project and for a (mail) subject line. It also performs a global diff of your project.

4.1.2.2 Mail Adaptation

Assuming that you are already editing a message (with your usual MUA), you can adapt it to Patcher. This might be useful if you want to reply to a normal message with a Patcher mail and your MUA is unknown to Patcher (see Section 2.3 [Insinuation], page 5). Start by creating the reply, and then adapt it to Patcher.

`patcher-mail-adapt` [Function]

Adapt an existing message to Patcher by prompting you for the name of a project and possibly a new subject. This function also performs a global diff of your project.

When adapting a message to Patcher, you are always prompted for a new subject line, although you can just hit *`Return`* to leave it empty. If there is indeed a subject change (that is, if there is both an old subject and a new one), Patcher uses a project option called `:subject-rewrite-format` to modify the subject line. The subject rewrite format is a string in which a '`%s`' is replaced with the new subject, while a '`%S`' is replaced with the old one.

By default, the subject rewrite format is '`"%s (was: %S)"`'. Note that the subject prefix (see Section 4.2.2 [Message Customization], page 20) is added in front of the subject line *after* the subject has been rewritten.

4.1.2.3 Gnus Insinuation

If you're using Gnus to read mail and have properly insinuated it (see Section 2.3 [Insinuation], page 5), Patcher offers different Gnus-like ways to answer mails and adapt them to Patcher. All the functions below are available from both the Gnus Summary and Article buffers.

`patcher-gnus-summary-followup` [Function]
Compose a followup to an article, and adapt it to Patcher. This function is bound to *C-c C-p f*.

`patcher-gnus-summary-followup-with-original` [Function]
Idem, but also cite the original article. This function is bound to *C-c C-p F*.

`patcher-gnus-summary-reply` [Function]
Like `patcher-gnus-summary-followup`, but compose a reply. This function is bound to *C-c C-p r*.

`patcher-gnus-summary-reply-with-original` [Function]
Idem, but also cite the original article. This function is bound to *C-c C-p R*.

4.1.3 Project Relocation

Earlier (see Section 4.1.1 [Project Descriptors], page 11), we talked about floating projects (those having a null directory). There might also be times when you want to temporarily relocate a non-floating project (for instance just this once, without modifying the project descriptor). You can relocate a project by calling any of the entry point functions with a prefix of 1 (`C-u 1`).

Since people have a tendency to keep clones under the same umbrella directory, it seems convenient to start prompting you for the relocation directory under the parent of the project's original directory. Patcher does that.

As previously mentioned for floating projects (see Section 4.1.1 [Project Descriptors], page 11), an interesting side effect of relocation is that it allows you to use one particular project descriptor for another, completely independent project which would happen to use exactly the same set of options.

4.1.4 Subprojects

As mentioned before (see Section 4.1.2 [Entry Points], page 14) the entry point functions all perform a global diff of your project just after having prepared the mail buffer. There might be times, however, when you want to work on a project subset only (a specific set of files or directories), for instance in order to commit only a few of the current changes. This concept is know to Patcher as working on a "subproject".

A subproject is essentially defined by the project on which it is based, an optional subdirectory in which the whole subproject resides and an optional set of specific files to work on, in that subdirectory.

Warning: for technical reasons (and also because right now I don't want to clutter Patcher's code too much with workarounds for deficient RCSes), it is not possible to define Mercurial subprojects with a specific subdirectory. This problem will go away when issue 2726 is resolved (`http://mercurial.selenic.com/bts/issue2726`).

When you provide an explicit set of files to work on, it is not necessary (it is even forbidden) to specify the ChangeLog files. Patcher will automatically find them for you. In other words, only specify source files, not ChangeLog files.

Patcher offers to ways of working on subprojects: either temporarily or by defining them in a permanent fashion.

4.1.4.1 Temporary Subprojects

In order to work on a temporary subproject, call any of the entry point functions (see Section 4.1.2 [Entry Points], page 14) with a simple prefix argument (`C-u`). Patcher will then prompt you for an optional subdirectory and for a specific set of files to work on, under that particular subdirectory. There, you can in fact specify files as well as directories, use wildcards, just as you would construct a shell command line diff.

Note that since the files you provide can in fact be directories, you can circumvent the Mercurial limitation mentioned above by *not* providing a specific subdirectory, but instead give it as a file at the second prompt. This workaround also applies to permanent subprojects, as described in the next section.

4.1.4.2 Permanent Subprojects

If you happen to work more than once on the same project subset, it will quickly become annoying to have to specify explicitly the same subdirectory and/or files over and over again. Consequently, Patcher offers you a way to permanently define subprojects.

Defining Subprojects

The user option `patcher-subprojects` stores a list of *subproject descriptors*. A subproject descriptor is almost the same as a project descriptor, with a few exceptions:

- Instead of the project directory field (the second field in a project descriptor), you rather specify the name of the project this subproject is based on.
- In addition to the standard project options we've already seen, two subproject options are available:

  `:subdirectory`
  : This lets you specify a subdirectory of the original project's directory in which the whole subproject resides. This subdirectory must be provided *relative* to the original project's directory.

  `:files`
  : This lets you specify a list of files or directories composing the subproject. Each file specification may be provided *relative* to the subdirectory above, if any, or to the original project's directory. They may also contain wildcards.

  Please note that these subproject options have no corresponding fallback (that would be meaningless). They can't appear in a theme either.
- Subprojects don't have an `:inheritance` mechanism. Instead, they implicitly inherit from their base project (which in turn can inherit from other projects).

Here are some important remarks about permanent subprojects:

- Permanent subprojects are accessible in exactly the same way as normal projects, that is, via the entry point functions (see Section 4.1.2 [Entry Points], page 14). A subproject *is* a project, after all. Because of that, projects and permanent subprojects can't share names. Patcher always looks for subprojects first, and then regular projects.
- A subproject with neither a `:subdirectory` nor a `:files` option is exactly the same as the base project, apart from project options that you would override. This can hence be seen as an elegant (or kludgy, depending on the viewpoint) way to define project "variants".

- Since Patcher doesn't really make a distinction between projects and subprojects, it is possible to work on a temporary subproject based itself on a subproject: simply call one of the entry point functions with a simple prefix argument, and select a permanent subproject when prompted. The effect is then to work on a subsubproject: you can specify an optional subsubdirectory and override the set of files affected by the patch.
- Finally, note that it is even possible to both relocate a project *and* work on a temporary subproject. In order to do that, use a prefix argument of -1 instead of just 1 (`C-u -1`). And now, try to imagine the brain damage that is caused by using a prefix of -1 and then select a permanent subproject as the base project. The effect is to work on a sub-sub-relocated project. . . .

Project Naming

As you already know, Patcher distinguishes (sub)projects by their *NAME* field in the `patcher-projects` and `patcher-subprojects` user options. This name is meant to be explicit and convenient for the user to read. However, some RCSes like (the decidedly weird) PRCS require the *actual* project name in their commands. It would then be difficult to define project variants for the same directory but with different names.

To remedy this problem, patcher provides a `:name` project option. If set, it will be used by diff and commit commands instead of the project's name when necessary. See Section 4.3.1 [Diff Command], page 21 for details on how to do that.

Command Directory

Most of the revision control systems out there can perform in any of the project's subdirectories. For that and other technical reasons, Patcher will normally execute all commands in the specified (sub)directory of the specified (sub)project. This principle does not always hold however. For example, PRCS (weird, did I mention it already?) can only work in the project's root directory.

If you want to define projects for which the revision control system can be executed in only one directory, Patcher provides you with the `:command-directory` project option (a string). This directory must be provided relative to the project's directory (but note that it must usually go upwards).

All commands (diff and commit ones) will be executed from there. Also, note that the command directory does not change the way you might specify files. Patcher modifies all needed paths automatically to handle the command directory properly. This means that you should continue to specify files relatively to the (sub)project's (sub)directory, regardless of the existence of a command directory.

When needed, a command directory should always be specified in a project (in which case it will most likely be the same as the project's directory) and never in subprojects. Otherwise, temporary subprojects would fail to get it.

4.1.5 Submodules

Related to the notion of subproject is that of "submodule" (or "subrepo") as some RCSes would put it. A submodule is a standalone project that appears under another project (so it looks like a subproject, only it isn't).

Normally, there is nothing special about submodules, in the sense that if you want to handle them from Patcher, you would define them as regular projects. However, there are ways to detect submodules automatically, which can be very convenient if you have plenty of them (this happens for XEmacs packages for instance).

Patcher currently knows how to detect submodules of Mercurial and Git projects. The effect is to define new projects that inherit from the umbrella project automatically, with their own name and directory, so that you don't need to define them by hand.

Automatic detection of submodules is controlled via the `:submodule-detection-function` project option. Its value is a symbol naming a function, or `nil` if you don't want autodetection. The built-in Git and Mercurial themes set this option to `patcher-hg-detect-submodules` and `patcher-git-detect-submodules` respectively.

Submodules are detected automatically by scanning the value of `patcher-projects` the first time you use Patcher in an XEmacs session. If you further modify this variable, it may be needed to recompute the list of known submodules. You can do this by calling `patcher-detect-submodules` interactively.

4.1.6 Patcher Instances

The concept of subproject brings up the question of having Patcher working on different patches at the same time. It is possible under some conditions:

- You can have as many simultaneous Patcher instances as you want on projects that don't overlap.
- You can have as many simultaneous Patcher instances as you want on the same project, as long as there is no overlapping between each subproject. This means that you can't have source files, or even ChangeLog files in common.
- It is also possible, to some extent, to work simultaneously on overlapping instances of Patcher, although this is mostly uncharted territory. More precisely, Patcher keeps track of which project(s) refer to specific source or ChangeLog files, and knows how to associate a particular ChangeLog entry with a particular project. However, Patcher does not support interactive selection of patches (à la Darcs or Git) yet, and if you commit one of two overlapping projects, you will most likely need to rediff the other one.

4.2 Message Generation

Patcher starts working on the project (by first creating the patch) after the message is prepared. Because of this, we'll start by reviewing the mail-related customizations you might want to setup.

4.2.1 Mail Methods

Since there are different mail packages working in XEmacs, Patcher supports different methods for preparing messages. You can specify the method you prefer in the `:mail-method` project option. The value must be a symbol.

4.2.1.1 Standard Mail Methods

Patcher currently supports '`sendmail`', '`message`' and '`Gnus`' natively and through the '`compose-mail`' interface. Other MUA might be partly supported when used with `compose-mail`. Patcher will probably suffer from non critical deficiencies in that case however (it will issue warnings).

`compose-mail`
: This is the default. It is implemented via the function `patcher-mail-compose-mail` which calls `compose-mail` to prepare the message. If you are not familiar with '`compose-mail`', you might also want to throw an eye to the user option `mail-user-agent`. If your project does not specify an address to send the message to (see Section 4.2.2 [Message Customization], page 20), it is prompted for.

`sendmail`
: A direct interface to the `mail` function from the `sendmail` package. It is implemented via the function `patcher-mail-sendmail`. If your project does not specify an address to send the message to (see Section 4.2.2 [Message Customization], page 20), it is prompted for.

`message`
: A direct interface to the `message-mail` function from the `message` library (it is part of `Gnus`). It is implemented via the function `patcher-mail-message`. If your project does not specify an address to send the message to (see Section 4.2.2 [Message Customization], page 20), it is prompted for.

`gnus`
: A direct interface to the `gnus-post-news` function from the `Gnus` package (it can also send mails...). It is implemented via the function `patcher-mail-gnus`. This mail method is interesting when you maintain a special mail group for messages that you send with Patcher, most probably because they are sent to some mailing-list, such as `xemacs-patches@xemacs.org`.

  This method uses a Gnus group name and acts as if you had type '`C-u a`' on that group in the `*Group*` buffer, hence honoring the group parameters and posting-styles. If your project does not specify a Gnus group name (see Section 4.2.2 [Message Customization], page 20), it is prompted for.

This last mail method is special in the sense that it requires a running Gnus session to work. If that's needed, Patcher can start Gnus for you in several ways, according to the following user options:

`patcher-mail-run-gnus`
: If `nil`, Patcher will never start Gnus and abort the operation instead. If `t`, Patcher will always start Gnus when needed. If `prompt`, Patcher will ask you what (you want) to do. This is the default behavior.

`patcher-mail-run-gnus-other-frame`
: Used when Patcher has to start Gnus by itself. If `nil`, continue using the current frame. If `t`, start Gnus in another frame (this is the default). If `follow`, start Gnus in another frame, and use this new frame to prepare the Patcher mail.

4.2.1.2 Fake Mail Method

At that point, you might be wondering why the mail method is a project option and not simply a user option, since you probably only use one mail agent at all. Right. But you might one day work on projects for which you don't need to send messages at all. This could happen if you start using Patcher on a project of your own for instance. For that reason, there is a `fake` mail method available. It is implemented via the `patcher-mail-fake` function and calls no particular mail user agent. Once you type *`C-c C-c`* to virtually send the fake message, it only performs some cleanup.

All right. But did we really need this fake method? I mean, one could use the usual mail method, and simply not send the message in the end. Huh, yeah, ok... Actually, it is very probable that in a future release of Patcher, the mail buffer won't be the master buffer anymore, and mail sending will be just another optional step in the process. In that case, the mail method is likely to move away from project option to standard user option.

4.2.1.3 Other Mail Methods

If you're not satisfied with the provided mail methods (want a `vm` one?), you can provide your own, more or less (patches welcome if you do so). Here's what to do: set your `:mail-method` project option to, say, `foo`, and write your own function which must be named `patcher-mail-foo`.

This function must take two arguments (a project descriptor and a string containing the subject of the message), and prepare a mail buffer. If you want to do this, you should see how it's done for the built-in methods.

Note that the mail adaptation facility won't be available for your custom method. For that, you would have to hack the internals of Patcher.

4.2.2 Message Customization

When preparing a message, Patcher can fill some parts of it for you. Here's a list of mail-related project options.

`:user-name`
: The name (your name) to use when composing the message. It will affect the `From:` header. This option is used by all mail methods but `fake`. If not given, `user-full-name` is used.

`:user-mail`
: The mail (your mail) address to use when composing the message. It will affect the `From:` header. This option is used by all mail methods but `fake`. If not given, `user-mail-address` is used.

`:to-address`
: The address to send messages to (a string). This option is used by all mail methods but `gnus` and `fake`. If not given, it is prompted for when calling `patcher-mail`.

`:gnus-group`
: The Gnus group name to use for posting messages (a string). This option is used only by the `gnus` mail method. If not given, it is prompted for when calling `patcher-mail`.

  Note that if you configured your name and mail in Gnus, for instance through posting styles, these configurations take precedence over the corresponding Patcher options.

`:subject-prefix`
: A prefix for the subject line of messages. It can be `nil` or a string. By default, "[PATCH]" is used. This part of subjects is never prompted for. The subject prefix understands '`%n`' and '`%N`' substitutions. See Section 4.3.1 [Diff Command], page 21, and ⟨undefined⟩ [Project Naming], page ⟨undefined⟩ for more

information. Also, a space is inserted between the prefix and the remainder of the subject, when appropriate.

`:subject` A default value for prompted subjects (a string). Please note that this is used **only** to provide a default value for prompted subjects. Subjects are **always** prompted for. The subject understands '`%n`' and '`%N`' substitutions. See Section 4.3.1 [Diff Command], page 21, and ⟨undefined⟩ [Project Naming], page ⟨undefined⟩ for more information.

`:mail-prologue`
A prologue to insert at the top of a message body (a string).

4.3 Patch Generation

Patcher creates patches by diffing your local copy of the project against the repository. This is done automatically after preparing a message, so you shouldn't normally bother to do it manually. There are however situations in which you need to diff your project again, for instance if an error occurred during the initial diff (see Section 4.7 [More On Commands], page 32), or if you suddenly decided to further modify the source files.

The way to regenerate the patch manually is to call `patcher-mail-diff` from the mail buffer. This function is bound to *`C-c C-p d`* in this buffer.

When possible, Patcher also tries to check that your project is up-to-date with respect to the archive, and will inform you otherwise.

In the remainder of this section, we describe the different ways to customize the diff process and the appearance of its output.

By the way, (re)generating the patch does not necessarily mean that it is directly inserted into the mail buffer. This also depends on the ChangeLogs behavior (see Section 4.4 [ChangeLogs Handling], page 23).

4.3.1 Diff Command

The diff command used to generate the patch is specified by the `:diff-command` project option. You can also punctually change this command by calling `patcher-mail-diff` with a prefix argument. Patcher will then prompt you for a new command and use it exclusively for this particular patch.

By the way, don't use the prefix argument of `patcher-mail-diff` as a way to specify files (that is work on a subproject). It is not meant for that. It is meant only to modify the diff command for this instance only, not the files to which it applies.

The diff command is in fact a template string that supports dynamic expansion for a set of special constructs. The following ones are currently available.

`%n` A '`%n`' will be replaced with the project's name, that is, either the value of the '`:name`' option (see ⟨undefined⟩ [Project Naming], page ⟨undefined⟩) or the name of the project descriptor. This may be useful in commands with weird options syntax, like PRCS.

`%N` If you want to use the project descriptor's name, regardless of the value of the `:name` option, use `%N` instead of `%n`.

`%f` A '`%f`' will be replaced with explicitly diff'ed files and their accompanying ChangeLog files if any, or will simply be discarded for a global diff.

`%?f{STR}` If there are explicitly diff'ed files, this construct will be replaced by '`STR`'. Otherwise, it will simply be discarded.

`%!f{STR}` This is the opposite of the previous one: if there are explicitly diff'ed files, this construct will be discarded. Otherwise, it will be replaced by '`STR`'.

Here is an example to clarify this: the default diff command for Git in the '`git`' built-in theme (see Section 4.1.1.1 [Themes], page 11) is the following:

'`git diff --no-prefix HEAD%?f{ -- }%f`'

One important note: all diff commands in Patcher **must** have a '`%f`' construct somewhere, even if you always perform global diffs only (but in fact, you never really know that for sure). The reason is that there are situations in which Patcher may need to diff specific files, even for a global diff.

See also Section 4.7 [More On Commands], page 32 for cases where a diff command would fail.

4.3.2 Diff Headers

When Patcher generates a diff, it needs to associate every hunk with the corresponding source file (and possibly with the corresponding ChangeLog file as well). Unfortunately, different revision control systems might generate different diff outputs, making this association difficult to establish.

Patcher provides a `:diff-header` project option to help. Its value is of the form `(REGEXP NUMBER1 NUMBER2)`. `REGEXP` is used to match the beginning of a diff output while NUMBER1 and NUMBER2 are the parenthesized levels in which to find the corresponding old and new file names.

When the change involves modifying a file's contents, the old and new file names will be the same. However, they can be different in several situations, like when a file is renamed, created or deleted. In case of creation or deletion, some revision control systems use "`/dev/null`" to denote a virtual old or new file.

If you want to see some examples, have a look at the built-in themes in `patcher-built-in-themes` (see Section 4.1.1.1 [Themes], page 11). They contain presets for different revision control systems, along with suitable `:diff-header` options.

Also, you should pay attention to the fact that the values of the `:diff-header` and `:diff-command` options may depend on one another to work properly. For instance, the diff output of Mercurial looks different when you use the `--git` option.

4.3.3 Diff Line Filter

When generating a global diff, that is, without specifying the files affected by the patch explicitly, some uninformative lines might be present in the output. A typical example occurs in CVS: it indicates files present in your local copy but otherwise unknown to the server with a question mark in diff outputs.

Patcher has a project option named `:diff-line-filter` that lets filter out such unwanted lines. This must be a regular expression matching a whole line. Caution however: do not put beginning or end of lines markers in your regexp. Patcher will do it for you.

4.3.4 Diff Prologue

Patcher can (and does) insert a special prologue just above a patch in the message in preparation. This prologue gives information such as the diff command used, the files affected and so on.

The function used to generate this prologue can be specified with the `:diff-prologue-function` project option. A value of `nil` means don't insert any prologue. By default, the internal function `patcher-default-diff-prologue` is used. If you want to provide your own, here's how to do it.

Your function should take two mandatory arguments: `name` and `kind`. `name` is the name of the project and `kind` is the kind of diff. Possible values for the `kind` argument are:

`:sources` indicates a source diff only,

`:change-logs`
: indicates a ChangeLog diff only,

`:mixed` indicates a diff on both source and ChangeLog files.

Your function should also accept the following set of Common Lisp style keyword arguments (take a look at the provided function if you don't know how to do this). These arguments will be bound when appropriate, according to the kind of diff being performed.

`source-diff`
: the command used to create a source diff,

`change-log-diff`
: the command used to create a ChangeLog diff,

`source-files`
: sources files affected by the current patch,

`change-log-files`
: ChangeLog files affected by the current patch.

In the case of a mixed diff, a `nil` value for `change-log-diff` indicates that the same command was used for both the source and ChangeLog files.

Finally, your function should perform insertion at the current point in the current buffer.

4.4 ChangeLogs Handling

ChangeLogs management in Patcher involves two aspects: how ChangeLog entries are created, and how they appear in the messages. Both aspects can be customized beyond your craziest dreams.

It is possible to kill a project from a related ChangeLog file by using the same binding as in the mail buffer: *C-c C-p k* (`patcher-change-log-kill`).

4.4.1 ChangeLogs Naming

By default, Patcher thinks that ChangeLog files are named "ChangeLog". That is very clever, but if for some obscure reason that is not the case in your project, you can change this by setting the `:change-log-file-name` project option (a string).

4.4.2 ChangeLogs Updating

The way Patcher deals with ChangeLogs is controlled via the `:change-logs-updating` project option. Its value (a symbol) must be one of `automatic` (the default), `manual` or `nil`.

4.4.2.1 Automatic ChangeLogs

Automatic ChangeLogs mode is the default. Each time you (re)generate a diff, Patcher (re)creates ChangeLog skeletons in the appropriate ChangeLog files, by analyzing the generated diff. You then need to fill the entries manually.

Note that when Patcher creates skeletons for you, you should **never** kill the ChangeLog buffers while a project is running. Otherwise, Patcher will loose track of what it has or has not generated.

ChangeLog skeletons are not generated by Patcher directly, but rather by the function `patch-to-change-log` from the `add-log` library, itself from the `xemacs-base` package. This function supports only standard and CVS diff, in unified format.

For revision control systems that output something different, Patcher provides a `:diff-cleaner` option. This option names a function that will be used to "cleanup" the diff (so that it looks like a standard one, just before calling `patch-to-change-log`.

Patcher comes with a generic cleaner function named `patcher-default-diff-cleaner` which is used by default and works correctly with Git, Mercurial, Darcs and PRCS, as long as you use the corresponding built-in themes (see Section 4.1.1.1 [Themes], page 11), or in fact, as long as the corresponding `:diff-header` option is correct (see Section 4.3.2 [Diff Headers], page 22).

Patcher has two project options that give you some control on the generated ChangeLog skeleton: `:change-logs-user-name` and `:change-logs-user-mail`. As you might expect, these are strings defining your name and mail address for ChangeLog entries'headers. When `nil`, Patcher falls back to (respectively) the `:user-name` and `:user-mail` project options. If in turn set to `nil`, Patcher lets the function `patch-to-change-log` decide what to use (most probably what the user options `user-full-name` and `user-mail-address` say).

Normally, you don't modify source files when working with Patcher. However, ChangeLog files need update and saving in automatic mode. Patcher provides two hooks for plugging in additional processing on ChangeLog files.

- `:link-change-log-hook` This hook is run every time Patcher "links" a ChangeLog file to a project. Linking a ChangeLog file in this context means figuring out that it is involved in the current patch. Every function in this hook will be given the ChangeLog file name (relative to the project's directory) as argument. Also, it is guaranteed that when this hook is run, the current directory (in whatever the current buffer is) is set to the project's directory.
- `:after-save-change-log-hook` This hook is run every time you save a ChangeLog file. The functions in this hook are executed in the ChangeLog's buffer. To be honest with you, I didn't invent anything here, and I must confess that this is not a real hook. Instead, what you specify in this project option is simply added to the ChangeLog's local after-save-hook.

Now you're wondering what you could possibly use these two options for (apart from ringing the terminal bell I mean), and you're right. In fact, their existence comes from my desire to support Git projects by index.

If you look at `patcher-built-in-themes`, you will find two themes for Git (along with their their whitespace-cleaning counterpart): `git` and `git-index`. The `git-index` one will only work on what's in the Git staging area. This is cool as long as ChangeLog files are written by hand see Section 4.4.2.2 [Manual ChangeLogs], page 25. However, in automatic mode, we need a way to add them to the index once the skeletons are filled in. This is done by another built-in theme that you must add explicitly to your project, called `git-index-automatic-change-logs`. This theme uses the two options described above to automatically add ChangeLog entries to the staging area.

4.4.2.2 Manual ChangeLogs

In manual mode, Patcher assumes that you create ChangeLog entries manually, as you write the code, so it won't create ChangeLog skeletons. It is important to understand that in this situation, ChangeLog entries **must** have been written **before** you call Patcher. Patcher won't let you write them in the process.

Even in `manual` mode, Patcher might still need to know the affected ChangeLog files (for the commit process) and your exact ChangeLog entries in each of these files (for insertion in the message). The ChangeLog files are automatically deduced from the patch. When that's required, however, you will be presented with each ChangeLog file in turn, and invited to precise the number of ChangeLog entries concerning this patch. These entries must of course appear at the top of the file.

4.4.2.3 No ChangeLogs

This mode is for projects that don't do ChangeLogs. Patcher won't try to create ChangeLog entries, and won't expect that you have written ChangeLog entries either.

Note that if you *do* have ChangeLog files in this mode, they will be regarded as ordinary source files. As a consequence, this is a case where it is not forbidden to list them explicitly as part of a subproject, although I don't see why you would want to do that.

4.4.3 ChangeLogs Navigation

Patcher provides commands for navigating across ChangeLog and mail buffers, something especially convenient when you need to fill them by hand after skeletons have been created.

Patcher sees a project's mail and ChangeLog buffers as a circular chain that can be walked forward and backward by typing *C-c C-p n* or *C-c C-p p* respectively (depending on the buffer you're in, this involves either `patcher-change-log-next`, `patcher-change-log-previous`, `patcher-mail-first-change-log` or `patcher-mail-last-change-log`).

From a ChangeLog buffer, you can also shortcut the cycle and switch back to the mail buffer directly by typing *C-c C-p m* (`patcher-change-log-mail`), or switch to the first / last ChangeLog buffer respectively by typing *C-c C-p P* (`patcher-change-log-first`) / *C-c C-p N* (`patcher-change-log-last`).

4.4.4 ChangeLogs Appearance

The appearance of ChangeLog entries in the message is controlled by the `:change-logs-appearance` project option. Its value must be a symbol from the following:

`verbatim` This is the default. ChangeLog entries appear just as text in the message, above the patch. Most people prefer this kind of appearance since it is the most readable.

`pack` ChangeLog entries appear as a patch (they are diff'ed against the archive). This patch is however distinct from the source patch, and appears above it.

`patch` ChangeLog entries appear as a patch (they are diff'ed against the archive), and this patch is integrated into the source patch. In other words, the message looks like a global patch integrating both the sources and the ChangeLogs.

`nil` The ChangeLog entries don't appear in the message at all.

When the ChangeLogs appearance is either `pack` or `patch`, the diff command used to generate the patch is controlled by the `:change-logs-diff-command` project option. The value can be `nil`, meaning that the same diff command is to be used as for the sources (see Section 4.3.1 [Diff Command], page 21), or it can be a string specifying an alternate command.

When diffing ChangeLog files, it is strongly recommended that you remove contexts from the diff, because otherwise, ChangeLog patches often fail to apply correctly.

The `:change-logs-diff-command` project option supports the same substitution constructs as the `:diff-command` one (see Section 4.3.1 [Diff Command], page 21). For example, here is the ChangeLogs diff command used in the `git` built-in theme (see Section 4.1.1.1 [Themes], page 11): '`git diff -U0 --no-prefix HEAD%?f{ -- }%f`'.

When ChangeLog entries are written in advance (see Section 4.4.2.2 [Manual ChangeLogs], page 25), Patcher can (and does) insert them into the mail buffer automatically. However, Patcher cannot tell when you're done filling in skeletons (see Section 4.4.2.1 [Automatic ChangeLogs], page 24), so in such a case you need to insert the ChangeLog entries explicitly. This is done by calling the function `patcher-mail-insert-change-logs`. It is bound to *C-c C-p l* in the mail buffer.

With an additional prefix argument, or when your project is set to not normally include ChangeLogs in mail buffers, you will also be prompted for an alternate appearance (for this time only).

In fact, this function can also be used in all situations to **reinsert** the ChangeLog entries into the mail buffer, whatever their appearance. This comes in handy if you decide to further modify them after the initial insertion (it's never too late to fix a typo).

One last note: the same binding is available in ChangeLog buffers as well. The effect is to call `patcher-change-log-insert-change-logs`, which in essence switches to the mail buffer and performs insertion in a row. This saves you one *C-c C-p m* keystroke.

4.4.5 ChangeLogs Prologue

ChangeLog prologues are small pieces of informative text that Patcher adds above each ChangeLog insertion in the mail buffer.

When the ChangeLogs appearance is `verbatim`, Patcher inserts one prologue per ChangeLog file. The prologue's contents is controlled by the `:change-logs-prologue` project option (a string). A '`%f`' appearing in this string will be replaced with the ChangeLog filename. The default value for `patcher-default-change-logs-prologue` is `"%f addition:"`.

When the ChangeLogs appearance is `pack`, Patcher inserts only one prologue for the whole ChangeLogs patch. When `patch`, there is a single prologue for both the ChangeLogs and the sources. For customizing the prologue in both of these cases, see Section 4.3.4 [Diff Prologue], page 23.

4.4.6 ChangeLogs Status

Let's face it. ChangeLogs are in fact an obsolete concept that dates back to the old days when we used to work without revision control systems. Now the story is different: we have commit log messages, `git blame` and what not, to the point that ChangeLog files are big and not really useful anymore.

On the other hand, the ChangeLog file *format* is still convenient to describe modifications, for instance in the commit log message. So how nice would it be to continue manipulating ChangeLog entries, as usual, but just not store them into files?

Patcher can do that. It has a project option named `:change-logs-status` which can have two values (symbols). A value of `persistent` (the default) is in fact what we have assumed so far: there are ChangeLog files and they are part of the project. This is the traditional approach.

A value of `ephemeral` on the other hand means that your ChangeLog entries exist only temporarily, to be used in the commit log message and/or inserted verbatim in the mail. Patcher does this by creating a temporary ChangeLog file (named after the `:change-log-file-name` project option) in the project's base directory, and getting rid of it after the mail is sent. As a result, everything works just as if the ChangeLog file was real: ChangeLog entries can be generated automatically or written manually *etc.*

The only restriction is that you cannot diff the ephemeral ChangeLog entries because they are not really part of the project, so their appearance can only be `verbatim`. Also, when you use an ephemeral ChangeLog, beware to use a file name that doesn't conflict with existing files (old ChangeLog files may for example be renamed to '`ChangeLog.dead`').

Because there's only one, virtual, ephemeral ChangeLog file located at the project's base directory, the default value for the ChangeLogs prologue doesn't work very well in the ephemeral case. It doesn't make sense to refer to the file itself, since it's only temporary. A simpler prologue like "ChangeLog entries:" would suffice. Patcher provides a built-in theme called '`ephemeral-change-logs`' that you can use to both set the ChangeLog status to '`ephemeral`' and modify the prologue at the same time.

One final note: if you use the `git-index` built-in theme with ephemeral ChangeLogs, don't use it in conjunction with `git-index-automatic-change-logs`, even if the ChangeLogs entries are generated automatically by Patcher. Otherwise, they would be added to the staging area, which is definitely not what you want.

4.5 Project Check In

If you have the privilege to commit your changes yourself, you might do so directly from the mail buffer, as the last operation before actually sending the message. This is done by calling the function `patcher-mail-commit` which is bound to *`C-c C-p c`* in the mail buffer.

Committing directly from Patcher has the advantage that both the commit log message and command-line are constructed automatically. Of course, you still have an in-depth control on the commit process.

4.5.1 Commit Command

The command used to to commit a patch is specified by the `:commit-command` project option (a string). You can also temporarily change the command in question by calling `patcher-mail-commit` with a prefix argument. As usual, note that this prefix argument is not meant to modify the affected files on the command-line. It's meant only to punctually modify the commit command itself. The affected files are computed automatically by Patcher.

The commit command supports the same dynamic expansion constructs as the diff command (see Section 4.3.1 [Diff Command], page 21), and also adds two of its own.

`%s` A `%s` occurring in the commit command string will be replaced with the name of a file containing the commit log message (see Section 4.5.2 [Log Message Handling], page 28). This file is a temporary file handled by Patcher.

`%S` A `%S` occurring in the commit command string will be replaced with the commit log message itself (see Section 4.5.2 [Log Message Handling], page 28). Since the intent here is to use the message as a command-line argument, it will be automatically quoted against shell expansion.

Please note that exactly as for the diff command, a `%f` is required in your commit commands, unless you know for sure that you will never ever work on a subproject. But you never know that. Besides, you should always also provide either a `%s` or a `%S`, unless your archival software does not support log messages. I'm not actually sure such a beast exists.

As an example, here is the commit command for the Git built-in theme (see Section 4.1.1.1 [Themes], page 11): '`git commit %!f{-a }-F %s%?f{ -- }%f`'

4.5.2 Log Message Handling

Most project management tools understand the concept of a *log message*: a short yet informative message that accompany the commit operation, which is also stored in the repository.

Before a commit operation, Patcher always builds an initial log message, based on certain elements under your control. What happens next is controlled the `:edit-log-message` project option: if `t` (the default), you will be able to manually edit the log message. If `nil`, Patcher will proceed directly to the next step (see Section 4.5.3 [Commit Operation], page 30).

Please note that Patcher stores log messages in temporary files that may be used later by the commit command.

4.5.2.1 Log Message Elements

Patcher has the ability to initialize the log message with different elements. These elements are specified with the `:log-message-items` project option. Its value is either `nil`, meaning that you don't want any initialization, or a list of symbols specifying the elements you desire. The available items are:

`subject` The subject of the message. The subject's prefix is automatically removed.

`compressed-change-logs`
The "compressed" ChangeLog entries. Only the most important part of the ChangeLogs is preserved, so the entries appear in a more compact fashion.

`change-logs`
: The raw ChangeLog entries.

By default, only the message's subject is used. When using more than one item, they appear in the order specified above. If anything appears before the raw ChangeLog entries, a separator string is used. This string is specified by the `:change-logs-separator` project option. By default the string looks like "— ChangeLog entries follow: —".

Note that it only makes sense to set `:log-message-items` to `nil` if you also ask Patcher to let you edit the message (see Section 4.5.2.2 [Log Message Editing], page 29). Otherwise, your commit would end up with empty log messages.

4.5.2.2 Log Message Editing

If so required, Patcher lets you manually edit the log message after having initialized it. Log message editing happens in a special buffer called `*<project name> Patcher Project Log Message*`.

This buffer is governed by a major mode called `patcher-logmsg-mode`. This mode offers a hook, `patcher-logmsg-mode-hook`, which you can use to plug additional behavior like turning on font lock. If you do so, you might also want to have a look at `patcher-logmsg-font-lock-keywords`, `patcher-comment-face` and `patcher-reference-face` which are the built-in elements for log message fontification.

When the log message buffer is initialized, it starts with an informative comment header. The actual log message starts at the first non blank line after this header.

While editing this buffer, commands to insert the items described in Section 4.5.2.1 [Log Message Elements], page 28 are at your disposal. These commands perform insertion at point:

`patcher-logmsg-insert-subject`
: Bound to *C-c C-p s*. Insert the message's subject (sans the prefix).

`patcher-logmsg-insert-change-logs`
: Bound to *C-c C-p l*. Insert the ChangeLog entries. Use a prefix argument if you also want the ChangeLogs separator string to be inserted.

`patcher-logmsg-insert-compressed-change-logs`
: Bound to *C-c C-p L*. Insert the compressed ChangeLog entries. Use a prefix argument if you also want the ChangeLogs separator string to be inserted.

In addition to these commands, you can also completely reinitialize the log message by calling the function `patcher-logmsg-init-message`, bound to *C-c C-p i*. Caution: this command first erases the buffer.

Once you're happy with your log message, you proceed to the commit operation by calling the function `patcher-logmsg-commit`, bound to either *C-c C-p c* as in mail buffers, or directly to *C-c C-c*.

Finally, the log message offers two more commands in case you change your mind about the commit:

`patcher-logmsg-cancel`
: Bound to *C-c C-z*. Use this when you decide to cancel the commit operation (but not the whole project). Patcher will simply bring you back to where you came from; typically the mail buffer.

`patcher-logmsg-kill`
: Bound to *C-c C-k*. Use this to completely kill the project. Remember that you can also do that from mail or ChangeLog buffers.

4.5.3 Commit Operation

The commit operation occurs after typing *C-c C-p c* from the mail buffer if you have not required log message editing, or after typing *C-c C-p c* or *C-c C-c* from the log message buffer otherwise.

At that point, Patcher has constructed a proper commit command. What happens next depends on the value of the `:edit-commit-command` project option: if `nil`, Patcher performs the commit operation directly. Otherwise (the default), you have the ability to edit or at least confirm the commit command.

Commit command editing happens in a special buffer called `*<project name> Patcher Project Commit Command*`.

This buffer is governed by a major mode called `patcher-cmtcmd-mode`. This mode offers a hook, `patcher-cmtcmd-mode-hook`, which you can use to plug additional behavior like turning on font lock. If you do so, you might also want to have a look at `patcher-cmtcmd-font-lock-keywords`, `patcher-comment-face` and `patcher-reference-face` which are the built-in elements for commit command fontification.

This buffer starts with an informative comment header. The actual commit command consists in all non blank and non comment lines concatenated together.

Once you're happy with your commit command, you finally perform the operation by calling the function `patcher-cmtcmd-commit`, bound to both *C-c C-p c* as in mail buffers, and to *C-c C-c*.

The commit command buffer offers two more commands in case you change your mind about the commit:

`patcher-cmtcmd-cancel`
: Bound to *C-c C-z*. Just as in log message buffers, use this when you decide to cancel the commit operation (but not the whole project). Patcher will simply bring you back to where you came from; typically the mail buffer or the log message buffer.

`patcher-cmtcmd-kill`
: Bound to *C-c C-k*. Just as in log message buffers, use this to completely kill the project. Remember that you can also do that from mail or ChangeLog buffers as well.

After the commit operation, Patcher changes some parts of the mail buffer in the following manner:

- The subject prefix is changed to that specified by the `:subject-committed-prefix` project option (a string), unless it is `nil`. By default, "[COMMIT]" is used.
- A commit notice is added at the very beginning of the message's body. This notice is specified by the `:committed-notice` project option. It can be `nil` or a string. By default, it reads "NOTE: this patch has been committed.".

4.6 Mail Sending

Sending the message will most probably be done by typing '`C-c C-c`' in the mail buffer. This is also the case when you're using the fake mail method, by the way.

4.6.1 Before Sending

There are circumstances in which Patcher will perform some checkings on your message when you send it, just before it is actually sent:

- ChangeLogs insertion

  In case of manual ChangeLog insertion (see Section 4.4.4 [ChangeLogs Appearance], page 25), Patcher can check that you have indeed inserted the ChangeLog entries before sending the message. This behavior is controlled by the `:check-change-logs-insertion` project option. A value of `nil` means never check (the message will be sent as-is). A value of t means check, and abort the sending if the ChangeLog entries are missing. A value of `ask` (the default) means ask for your opinion on this terrible matter.

- Commit Operation

  Patcher has a `:commit-privilege` project option; a Boolean specifying whether you're likely to commit your changes by yourself.

  In case of commit privilege, Patcher can check that you have indeed committed your changes before sending the message. This behavior is controlled by the `:check-commit` user option. A value of `nil` means never check (the message will be sent as-is). A value of t means check, and abort the sending if the commit operation has not been performed. A value of `ask` (the default) means ask for your opinion on this rather unfortunate situation.

Two notes on these checkings:

- For uninteresting technical reasons, Patcher does not currently (and will probably never) offer you an automatic ChangeLogs insertion or commit operation, at mail sending time, but just abort the sending process in some circumstances. That's not a big deal though.
- For other uninteresting technical reasons, these checkings require a native knowledge of your mail user agent. Patcher does not currently support all mail user agents on earth (I'll add them on demand however). If that's the case, you will be warned and invited to send me a message. Also, you can send me one on April 21st: it's my birthday.

4.6.2 After Sending

After sending the message, Patcher also performs some cleanup operations, that you can customize. The cleanup is controlled by the following project options. Each one is a Boolean option which defaults to `t`.

`:kill-sources-after-sending`
: Whether to kill source files after sending the message. If `nil`, the source files will remain visited.

`:kill-change-logs-after-sending`
: Whether to kill ChangeLog files after sending the message. If `nil`, the ChangeLog files will remain visited.

When Patcher is allowed to kill a source or ChangeLog file, it will only actually kill it if it was responsible for loading it for this particular project in the first place. For instance, if the file was already visited before Patcher was launched, or if another Patcher project is also using the file, then it won't be killed regardless of the value of these options.

4.7 More On Commands

This section deals with information that apply to all commands used by Patcher (diff and commit operations).

4.7.1 Prefixing Commands

If you're working on a distant archive and you're behind a firewall, you might need to prefix all your commands with something like `runsocks`. Of course, this can be done manually in all your command settings, but Patcher offers you a simpler way to do it.

There is a project option named `:pre-command` which can be used for this kind of thing. It must be a string that will be prepended to all operations performed by Patcher.

4.7.2 Error Handling

From time to time, commands may fail for different reasons. Patcher tracks command failures and lets you know when that happens.

The first thing Patcher does is to check the external processes exit codes. A non-zero exit code will normally trigger a Patcher error. There is however one notable exception: `cvs diff` has this incredibly stupid idea of returning an exit code of 1 when the diff succeeds, and is non-empty. Because of this, Patcher provides an option named `:ignore-diff-status` that is set to `t` in the CVS theme. There should be no reason to use it in any other context.

Next, Patcher looks for specific strings in process output. The `:failed-command-regexp` project option lets you specify a regular expression to match with the output of an aborted command. In the CVS built-in theme for example (see Section 4.1.1.1 [Themes], page 11), the value is ‘`"^cvs \\[[^]]* aborted\\]"`’.

Appendix A XEmacs Development

XEmacs development occurs on a Mercurial repository. Patches are advertised on `xemacs-patches@xemacs.org`. We assume that you are a committer and have a clone of the main repository located at '`/usr/local/src/xemacs/21.5`'. Your '`hgrc`' file should look like this (replace my identity with yours):

```
[ui]
username = Didier Verna <didier@xemacs.org>

[paths]
default      = ssh://hg@bitbucket.org/xemacs/xemacs
```

The following project settings will do nicely for hacking XEmacs:

```
'("XEmacs 21.5" "/usr/local/src/xemacs/21.5"
   :to-address "xemacs-patches@xemacs.org"
   :change-logs-user-mail "didier@xemacs.org"
   :commit-privilege t
   :log-message-items (subject change-logs)
   :themes (mercurial))
```

If you want to also work on the packages, you may clone the big umbrella repository. Let's assume you do so in '`/usr/local/share/emacs-lisp/source/xemacs-packages`'. Your '`hgrc`' file should look like this:

```
[ui]
username = Didier Verna <didier@xemacs.org>

[paths]
default      = ssh://hg@bitbucket.org/xemacs/xemacs-packages
```

The following project settings will do nicely for hacking the packages (all settings are in fact inherited from the main XEmacs 21.5 project):

```
'("XEmacs Packages" "/usr/local/share/emacs-lisp/source/xemacs-packages"
  :inheritance ("XEmacs 21.5"))
```

However, note that you shouldn't work on this project directly. Every package is in fact stored under this project as a Mercurial submodule. Patcher detects every such submodule automatically and creates a corresponding project for you (submodule projects are named '`XEmacs Packages (<submodule>)`'. Because those really are independent projects, you should probably also update every '`hgrc`' file with your identity.

Appendix B Indexes

B.1 Concepts

B.2 Variables

B.3 Functions

B.4 Keystrokes