A self-referential HOWTO on release engineering[*]

Mark Galassi[†]

Space Science and Applications group
Los Alamos National Laboratory[‡]

August 16, 2018

Abstract

Release engineering is a fundamental part of the software development cycle: it is the point at which quality control is exercised and bug fixes are integrated. The way in which software is released also gives the end user her first experience of a software package, while in scientific computing release engineering can guarantee reproducibility. For these reasons and others, the release process is a good indicator of the maturity and organization of a development team.

Software teams often do not put in place a release process at the beginning. This is unfortunate because the team does not have early and continuous execution of test suites, and it does not exercise the software in the same conditions as the end users.

I describe an approach to release engineering based on the software tools developed and used by the GNU project, together with several specific proposals related to packaging and distribution. I do this in a step-by-step manner, demonstrating how *this very paper* is written and built using proper release engineering methods. Because many aspects of release engineering are not exercised in the building of the paper, the accompanying software repository also contains examples of software libraries.

[*]For use with software version 1.0.0plus
[†]mark@galassi.org
[‡]LA-UR-14-21151

Contents

Chapter 1

Motivation and plan

A few years ago in the course of a social conversation a fellow scientist and I dwelled upon software development. He told me that like most physicists he could write software for his own purposes, but that he could not imagine writing software that others could use.

I have always wanted others to be able to use my scientific software, which was a big part of the driver behind the GNU Scientific Library (GSL) (Galassi, Theiler, Gough, et al. 2009), to which I contributed. I have also never felt that "usability by others" was such a difficult goal.

In this article I present some procedures and examples which I hope will guide you into releasing your software in a careful and principled way. If we are successful this will improve the experience of your end users and make it straightforward for you to implement robust quality control.

1.1 A historical example: the GNU Scientific Library

The GSL is a large and complex piece of software, yet it is robustly engineered and easily installed from source code. It has been straightforwardly packaged for the major GNU/Linux packaging approaches (Debian and Red Hat) as well as being available on the proprietary Macintosh and Windows operating systems.

This "robust engineering" grew out of a few design principles:

1. a road map with clear goals
2. a design document with coding standards
3. a build and release system
4. a testing framework
5. comprehensive high-quality documentation

We derived almost all of our approach from the established practices of the free software movement (Stallman 1985). We also adopted the GNU Coding Standards (Stallman et al. 1992-2013) in their entirety.

A decade and a half later the GSL is still robust, relevant and widely used, confirming the large return on a small upfront investment.

1.2 Requirements

A release engineering approach should, at a minimum, ensure:

Reproducibiliy It should always be clear if the installation of your software is a proper release (and which version it is), a development snapshot, a beta test, ... In addition, it should be possible to pull together the exact set of external tools and internal software which went into any past release.

Quality control The release process should guarantee frequent execution of the software test suites.

Easy installation The software should install easily on target platforms (your end user's computers). This often means using the native packaging approach for that system.

Upgrade path You should be able to upgrade the software, or even remove it, and know that there are no files from the previous version littering your computers disk and inviting trouble.

Rapid problem resolution There should be a straightforward way to connect problem reports with the testing infrastructure and the version control system, so that developers and end users know what bugs are expected to be fixed in a new release.

Portability The software should be robust enough to run on different operating system distributions.

1.3 A gritty and *real* business

Before we embark upon this tutorial I must emphasize that release engineering is an area of computing that is seldom clean and tidy.

Occasionally you get lucky and things are tidy: for example, the GSL is purely mathematical software, where each routine is a black box that takes an input and gives a deterministic output. It uses very stable infrastructure components: the C programming language, floating point arithmetic and the standard math library follow standards that have barely changed in recent decades.

Even just using C++ as your programming language increases the complexity: the C++ standard is still evolving, and if you write code that uses the C++11 standard you will have to do some careful checking to see that your program can be built on a given computer.

Release engineering is even more difficult if, for example, your software uses a graphical widget toolkit: these pieces of infrastructure are seldom standardized, they are implemented very differently on different platforms, and they change rapidly with each new version.

When your code uses this fussy infrastructure you will find that you need to test it on many different platforms, and each platform will require that your code handle special cases according to which version of the graphical toolkit is installed on the target system.

This is even more true if your package involves a web user interface: the dust is barely beginning to settle in that arena, with several contenders among server-side platforms and a variety of libraries for the client side.

So do not expect that following these "established practices" will make everything smooth: you will still have to come up with dirty tricks to deal with this real-world lack of neatness.

1.4 And what are these established practices?

The shortest summary of how to start a "GNU style" project would be:

1. Use version control from the very beginning. Nowadays (since 2005) use a distributed version control system, such as Mercurial or Git.

2. Write down a design which allows a reader to close her eyes and visualize the project as if it were a movie playing in her head.

3. Create a skeleton project with just a few files of source code.

4. While the project is tiny create a build system using the GNU autotools or something equivalent. (Vaughan et al. 2000) (I discuss build systems more in Section 9.2.)

5. Start writing documentation immediately and continue to (re)write it as you develop the software. Your documentation should include (at least) a pedagogical tutorial and a full reference.

6. When you *release to the public* follow a careful procedure leading up to the publication of your source code. The GNU coding standards specify what the "tarball" with your source code should look like. The GNU autotools make it easy to generate a tarball by typing "`make dist`" or "`make distcheck`".

7. Work with the "package maintainers" – these are people who prepare binary packages for the various distributions (it could even be you or someone on your team).

1.5 A preview of our tour

My plan is to give you a *self-referential HOWTO*: this article guides you through the release engineering process for a project that includes little other than ... this article!

We will start by creating a skeleton project with a mostly empty program — just enough to demonstrate a build system that compiles it automatically.

Then we will create such a build system using the GNU autotools; this is where we start introducing version numbers.

At this point we will introduce a simple procedure for *making a release*.

Then we will add some documentation source files, such as this tutorial and a man page, and we will have the build system generate the documentation product (pdf files, man page output, ...)

Then our first major step: we will adapt our build system to make binary packages, starting with *RPM* packages for Red Hat based GNU/Linux systems.

Then we will add the ability to make *deb* packages for Debian and Ubuntu GNU/Linux systems.

Once we have a smooth approach to generating packages we will make our software project more complete by introducing a *library*. This will then prompt us to introduce a *test suite* for the library.

We will then add a bug and introduce a test program that fails because of the bug. This is an important formal step because it guarantees that we will know if the same bug reappears.

We then delve into some special topics: a discussion of *versioning*, how that communicates information to recipients of your software, and how you can manage versions. Then a discussion of some useful tools for project management, and then more on packaging.

We will conclude with some miscellaneous notes and tricks, and finally an appendix with checklists for software releases.

Chapter 2

Getting started with our `toy-releng` project

2.1 Cloning the repository

First you should clone the repository which I am using to build this paper. The version control system is Mercurial (Mackall 2006), and the project is hosted on the public server bitbucket (Atalassian 2015). You can clone the repository with these commands:

```
hg clone https://markgalassi@bitbucket.org/markgalassi/toy-releng
cd toy-releng
```

You should also make your own work area to try out these examples. In a separate terminal you should type:

```
mkdir my-releng
cd my-releng
```

and put this area under version control with:

```
hg init .
```

(note that there is a `.` (period) at the end of that line.)

If you have done both (cloning the `toy-releng` repository with this paper and making your own `my-releng` repository) then you could work with them side-by-side, possibly with two separate terminal windows.

If you are more comfortable with git then go ahead and use that; just make sure it's a modern distributed version control system (DVCS). At the time of writing the robust, full-featured and widely used DVCSs are Mercurial and Git.

Now that you have cloned it you will be able to find the files I refer to below; you can copy them into your own repository step by step.

2.2 Creating a first program and build system

Create files called `toy-releng-sample.c` and `toy-releng-sample.h` with these lines in them:

```
#include <stdio.h>
#include "toy-releng-sample.h"

int main(int argc, char *argv[])
{
  return 0;
}
```

Listing 2.1: toy-releng-sample.c

```
#pragma once

#define TOY_RELENG_SAMPLE_CONSTANT 3.7
```

Listing 2.2: toy-releng-sample.h

Add these files to version control with

```
hg add toy-releng-sample.c toy-releng-sample.h
hg commit -m "Added␣first␣draft␣of␣toy-releng-sample␣program"
```

For good measure we will also add a (brief) `README` file:[1]

```
toy-releng  is  a  project  to  demonstrate  how  to  do
comprehensive and  highly  principled  release  engineering
on  a  "toy"  project
```

Listing 2.3: README

Now we will add the files needed by automake and autoconf so that we can build this program. Since our program is be quite trivial and the GNU autotools system has very advanced capabilities, this will be a bit like using the proverbial sledgehammer to kill a fly. But it will scale well to more complex projects, and it also gives us RPM and Debian packaging, as well as a testing framework, for free.

Create the files `configure.ac` and `Makefile.am`:

```
AC_INIT([toy-releng], [0.0.0plus], [mark@galassi.org])
AM_INIT_AUTOMAKE([-Wall foreign])
AC_CONFIG_SRCDIR([config.h.in])
AC_CONFIG_HEADERS([config.h])
# Checks for programs.
AC_PROG_CC
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

Listing 2.4: configure.ac

```
bin_PROGRAMS = toy-releng-sample
toy_releng_sample_SOURCES = toy-releng-sample.c
include_HEADERS = toy-releng-sample.h
```

Listing 2.5: Makefile.am

I don't intend to give a full tutorial on autotools here (Vaughan et al. 2000), but I will show you the most common formula for building your program. Starting from your top level source directory type the following commands:

[1] Of course every time you add a file you need to make sure that you also add it to the version control system; in this case with "`hg add README`" and you will eventually also need to commit your work with "`hg commit`"

```
autoreconf -i
mkdir _build
cd _build
../configure
make
```

Most of these steps were only needed the first time. In future invocations you can usually just type

```
make
```

and of course you then run the program with

```
./toy-releng-sample
```

and it will do nothing, since the program has no instructions in it yet.

Although the program did nothing, please note an important thing in how we built it: *we built the program in a separate directory from where the sources are.* We ran "configure", "make" and "./toy-releng-sample" in the `_build` subdirectory. This is standard practice in software development: using *separate build directories* is a good habit because it flushes out some unhealthy build system assumptions and it does not litter your source directories with compiler output and other computer-generated files.

2.3 First stab at tagging and releasing: putting out release 0.1.0

Now that you have a build system you could go ahead and release your program. The checklist is for this first-ever release is:

1. Make sure you are conceptually happy with what you have done.

2. Create a `ChangeLog` file in your top level source directory. This is a file that lists all changes you make to your source code, describing briefly what has been done in each file. This is for developers to be able to get a feeling for what has changed in the source code.[2] An example is in Listing 2.6, and you can also see much bigger examples entries in the repository for this paper. A full description of the ChangeLog specification is in the GNU Coding Standards (Stallman et al. 1992-2013). Add it to version control with "`hg add ChangeLog`" in the source directory.

3. Create a `NEWS` file in your top level source directory with end-user-visible release notes (an example is shown in Listing 2.7). Add it to version control with "`hg add NEWS`" in the source directory.

4. Make sure that everything builds and installs with "`make distcheck`" (in the `_build` directory.)

5. Commit to version control.

6. Edit `configure.ac` and change the version from `0.0.0plus` to `0.1.0`.

7. Commit to version control

8. Tag the version control repository with

   ```
   hg tag release-0.1.0
   ```

   This tagging step is extremely important because it guarantees that you can always reproduce what went into version 0.1.0 from your version control repository. This *reproducibility* will become very important as your project grows, but even now you should make a habit of it.

9. You can now make the tarball with

   ```
   make distcheck
   ```

```
2015-06-10 Mark Galassi <mark@galassi.org>

    * NEWS: wrote news entries for release 0.1.0

    * configure.ac: created for toy-releng project

    * Makefile.am: created for toy-releng project
```

Listing 2.6: ChangeLog

```
* Noteworthy changes in release 0.1.0 (2014-02-14)
** Features
*** Simple program and build system
Simple program which builds and executes with an \
     autotools
build system.
```

Listing 2.7: NEWS

You now have a source code tarball `toy-releng-0.1.0.tar.gz` which you can distribute to end users. You are now ready to return to development, but I will add one last item to the checklist.

To motivate this last item on the checklist let me remind you that in Section 1.2 I stressed the importance of *reproducibility*. Now that we have built a tarball called `toy-releng-0.1.0.tar.gz`, imagine what would happen if we made changes to the code and then typed "`make dist`" (or "`make distcheck`"): you would have created distributions of your code that are both labeled as being version 0.1.0, but they are different!

The need be certain about what is in our *formally released* versions motivates the last item in our checklist:

10. As you return to development you should edit `configure.ac` and change the version from `0.1.0` to `0.1.0plus`. The "plus" is our convention to signal that these are "code snapshots" and should not be confused with our "formal released" `0.1.0` version. You can commit with

    ```
    hg commit -m "return␣to␣development␣snapshots"
    ```

[2] A good practice is to use entries in the `ChangeLog` file as a commit log in version control. The opposite approach also works: to convert version control logs into a ChangeLog file.

2.4 What do your "end users" do with a tarball?

Your end users can build the software from a tarball with a rather minimal set of developer tools. They do not need all the fancy release engineering apparatus that you have (autotools, for example). They only need a basic version of `make`, a C compiler, and a shell.

Here is how the end user would build and install the software from your tarball:

```
tar zxvf toy-releng-0.1.0.tar.gz
cd toy-releng-0.1.0
./configure
make install
```

at which point the toy-releng-sample binary is installed in `/usr/local/bin` and can be run from the command line.

This has been the standard way of distributing software in the GNU project for a very long time (Stallman et al. 1992-2013).

(Now that you have tried that out, clean it all up so it does not interfere with our future builds: "`make uninstall`" and "`cd ..; rm -r toy-releng-0.1.0`")

2.5 Adding a few more needed files

From a pedagogical point of view it was good to quickly put out a release and start getting comfortable with the process. But now we must pause and discuss a couple of crucial steps in this process: installation docs and copyright/licensing.

Standard GNU tarballs install in a very uniform manner as you saw in Section 2.4, and there is a boiler-plate file called `INSTALL` that is distributed by automake. You can add it to your repository with the following commands (in your top level source directory):

```
cp /usr/share/automake-1.14/INSTALL .
hg add INSTALL
```

and it will be automatically included in the tarball.

You should also immediately start clarifying the legal status of your source code, since your files will soon start getting long enough (more than about 10 lines) that they cannot be copied unless you grant explicit permission.

We will do this by creating two files called COPYRIGHT and LICENSE. The file COPYRIGHT is a brief document which tells the recipient that toy-releng is free software and that the licensing terms (the GNU General Public License) are spelled out in the file called LICENSE.

We then add these two files to the `EXTRA_DIST` section of `Makefile.am`[3]

2.6 Making the build system interact with the program

Many programs need to know what version they are, if nothing else to be able to inform the user. To add some toy complexity to our program we will make it interact with the build system in the following manner: the build system makes a file called config.h which defines C preprocessor macros `PACKAGE` and `VERSION` to match the arguments to `AC_INIT()` in `configure.ac`

Modify `toy-releng-sample.c` to `#include "config.h"` and to have a line that references `PACKAGE` and `VERSION` in its `main`:

```
#include <stdio.h>
#include "toy-releng-sample.h"
#include "config.h"

int main(int argc, char *argv[])
{
  printf("welcome to the package %s, version %s\n",
         PACKAGE, VERSION);
  return 0;
}
```

Listing 2.8: toy-releng-sample.c

If you compile and run it you will see that information from the build system (package name and version) appears in the program's output:

```
$ make
[ lots of output from build ]
$ ./toy-releng-sample
welcome to the package toy-releng, version 0.1.0plus
```

2.7 Adding documentation

Automake provides automatic handling of *UNIX man pages* and TeXinfo documentation: they will be built and installed in standard destination directories on your system.

There are many types of documentation beyond man pages and TeXinfo docs, but in this toy project we will only have two toy documents for inclusion: a man page for `toy-releng` and a journal article written in LaTeX (this article!) We will create simple documents, and then show how the build system builds them.

First create a man page called `toy-releng-sample.1` (you can lift it from the toy-releng repository.) Put it in the top level directory and put this line in `Makefile.am`:

```
dist_man_MANS = toy-releng-sample.1
```

Listing 2.9: Makefile.am

and when you type `make install` the man page will be generated and installed in `/usr/local/share/man/man1/toy-releng-sample.1` and can be viewed with

```
export MANPATH=/usr/local/share/man
man toy-releng-sample
```

[3]Note that you can look at my version control repository for toy-releng for examples of the `COPYRIGHT` and `LICENSING` files, and to see what EXTRA_DIST looks like inside `Makefile.am`.

Then let us create a LaTeX document with a "scholarly" article. For simplicity you can create a trivial almost-empty LaTeX document, or you can lift this very paper from my repository.

Either way, you put the two files `toy-releng-howto.tex` and `releng.bib` in a subdirectory (for example `howto-paper`). Building from the `Makefile.am` is a bit more elaborate. Add these lines to `Makefile.am`:

```
pdf-local: toy-releng-howto.pdf

toy-releng-howto.pdf: howto-paper/toy-releng-howto.tex \
     howto-paper/releng.bib
   BIBINPUTS=$(srcdir)/howto-paper latexmk -f -pdf $<

install-pdf-local: pdf-local
   mkdir -p $(DESTDIR)$(pkgdatadir)/
   cp toy-releng-howto.pdf $(DESTDIR)$(pkgdatadir)/
```

Listing 2.10: Makefile.am

and add `howto-paper/toy-releng-howto.tex` and `howto-paper/releng.bib` to the `EXTRA_DIST` variable in Makefile.am. You can build the PDF document with "make pdf" and install it with "make install-pdf".[4]

2.8 Putting out release 0.2.0

Now that we have introduced these new features and documentation we can put out a new version. We will update the `NEWS` file (from the top) with the entry

```
 * Noteworthy changes in release 0.2.0 (2014-02-19)
 ** Features
 *** Added interaction of build system with program:
 toy-releng now uses PACKAGE and VERSION out of configure.ac
 *** Wrote man page
 You can now type "man␣toy-releng" to see the man page.
 *** Started writing tutorial HOWTO article
 There is now a good amount of work in \
      howto-paper/toy-releng-howto.tex
```

Listing 2.11: NEWS

You should run "`make distcheck`" to see that a good tarball is being built.

Now that you have documented your release you can follow the simple checklist in Section 2.3 by committing, then editing `configure.ac` to set the version from `0.1.0plus` to `0.2.0`, committing that new version, running "`hg tag release-0.2.0`", then building the tarball with "`make distcheck`".

Before moving on remember to follow the step given in Item 10 of the checklist in Section 2.3 and set the version to `0.2.0plus`

[4] As usual, check the accompanying software repository for extra detail on `Makefile.am`: there are extra details that I will not always put into the text of this paper, such as the `CLEANFILES` variable and the `clean-local:` target.

Chapter 3

Completing `toy-releng`

Version `0.2.0` got us to the point of releasing a source code tarball for our program `toy-releng-sample`. This tarball could build a C program, a manual, and a longer LaTeX document.

But since the 1990s the preferred way to deliver software has been through *binary packages*. Binary packaging approaches differ from one operating system to another and even from one distribution to another. We will discuss two GNU/Linux packaging approaches:

- Redhat Packager Manager (RPM), used by the Red Hat and Fedora distributions of the GNU/Linux operating system (among others).
- Debian packages (deb), used by the Debian and Ubuntu distributions of the GNU/Linux operating system (among others).

Packaging is closely tied to the build system: the software which helps create packages makes use of some Makefile targets which are specified by the GNU coding standards and are automatically present when you use autotools.

3.1 Making an RPM package

The Fedora project offers an up-to-date introduction to RPM packaging (Fedora team 2013b), as well as full guidelines for packaging (Fedora team 2013a).

Since these Fedora project documents are well-written I will not explain the concepts, nor give a full reference, but rather we will discuss how the RPM packaging procedure interacts with our autotools-based build system.

3.1.1 Preparing a spec file

To make an RPM package we prepare a file called `toy-releng.spec`. This file gives instructions on how to get, build, install and bundle the binary program and its supporting files.

The smallest skeleton of a spec file for toy-releng (named `toy-releng.spec`) would look like this:

```
Name:           toy-releng
Version:        0.2.0plus
Release:        1%
Summary:        toy project to demonstrate release \
        engineering

License:        GPL
URL:            \
        https://bitbucket.org/markgalassi/toy-releng/overview
Source0:        %{name}-%{version}.tar.gz
BuildRoot: %{_tmppath}/%{name}-%{version}-%{release}-root

BuildRequires: texlive

%description
toy-releng is a project to demonstrate how to do \
        comprehensive and
highly principled release engineering on a "toy" project

%prep
%setup -q

%build
%configure
make %{?_smp_mflags}
make pdf

%install
rm -rf $RPM_BUILD_ROOT
%make_install
make install-pdf DESTDIR=$RPM_BUILD_ROOT/

%files
%doc README NEWS ChangeLog toy-releng-howto.pdf
%{_datadir}/%{name}/*
%{_mandir}/man1/*
%{_includedir}/*.h
%{_bindir}/*

%changelog
* Tue Feb 18 2014 Mark Galassi
- first packaging; see distribution ChangeLog file for \
        ChangeLog info
```

Listing 3.1: toy-releng.spec

3.1.2 Building and installing the RPM

Once `toy-releng.spec` is ready you prepare to build the RPM by putting your tarball in a directory called `~/rpmbuild/SOURCES/`

```
mkdir ~/rpmbuild/SOURCES
make dist
cp toy-releng-0.2.0plus.tar.gz ~/rpmbuild/SOURCES
rpmbuild -ba ../toy-releng.spec
```

you will see how the rpmbuild program goes through all the sections of the spec file and ends up placing your RPM in `~/rpmbuild/RPMS/x86_64/toy-releng-0.2.0plus-1.x86_64.rpm`. You can now install that RPM, possibly with the command:

```
sudo rpm -Uvh \
  ~/rpmbuild/RPMS/x86_64/toy-releng-0.2.0plus-1.x86_64.rpm
```

and see what files have been installed:

```
$ rpm -ql toy-releng
/usr/bin/toy-releng
/usr/include/toy-releng.h
/usr/share/doc/toy-releng
/usr/share/doc/toy-releng/ChangeLog
/usr/share/doc/toy-releng/NEWS
/usr/share/doc/toy-releng/README
/usr/share/doc/toy-releng/toy-releng-howto.pdf
/usr/share/man/man1/toy-releng.1.gz
/usr/share/toy-releng/toy-releng-howto.pdf
```

You can now exercise the installation by typing commands such as:

```
toy-releng
man toy-releng
evince /usr/share/dodc/toy-releng/toy-releng-howto.pdf &
```

3.1.3 Generating the spec file from the build system

In listing 3.1 I demonstrated a very simple spec file for building toy-releng RPMs. The first two lines of the spec file look like:

```
Name:         toy-releng
Version:      0.2.0plus
# ...
```

Listing 3.2: toy-releng.spec - first two lines

Since the package name and version are already given in configure.ac, we appear to be violating one of our fundamental software engineering principles: *do not duplicate information; derive it instead*. If you violate this principle it is very easy to have "user error" in which you update `configure.ac` but forget to update `toy-releng.spec` – this kind of thing happens all the time.

Fortunately the autotools system allows us to have the `configure` script generate `toy-releng.spec` from a skeleton file called `toy-releng.spec.in` with the first two lines modified. Let us rename `toy-releng.spec` and add it to our version control:

```
mv toy-releng.spec toy-releng.spec.in
hg add toy-releng.spec.in
```

and then modify the top two lines:

```
Name:         @NAME@
Version:      @VERSION@
# ...
```

Listing 3.3: toy-releng.spec.in - first two lines with substitution variables

where `@NAME@` and `@VERSION@` will be substituted by the `configure` script. We also need to modify the end of `configure.ac`:

```
# ...
AC_CONFIG_FILES([ Makefile toy-releng.spec ])
AC_OUTPUT
```

Listing 3.4: configure.ac - last lines which generate spec file

With these modifications to `configure.ac` and `toy-releng.spec.in` you can use the new automatically built spec file to build your RPM. Type in the following sequence:

```
cd _build
make distcheck
cp toy-releng-0.2.0plus.tar.gz ~/rmpbuild/SOURCES/
rpmbuild -ba toy-releng.spec
```

which will give the same result we got in Section 3.1.2 but our configuration will be more robust as the software evelves.

3.1.4 Going directly from tarball to RPM

You might notice that there were some repetitive (and possibly error-prone) steps involved in building the RPM: you make a tarball, then you copy it to `~/rpmbuild/SOURCES/`, then you run `rpmbuild`.

There is a way of building an RPM directly from a tarball, if (a) the tarball conforms to the GNU coding standards (which automake gives us) and (b) the top level directory contains a spec file.

Let us modify `Makefile.am` slightly to make sure we distribute `toy-releng.spec.in` and `toy-releng.spec`. The `EXTRA_DIST` variable now looks like:

```
EXTRA_DIST = COPYRIGHT LICENSE toy-releng.spec.in \
   toy-releng.spec howto-paper/toy-releng-howto.tex \
   howto-paper/releng.bib
```

Listing 3.5: Makefile.am - EXTRA_DIST variable

You can now build your distribution tarball and your RPM packages with two commands:

```
make distcheck
rpmbuild -ta toy-releng-0.2.0plus.tar.gz
```

Note the difference in invoking `rpmbuild`: when it works from a spec file it takes the options `-ba`, while if it's working from a tarball it takes the options `-ta`

There are different points of view on whether the spec file should be distributed separately from the application or if it should be included in the tarball[1].

One point of view is the "official" one from the engineers whose business it is to take a huge number of programs and prepare them for a distribution, as the Fedora and Red Hat engineers do. They have special conventions they follow for spec files so they end up doing things quite differently from how a developer would, and the developer-provided spec file might just be misleading.

I instead prefer the "build RPM from tarball" approach. Part of this is the simplicity of the command "`rpmbuild -\ ta file.tar.gz`" and of not having to explicitly move your sources around. More importantly, though: when you provide a spec file with your program you achieve two goals:

- You provide an easy way of making RPMs if your program will *not* be part of a major distribution like Fedora or Red Hat.

[1]see Section 8.1.

- If your package gets picked up for a distribution you give a hint to the people who will prepare those spec files: they can start from yours.

3.2 Putting out release 0.3.0

Adding the ability to make RPMs is a milestone for our project and we will now put out a new release. Our release procedure from Section 2.3 should still apply, but we will add one more instruction to build the RPM.

We will get a bit fancier in Section 3.4 where we will start using release branches, but for now let us enjoy our very simple release checklist.

1. Add to the `NEWS` file with end-user-visible release notes. For example:

```
* Noteworthy changes in release 0.3.0 (2014-02-20)
** Features
*** Added the ability to build RPMs
    We now offer a toy-releng.spec.in  file  which allows you to build
    RPMs with rpmbuild -ta toy-releng-0.3.0.tar.gz
*** Added significantly  to toy-releng-howto.tex
    Wrote a good part of the "Complete_toy-releng" section.
```

Listing 3.6: NEWS file for 0.3.0

2. Make sure that everything builds and installs with "`make distcheck`".
3. Commit to version control.
4. Edit `configure.ac` and change the version from `0.2.0plus` to `0.3.0`
5. Commit to version control.
6. Tag the version control repository with

   ```
   hg tag release-0.3.0
   ```

7. Prepare RPMs with:

   ```
   make distcheck
   rpmbuild -ta toy-releng-0.3.0.tar.gz
   ```

3.3 Debian packaging

It is also a good idea to distribute packages for Debian GNU/Linux systems. These should also work for Ubuntu and other Debian-based distributions.

I will not provide a full tutorial on Debian packaging, since the Debian Packaging Team has written an excellent one (Rodin and Aoki 1998-2013), but I will list the steps you need to carry out for our program.

3.3.1 Preparing for Debian packaging

Instead of a single spec file, with Debian you need to prepare several files – at least 4 for a straightforward package: `Debian/control`, `Debian/copyright`, `Debian/changelog` and `Debian/rules` – as well as some boiler-plate files.

The way Debian prepares these files is rather elaborate, so we will work in a fresh copy of everything. My favorite way of doing this is to make a directory called /tmp/sandbox and work with a fresh version control clone in there. From the top level directory in `toy-releng` we can do the following:

```
mkdir /tmp/sandbox
hg clone . /tmp/sandbox/toy-releng
cd /tmp/sandbox/toy-releng
```

To let the Debian packaging "helper" scripts do their work we need a Makefile ready in the current directory, so we will run:

```
autoreconf -i
./configure
```

and now we have a `Makefile` and we are ready to prepare the skeleton Debian files. The first command will create a directory called `Debian` with a bunch of empty skeleton files. We don't need most of them so we will remove them and set up just the ones we want:

```
dh_make --native --createorig -p toy-releng_0.3.0 -e \
     "YOUR_EMAIL_ADDRESS"
```

(then type `'s'` to select a "single package" and hit enter to accept.) Now remove the excess files and add the essential ones to version control:

```
rm Debian/*.ex Debian/*.EX
rm Debian/README*
hg add Debian/control Debian/changelog \
   Debian/copyright Debian/rules Debian/compat \
   Debian/source/format
hg commit -m 'added boiler-plate Debian files with \
   dh_make --native --createorig -p toy-releng_0.3.0 \
   -e "YOUR_EMAIL_ADDRESS"'
hg push
```

and add those same files in the `Debian` directory to `Makefile.am`'s `EXTRA_DIST` variable. That should now look like:

```
EXTRA_DIST = COPYRIGHT LICENSE project-admin.org \
  toy-releng.spec.in toy-releng.spec \
  howto-paper/toy-releng-howto.tex \
  howto-paper/releng.bib Debian/changelog Debian/compat \
  Debian/control Debian/copyright Debian/rules \
  Debian/source/format
```

Listing 3.7: Makefile.am

You will now need to modify a couple of these. The file `format` is boiler-plate and does not need changes. The file `rules` will also not need much work since we use autotools. On the other hand we want to be clear about copyright and licensing, so we edit the file `Debian/copyright`. You can just fill in your name and the years, and since the file is already set up for the GNU General Public License (GPL) version 3, you can leave it as it is to use the GPL. Alternatively you may replace it with a different license.

The most important file is `Debian/control`. It has already been filled out for you, so you only have to make a couple of small modifications. Edit it and choose a `Section:` from the list at `https://packages.Debian.org/unstable/`. In our case the section is `Documentation`. You can leave most other lines in `Debian/control` untouched, but put a brief half-line description on the line `Description:`, and on the next line you can start a longer description (possibly from your `README` file) indented with a space.

3.3.2 Building and installing the Debian package

You are now ready to build your Debian package: let us commit the changes with:

```
hg commit -m 'modified the Debian packaging files for our project'
```

now make sure you push these changes back to your main repository clone, since here we are in a sandbox directory that we will soon remove

```
hg push
```

The instruction to build the package is:

```
debuild -us -uc
```

You now have some files file (up one directory). The actual package is `../toy-releng_0.3.0_amd64.deb`. You can install this with:

```
sudo dpkg -i ../toy-releng_0.3.0_amd64.deb
```

and you can exercise it by typing:

```
cd ~
toy-releng
man toy-releng
```

and then you can remove it so that we can continue development:

```
sudo dpkg -i ../toy-releng_0.3.0_amd64.deb
```

We are now done with preparation of our Debian package. It was good to work in a sandbox (i.e. in the directory `/tmp/sandbox` because you might have a few false starts as you prepare the files. But from now on we will only make incremental changes to the Debian packaging files. So we will push our changes back into our main working copy of the repository and delete the sandbox:

```
hg push
cd /path/to/original-working-area
hg -v update
rm -r /tmp/sandbox/toy-releng ## WARNING: remove with care!
```

Note that in future releases you will simply run the command "`debchange`" which will prompt you for a `Debian/changelog` entry and update all Debian files as needed.

3.3.3 Generating `Debian/control` from the build system?

You might now ask "wait a minute, when we built an RPM spec file we made it so that `toy-releng.spec` was automatically generated. This was because the `Version:` entry in the spec file needed to match that in `configure.ac`, and also because the `rpmbuild` command allows you to use a spec file embedded in a tarball.

Debian packaging is different enough that at this time I do not see the need for auto-generating the `debian/control` file: it does not embed the version number and the only line in `Debian/control` that might be redundant is the line "`Source:  toy-releng`". Since the package name will not change often (if at all), it is not as important to interlock it to the `PACKAGE` variable in `configure.ac`.

3.4 Release branches and 0.4.0

3.4.1 Release branches: motivation and workflow

With our addition of Debian packaging files we are now ready to release version 0.4.0 of `toy-releng`. We will use a version control idea which is very important when your project becomes complex, and is vital if your project has several contributors.

Version control systems, and DVCSs in particular, have the concept of a *named branch*. This allows you to commit changes to version control without interfering with what other people are doing. There are many ways to use branches to prepare a release without freezing other people's coding efforts. I will describe one approach that is widely used[2] (Mercurial wiki team 2014) and has the advantage that *most contributors continue their work without interference*.

1. The team decides that the current state of the repository is approximately what needs to be in the release. At the same time some team members will probably introduce code that should *not* be in the release.

2. The release engineer creates a branch which will be used for this release (0.4.0) and its subsequent bugfix releases (0.4.1, 0.4.2, ...). Our convention will be to call this branch `branch-release-0.4` to indicate that it will be used for all releases in the 0.4 series.

3. Contributors who are not engineering this release continue working on the trunk.

4. The release engineer works on the code on the release branch, possibly merging in specific changesets from the trunk if something is added to the trunk that has to be in the release.

[2]Among others, the Gnu Compiler Collection (GCC) team uses a very similar approach (The GCC team 2015).

5. The release engineer puts out intermediate releases aimed at testing. The most common convention, which I use here, is to have *alpha* snapshots, *beta* snapshots and release candidates (*rc*). Each of these looks like a release: the release engineer generates tarballs, RPMs and Debian packages for them.

6. After the release is put out the release engineer will do a final merging of the branch back onto the trunk.

7. After the release is put out everyone can forget about the release branch until the time comes to put out a bug-fix release.

3.4.2 Putting out pre-releases and release on a branch

Let us apply this workflow idea to our repository: we will prepare the branch for release 0.4.0. Then we will put out a *pre-release* of the software. Pre-releases versions are discussed at length in Section 6.1.

We start by putting out an alpha release `0.4.0~alpha.0`. Most of the procedure resembles what we have already done in Section 2.3. Note that at this time I am proposing a specific choice of how we name our version. This is a vast topic which I will discuss in Section 6.1.

1. Make sure that all changes are committed and that your collaborators have committed what they want to go into this release. At this point the version in the repository is sometimes called *feature complete.*

2. Create the release branch and update your working copy to that branch. In mercurial the commands are:

   ```
   hg branch branch-release-0.4
   hg -v update branch-release-0.4
   ```

   and from here on all your commits will go to that branch.

3. If you are ready to put out your alpha release you can immediately set the version in `configure.ac`:

   ```
   AC_INIT([toy-releng], [0.4.0~alpha.0], \
       [mark@galassi.org])
   ```

   Listing 3.8: configure.ac

   make sure you commit this change! For example:

   ```
   hg commit -m 'putting out release 0.4.0~alpha.0'
   ```

4. Tag the repository and make tarballs/RPMs/debs:

   ```
   hg tag release-0.4.0~alpha.0
   make distcheck
   rpmbuild -ta toy-releng-0.4.0~alpha.0.tar.gz
   debchange -v 0.4.0~alpha.0
   debuild -us -uc
   ```

5. Move the version in `configure.ac` to `0.4.0~alpha.0plus`:

   ```
   AC_INIT([toy-releng], [0.4.0~alpha.0plus], \
       [mark@galassi.org])
   ```

   Listing 3.9: configure.ac

   and update the Debian version for snapshots with

   ```
   debchange -v 0.4.0~alpha.0plus
   ```

And we have put out an alpha release on this branch. As we fix various issues on the release branch we can put out version `0.4.0~alpha.1`, and so forth as needed for testing. Eventually we will do the same for beta test versions (where we only fix critical bugs; no minor bugs or feature additions). Finally we put out a series with version numbers like `0.4.0~rc.0`. Once those have passed all tests we put out a release with version `0.4.0` and we can ship it.

3.4.3 Branch merging

After we ship the release our release engineer should make sure that any bugs she fixed on the release branch are merged back onto the trunk. The command is simple to write:

```
hg -v update default
hg merge branch-release-0.4
## [resolve conflicts from the merge]
hg commit -m 'merged release branch into default branch'
```

but you will have to pay very close attention to resolving conflicts in the merge.

3.5 Testing installing and upgrading the packages

We can try an amusing and instructive exercise which demonstrates how useful package management can be, and how our release approach allows us to faithfully reproduce past versions of the software.

Let us do this particular exercise with a RedHat-style operating system, although the same would work for Debian-based systems.

If we are in our build directory we can generate tarballs for all our past releases like this:

```
hg -v update release-0.1.0
make distcheck
hg -v update release-0.2.0
make distcheck
hg -v update release-0.3.0
make distcheck
hg -v update release-0.4.0~alpha.0
make distcheck
hg -v update release-0.4.0
make distcheck
```

of course we are have a reputation to defend as hackers, so we might do this:

```
for rel in 0.1.0 0.2.0 0.3.0 0.4.0~alpha.0 0.4.0
do
  hg -v update release-${rel}
  make distcheck
done
ls toy-releng-*.tar.gz
```

or you could improve by taking a look at the output of `hg tags | grep release-` and then trying:

```
REL_TAGS=`hg tags | grep release-`
for rel in ${REL_TAGS}
do
  hg -v update release-${rel}
  make distcheck
done
ls toy-releng-*.tar.gz
```

This kind of scripting that pulls together version control tags and the build system can be very useful when managing a large project comprised of several packages.

After this, for good measure, we can update to the current non-tagged state of the repository with `hg -v update \ default`, which corresponds to version 0.4.0plus. Then `make distcheck` will build us a tarball of 0.4.0plus.

Now that we have all these tarballs we can generate RPMs from them:

```
for tarball in toy-releng-*.tar.gz
do
  rpmbuild -ta ${tarball}
done
```

Note that it would fail on 0.1.0 and 0.2.0 because we did not yet have RPM spec files for those releases!

We now have tarballs for all these releases. We can try to install them in sequence with:

```
sudo yum install -y toy-releng-0.3.0-1.fc20.x86_64.rpm
sudo yum install -y \
     toy-releng-0.4.0~alpha.0-1.fc20.x86_64.rpm
sudo yum install -y toy-releng-0.4.0-1.fc20.x86_64.rpm
sudo yum install -y \
     toy-releng-0.4.0plus-1.fc20.x86_64.rpm
```

Note that our version numbering scheme (discussed at length in Section 6.1) is consistent with what RPM considers to be "newer".

This little exercise has shown that we can faithfully reproduce a previous version of the software thanks our approach to tagging releases.

Chapter 4

Introducing a library, and release 0.5.0

In Section 1.3 I mentioned that release engineering is a gritty and real business, but so far I have only shown you how to release a software package that has a very simple program.

A slightly more complex program which exhibits the features we are interested in is one which:

- Builds and installs a library with a well-defined API.
- Builds and installs an end-user program, which might use that library.

We will design a toy library for use with our toy-releng program. The library will be called `toyreleng` and it will offer rather trivial functions: `tr_square(x)`, `tr_cube(x)` and `tr_hypot(x, y)`, where $\text{tr\_hypot}(x, y) = \sqrt{x^2 + y^2}$. The prefix `tr_` is a common way for C libraries to denote which functions are published and available to be called by programs that use the library.

4.1 A simple library, built and installed by hand

Such a library is often implemented with a few C files which we will call `tr-square.c`, `tr-cube.c` and `tr-hypot.c`, and one header file which we will call `toy-releng.h`:

```
#include "toy-releng.h"

double tr_square(double x)
{
  return x*x;
}
```

Listing 4.1: tr-square.c

```
#include "toy-releng.h"

double tr_cube(double x)
{
  return x*x*x;
}
```

Listing 4.2: tr-cube.c

And a slightly more elaborate function to calculate the hypotenuse given the two sides of a right triangle:[1]

```
#include "toy-releng.h"

double tr_hypot(double x, double y)
{
  return sqrt(x*x + y*y);
}
```

Listing 4.3: tr-hypot.c

```
#ifndef _TOY_RELENG_H
#define _TOY_RELENG_H
double tr_square(double x);
double tr_cube(double x);
double tr_hypot(double x, double y);
#endif /* _TOY_RELENG_H */
```

Listing 4.4: toy-releng.h

The way a traditional *static* library is built is by compiling each C file independently and then joining them in an archive file called `libtoyreleng.a`. This can be done with these commands:

```
gcc -c tr-square.c
gcc -c tr-cube.c
gcc -c tr-hypot.c
ar rvs libtoyreleng.a tr-square.o tr-cube.o tr-hypot.o
cp libtoyreleng.a /usr/local/lib/
cp toy-releng.h /usr/local/include/
```

We could now modify our program `toy-releng-sample.c` to exercise this library:

```
#include <stdio.h>

#include "toy-releng.h"

int main(int argc, char *argv[])
{
  printf("welcome␣to␣the␣package␣%s,␣version␣%s\n",
         PACKAGE, VERSION);
  double x = 3.2;
  double x_sq = tr_square(x);
  double x_cu = tr_cube(x);
  printf("square(%f)␣is␣%f␣--␣cube(%f)␣is␣%f\n", x, \
     x_sq, x, x_cu);
  x = 3;
  double y = 4;
  printf("hypot(%f,␣%f)␣is␣%f\n", x, y, tr_hypot(x, y));
  return 0;
}
```

Listing 4.5: toy-releng-sample.c

and we would compile this program with:

```
gcc -c toy-releng-sample.c
gcc -o toy-releng-sample toy-releng-sample.c -ltoyreleng
```

and a run would look like:

[1] If you vaguely remember that the `hypot` function should be computed differently because of possible overflows, you are right – in Section 5.1 we discuss how this function fails for some arguments and how it can be written more robustly. For now we will pretend that we did not notice this bug creeping in to our software: by ignoring now it we can later look at how to handle bug reports and test suites.

```
$ ./toy-releng-sample
square(3.2) is 10.24 --cube(3.2) is 32.769
hypot(3, 4) is 5
```

This paper is part of a software repository in which everything has been created in the same order as the concepts were introduced into the paper. To demonstrate how we create this library in our approach to software development and release engineering we:

1. Create the files `tr-square.c`, `tr-cube.c`, `tr-hypot.c`, `toy-releng.h` shown above.
2. Add them to version control with:
   ```
   hg add tr-square.c tr-cube.c tr-hypot.c toy-releng.h
   hg commit -m 'added the files needed for \
       libtoyreleng'
   ```
3. Modify our main C program `toy-releng-sample.c` as shown above.
4. Commit the changes to toy-releng-sample.c with:
   ```
   hg commit -m 'added the files needed for \
       libtoyreleng'
   ```

4.2 Building and installing that simple library with automake

The procedure shown in Section 4.1 for building `litboyreleng.a` is reasonably simple, but there are several good reasons to have it built by our build system. Apart from the automation of those steps, inserting it into our build system (specifically in `Makefile.am`) will generalize to very complex library scenarios with almost no effort.

To add a library to `Makefile.am` we add the following lines:[2]

```
bin_PROGRAMS = toy-releng-sample
toy_releng_sample_SOURCES = toy-releng-sample.c
toy_releng_sample_LDADD = libtoyreleng.la -lm

lib_LIBRARIES = libtoyreleng.a
libtoyreleng_a_SOURCES = tr-square.c tr-cube.c tr-hypot.c
include_HEADERS = toy-releng.h
```

Listing 4.6: Makefile.am

note that at this point our sample `.h` file `toy-releng-sample.h` is not useful anymore. We will remove it from `Makefile.am` and from the repository as well with

```
hg remove toy-releng-sample.h
hg commit
## commit message:
## added the files to make our library and removed the
## unnecessary toy-releng-sample.h
```

When we start using shared libraries with libtool in automake we also need to add a couple of lines to `configure.ac`, which now looks like:

```
AC_INIT([toy-releng], [0.4.0plus], [mark@galassi.org])
AM_INIT_AUTOMAKE([-Wall foreign])
AM_PROG_AR
AC_PROG_LIBTOOL
AC_CONFIG_SRCDIR([config.h.in])
AC_CONFIG_HEADERS([config.h])
AC_PROG_CC
AC_CONFIG_FILES([
Makefile
toy-releng.spec
])
AC_OUTPUT
```

Listing 4.7: configure.ac – after we introduce shared libraries

these changes to `configure.ac` are significant enough that we will need to run

```
autoreconf -i
```

Now you can run `make` and `make install` and it will install into `/usr/local/lib/libtoyreleng.a` and `/usr/local/include/toy-releng.h`, after which end user programs can use the library.

4.3 Making libtoyreleng a shared library

Most modern operating systems have a mechanism for linking to libraries at run time instead of bundling all the library code into the executable. The compiler is invoked in a particular way so that one can load and invoke code dynamically.

Building shared libraries is quite annoying, and doing so portably is almost intractable. In the 1990s the GNU libtool project (Matzigkeit et al. 1996) tried to provide a portable way of building shared libraries on the various UNIX-like system and on Windows. The `libtool` program integrates with automake and autoconf to make this almost transparent.

You do little more than changing the two lines in `Makefile.am` that build the library. You change LIBRARIES to LTLIBRARIES, `.a` to `.la`, and `_a` to `_la`:

```
## ...
toy_releng_sample_LDADD = libtoyreleng.la -lm

lib_LTLIBRARIES = libtoyreleng.la
libtoyreleng_la_SOURCES = tr-square.c tr-cube.c tr-hypot.c
## ...
```

Listing 4.8: Makefile.am

After this change the files installed in `/usr/local/lib` will be:

[2]Note that in this paper I sometimes show small portions of the files I modify, but the version control repository is public so you can always clone the repository and "play" through the project as it progresses, obtaining the correct full file at any moment.

```
$ ls -1 /usr/local/lib/libtoyreleng*
/usr/local/lib/libtoyreleng.a
/usr/local/lib/libtoyreleng.la
/usr/local/lib/libtoyreleng.so
/usr/local/lib/libtoyreleng.so.0
/usr/local/lib/libtoyreleng.so.0.0.0
```

where the `.a` file is the static library (which also gets installed), and the files that end with `.so*` are the shared libraries.

End user programs would still link with the same line:

```
gcc -o my-program my-program.c -ltoyreleng
```

although, depending on how your system is configured, you might need to add an option specifying that libraries are installed in `/usr/local/lib`:

```
gcc -o my-program my-program.c -L/usr/local/lib -ltoyreleng
```

4.4 Updating our packaging spec to install the library

Very little has to be done to add this library to our packaging schemes. The Debian package will not change at all, while the RPM spec file just needs a few more entries in its `%files` section:

```
# ...
%files
%doc README NEWS ChangeLog toy-releng-howto.pdf
%changelog
%{_datadir}/%{name}/*
%{_mandir}/man1/*
%{_includedir}/*.h
%{_bindir}/*
%{_libdir}/*
# ...
```

Listing 4.9: toy-releng.spec.in – adding library

4.5 Releasing 0.5.0: pre-releases and final release

The introduction of a library is a significant enough event that we will bump up the minor version number to 5 and release `toy-releng-0.5.0`.

Once we have committed all our work in creating the library we will follow our procedure to create a release branch and then work on that branch to put the release out:

```
hg branch branch-release-0.5
hg -v update branch-release-0.5
```

4.5.1 Finishing the release without noticing the tiny bug

Then we edit `configure.ac` and set the version to `0.5.0~alpha.0` and follow the steps for a release from Section 3.4:

1. Add to the `NEWS` file with end-user-visible release notes. For example:

```
* Noteworthy changes in release 0.5.0 (2014-02-27)
** Features
*** Introduced the libtoyreleng library into the \
     build system
*** Top level program is now toy-releng-sample
*** Added significantly to toy-releng-howto.tex
    Documented the addition of shared libraries.
```

Listing 4.10: NEWS file for 0.5.0

2. Make sure that everything builds and installs with "`make distcheck`"
3. Also confirm that the RPM spec file fits well by running "`rpmbuild -ta toy-releng-0.4.0plus.tar.gz`".
4. Commit to version control.
5. Edit `configure.ac` and change the version from `0.4.0plus` to `0.5.0~alpha.0`
6. Commit to version control.
7. Tag the version control repository with

```
hg tag release-0.5.0~alpha.0
```

8. Prepare RPMs and debs with:

```
make distcheck
rpmbuild -ta toy-releng-0.5.0.tar.gz
debchange -v 0.5.0~alpha.0
debuild -us -uc
```

9. Change the version in `configure.ac` to `0.5.0~alpha.0plus`:

```
AC_INIT([toy-releng], [0.5.0~alpha.0plus], \
     [mark@galassi.org])
```

Listing 4.11: configure.ac – update the version to plus

and update the Debian version for snapshots with

```
debchange -v 0.5.0~alpha.0plus
```

You now release version `0.5.0~alpha.0` to people on your team. They use it and point out that it seems to work well, but they do not see enough documentation. For example, the three library functions are not documented, and there is no real reason to provide a man page for the `toy-releng` program which is not used.

You quickly write man pages for `tr_square`, `tr_cube` and `tr_hypot` in files `tr_square.3`, `tr_cube.3`, `tr_`

[3]See my `toy-releng` mercurial repository for details on these files; I will not list them in full in this paper.

`hypot.3`. These files then get "hg add"-ed to the repository, and you mention them in `Makefile.am`'s `dist_man_MANS` variable so that they get automatically installed. [3]

You are now ready to release `0.5.0`: you update the NEWS file, you update `configure.ac` to have version `0.5.0` in the `AC_INIT` line, then you tag and distribute according to our usual checklist.

4.6 Closing out (for now) the 0.5 release branch

Now that we have put out the 0.5.0 release we want to return to our ordinary development. These are the things you need to do to close out a branch:

1. Make sure that you committed all our changes that were part of 0.5.0 and that you tagged the release with: "`hg tag release-0.5.0`" (you can type "`hg tags`" to make sure).

2. Change the version in `configure.ac` to `0.5.0plus` and commit with "`hg commit -m 'returning to development snapshots'`"

3. Go back to the trunk ("default") branch, and check that you are on the branch you think you are on:

   ```
   hg branch   ## should report branch-release-0.5
   hg -v update default
   hg branch   ## should report default
   ```

4. Merge the updates made on branch-release-0.5 into the trunk:

   ```
   hg merge branch-release-0.5
   ## (do any editing as needed; probably not in \
        this case)
   hg commit -m 'merged branch-release-0.5 into \
        default
   ```

From this point on your future commits will be on the trunk, and thanks to this merge the trunk will include any work you had done on the release branch.

Chapter 5

Bug-fixes, test suites and 0.5.1

I will now demonstrate how we handle the cycle of receiving a bug report, fixing the bug, and putting out a new bug-fix release.

We will imagine the scenario in which a programmer *outside* our company has received the distribution of our library, tried writing a program that uses it, and found a situation in which our functions return incorrect values.

Meanwhile, back at our company, all the programmers have been busily working on introducing new features that will end up in version 0.6.0

5.1 Receiving and verifying the bug report

A programmer called Ada Ritchie received a distribution of our library `libtoyreleng` and tried to call the `tr_hypot` function with the arguments $x = 2.0$, $y = 3.0 \times 10^{154}$. When she printed the value of `tr_hypot(x, y)` she got `inf`, which means that there was a numerical overflow. The correct result, up to a certain precision, is 3×10^{154}.

Ada is a programmer who knows how to submit a useful bug report: she sends us a minimal program that reproduces the bug, together with sample output:

```
/* program which demonstrates a bug in tr_hypot. Install \
    version
   0.5.0 of toy-releng, then compile and run with:

   gcc toy-releng-show-bug.c -o toy-releng-show-bug -\
       ltoyreleng -lm
   ./toy-releng-show-bug

   the output should be:

   tr_hypot(2, 3e+154) is inf
 */

#include <stdio.h>
#include <math.h>

#include "toy-releng.h"

int main(int argc, char *argv[])
{
  double x = 2.0;
  double y = 3.0e154;
  printf("tr_hypot(%g,␣%g)␣is␣%g\n", x, y, tr_hypot(x, y));
  return 0;
}
```

Listing 5.1: `toy-releng-show-bug.c`

We compile and run this sample program according to the directions and find that, indeed, the output is `inf`.

5.2 Returning to the release-0.5 branch

Now that we have *acknowledged* that a bug exists it is very important that we *return to the release branch* to fix it. This is because:

- We will want to send our favorite client programmer, Ada Ritchie, a bug-fix release of our program. This will be version 0.5.1
- We will not want to send any new work that might have been done on the trunk (that stuff is for version 0.6.0), since Ada has already written software which depends on version 0.5.0 and she only wants a fix to 0.5.0 so that she can continue her work.
- But we *do* want to merge the eventual bug fix into the main development stream (default branch) as well.

The way to achieve these goals simultaneously is to add bug-fixes to the release branch, thus not pulling in any material from the trunk, and then to *merge* the updates from the 0.5 release branch into the default branch.

To get started, make sure you are on the release branch:

```
hg branch   ## should report default
hg -v update branch-release-0.5
hg branch   ## should report branch-release-0.5
```

and our subsequent commits will go to `branch-release-0.5`

5.3 Adding a test program

Our client, that sharp programmer Ada Ritchie, has sent us a useful *test program* along with her bug report. We would be wise to include that program into our distribution so we can be certain to catch such bugs if they return.

We massage the program slightly to fit into automake's framework for "test suites" (Vaughan et al. 2000). We place this test program in a new subdirectory called `tests`. We use the C library function `is_normal(z)` to tell us if z is infinite (or some other bad value). We also rewrite the guts of her program to return a non-zero exit code if the test fails:[1]

[1]See the source repository for this full program.

```
int main()
{
  int n_errors = 0;
  double x = 2.0;
  double y = 3.0e154;
  double z = tr_hypot(x, y);
  if (!isnormal(z)) {
    printf("ERROR:␣%g␣is␣not␣normal\n", z);
    ++n_errors;
  }
  /* now try reversing the x and y arguments */
  x = 3.0e154;
  y = 2.0;
  z = tr_hypot(x, y);
  if (!isnormal(z)) {
    printf("ERROR:␣%g␣is␣not␣normal\n", z);
    ++n_errors;
  }
  if (n_errors > 0) {
    return 1;
  } else {
    return 0;
  }
}
```

Listing 5.2: test-hypot-overflow.c

We then update `Makefile.am` to have this segment:

```
check_PROGRAMS = test-hypot-overflow
test_hypot_overflow_SOURCES = tests/test-hypot-overflow.c
test_hypot_overflow_LDADD = libtoyreleng.la -lm
TESTS = $(check_PROGRAMS)
```

Listing 5.3: Makefile.am – introducing a test program

We can now type (after re-running `autoreconf -i`) "`make check`" and this will build our test program, run it, and show us where it fails:

```
==========================================
Testsuite summary for toy-releng 0.5.0plus
==========================================
# TOTAL: 1
# PASS: 0
# SKIP: 0
# XFAIL: 0
# FAIL: 1
# XPASS: 0
# ERROR: 0
==========================================
See ./test-suite.log
Please report to mark@galassi.org
==========================================
```

Listing 5.4: Test suite log

and if we look at `test-suite.log` we will see:

```
[repeat of the earlier messages]
.. contents:: :depth: 2

FAIL: test-hypot-overflow
=========================

ERROR: inf is not normal
ERROR: inf is not normal
```

Listing 5.5: Test suite log

Having added this test suite we should add and commit it with "`hg add tests/test-hypot-overflow.c`; hg commit".

5.4 Fixing the bug

Now we are ready to fix the bug. We do so by *scaling* the arguments to `tr_hypot`. One simple way of doing this is to re-write the guts of `tr_hypot` as:

```
double tr_hypot(double x, double y)
{
  double scale;
  if (x > y) {
    scale = x;
  } else {
    scale = y;
  }
  double a = x/scale;
  double b = y/scale;
  return scale * sqrt(a*a + b*b);
}
```

Listing 5.6: tr-hypot.c with scaling

If we compile `toy-releng-show-bug.c` with this new version of `tr_hypot` we will find that it reports the correct result of 3×10^{154} without numeric overflows.

But even more conveniently and gratifying: we can now type "`make check`" and we should see the output:

```
==========================================
Testsuite summary for toy-releng 0.5.0plus
==========================================
# TOTAL: 1
# PASS: 1
# SKIP: 0
# XFAIL: 0
# FAIL: 0
# XPASS: 0
# ERROR: 0
==========================================
```

Listing 5.7: Output of make check after fixing bug.

Our test has passed, so we commit this new version of `tr_hypot` to the branch: "`hg commit -m 'fixed tr_hypot; test suite passes'`"

5.5 Release 0.5.1

Now we go through the release process: update the NEWS file to read (at the top):

```
* Noteworthy changes in release 0.5.1 (2014-03-10)
** bug fixes
*** fixed tr_hypot to avoid overflow and underflow
** testing
*** set up testing infrastructure for automake
*** added first test program for tr_hypot overflows
```

Listing 5.8: NEWS

We check that we are also satisfied with the output of "`make distcheck`". Then we finalize this release by editing `configure.ac` and setting the version to 0.5.1, then we: "`make distcheck`, "`hg tag release-0.5.1`", edit the version to 0.5.1plus, and follow the procedure in Section 4.6 (reprised in Appendix A) to finish work on the branch and return to development on the trunk.

Chapter 6

Topics – versioning

I hope that the previous chapters have given you an example that you can use to create your own software project, and that you now have a simple procedure to quickly put out carefully reproducible releases.

Since variations in software projects can be quite great, some projects will be much more complex than others, and some additional protocols and tools might be useful in managing larger projects. I discuss some of these topics in this and the following chapters.

6.1 Discussion of version numbering

There are a large number of schemes for specifying software versions, as you can see with a quick tour of the Wikipedia article on "Software versioning." (Wikipedia 2014b)

This is not a good thing: there is no need for that many different schemes (Munroe 2011), so we will propose one which might work for all your projects.

First let us discuss *what we are trying to communicate* with version numbering. There are two possible recipients of our code:

6.1.1 A brand new user of our product

Brand new users don't get much information from a version number, since most of the information in there is to distinguish it from previous versions.

But a widely used convention is that projects will number their versions starting with 0 during early development phases: version 0.1.0 would be a very early one, 0.7.2 would come later, but it would still be considered an early development version.

Then comes the time in which the software is feature-complete, stable, and the developers have committed to not change the interfaces without a proper process. At that point the developers will release it as version 1.0.0

6.1.2 A user who is upgrading from a previous version

A user who had a previous version of the software will want to know the following things:

- Is the project in early development or is it mature?
- Is there significant new functionality?
- Does this break compatibility with previous versions?
- Is this a pre-release leading up to a proper release?
- Is this a development snapshot?
- Is this a bug-fix release?

6.1.3 Semantic versioning

Tom Preston-Werner has proposed a uniform way of naming software versions called *Semantic Versioning* (semver) (Preston-Werner 2013) which defines clear semantic meaning for version numbers.

Version numbers are specified as triads X.Y.Z where X is the *major version number*, Y is the *minor version number*, and Z is the *patch number* (also referred to as *bugfix version number*.)

Pre-releases are specified with a suffix to the X.Y.Z triad, with the form alpha.0, alpha.1, ... for alpha test releases, and the same for beta test releases. You can also have a series of release candidates with the suffix rc.0, rc.1, ...

Semver has answered almost all the questions in Section 6.1.2

- If the version is >= 1.0.0 then the program is mature.
- The major version number is updated when you introduce backward-incompatible changes.
- An increase in the minor number indicates new functionality.
- The pre-release is clearly marked with that suffix.

I find two problems with the current version of semver (2.0.0) for practical deployment.

The first is minor: there is no specified marking for development snapshots *after* a release. Adding the string "`plus`" after the patch number works well: `1.3.2plus` can be used for unreleased development snapshots *after* version 1.3.2.

The second problem is more serious: semver does a good job of expressing the semantics of what version numbers mean, but it also dictates the specific characters that are to go into the version number: a dot (period) between numbers, and a dash (hyphen) between the main version and the pre-release portion.

The problem with specifying syntax details is that it might be incompatible with how widely adopted packaging systems already do business. In particular, RPM and Debian have already specified what characters are allowed to go into a version number.

The biggest concern is using a hyphen in version names: `1.3.2-alpha.0`, for example, will not be a valid RPM version number, and it might also not work well with GNU tarballs and Debian packages. There are ways of tricking RPM into accepting the version number. One way is to remove the hyphen and use `1.3.2alpha.0`, but when you do that the version will be considered *bigger* than `1.3.2`, whereas you want pre-releases to be considered smaller. In practice the packaging system will refuse to *upgrade* from `1.3.2alpha.0` to `1.3.2`, which is clearly a very serious problem.

My proposal here is to use the tilde character (~) instead of the hyphen to separate the version and the pre-release string. The tilde character is treated specially by both Debian and RPM packaging systems: it has the lowest possible precedence in version ordering, which means that we have a guarantee that: $\texttt{1.3.2~alpha.0} < \texttt{1.3.2}$

This proposal of using the tilde to separate pre-releases is what I have used in the repository that accompanies this paper.

6.1.4 My full proposal for versioning

This proposal is based on using all the *semantics* of the semver 2.0.0 specification (Preston-Werner 2013).

I then propose using almost all of the *syntax* from the semver specification, with the following exceptions:

- Pre-releases (alpha, beta, release candidates) may be denoted by adding a ~ (tilde) and the string alpha, beta or rc, followed by a snapshot number. For example, leading up to version `0.5.1` we can have a sequence of testing pre-releases numbered:

$$\texttt{0.5.1~alpha.0} < \texttt{0.5.1~alpha.1} < \texttt{0.5.1~beta.0} < \texttt{0.5.1~rc.0} < \texttt{0.5.1}$$

- Unreleased post-release development snapshots may be denoted by adding the string "`plus`" after the patch number, with no separation. For example, after releasing `0.5.1`, unreleased snapshots will be marked as `0.5.1plus`

- (minor issue) The "build metadata" portion of the specification should be handled by the packaging "release" string, rather than being part of the software version number. The reason for this is that semver considers "build metadata" to be useful information, but void of semantic meaning.

Given that I cited Munroe's XKCD commentary on the proliferation of standards (Munroe 2011) as a compelling demonstration of the problem of "too many developers reinventing the wheel", I feel that I have to justify the fact that I am changing the standard (and thus adding to the proliferation).

The issue of "post-release development snapshots" is important. Look at this scenario: we release version 0.5.1 of our software, and then we start making changes to it. If we leave the version number unchanged, then every time we put out a snapshot the end user would report a bug against version 0.5.1. When we get this bug report we would not know if it is a big problem we have to worry about (a bug in the official 0.5.1 release) or something less urgent to investigate (a bug in a development snapshot). We often send *snapshots* to experienced end users, but we would send only proper *releases* to inexperienced end users. Knowing that an inexperienced end user is reporting a bug against a snapshot would help the software management process: we would immediately tell him to please use a stable release instead of a snapshot.

The other significant issue is the use of ~ (tilde) instead of '`-`' (hyphen) to separate the version from the pre-release information. This is also important because, as stated above, the two packaging systems used in almost all GNU/Linux distributions will refuse to upgrade from an alpha or beta to a final release if the original semver '`-`' (hyphen) is used.

6.2 Shared library versioning

Shared libraries have been the standard way of shipping a compiled software library for a very long time. Instead of linking all the library code into a single very large executable, the shared libraries are installed separately on the system, and when the main program needs that code it loads the libraries dynamically at run time.

This brings up a possible serious problem: if the main program and its libraries live in separate files, and are possibly distributed independently, how do you know that they are compatible with each other? An update of the shared library could change its behavior and cause the main program to crash or produce incorrect results.[1]

It is considered better for the program to exit with a shared library "incompatible version" error message than for it to crash mysteriously or produce incorrect results.

There are a few possible versioning schemes for shared libraries. The most important thing such a scheme must do is refuse to load an incompatible shared library into a program, but there are other aspects to it.

The libtool documentation (Matzigkeit et al. 1996) has a detailed chapter in which they describe a standard 3-number versioning scheme for shared libraries. While the

[1] Shared library versioning is one of the bigger aspects of a problem known in the Microsoft Windows operating systems as *DLL Hell* (Wikipedia 2014a). Another aspect of DLL Hell is not finding the shared library in the first place. GNU/Linux software which does not carefully handle shared library versioning can end up with a sort of DLL Hell.

concept of the shared library version numbers is related to the three numbers in semantic versioning, there are very important differences and shared library versioning should be studied as a separate topic: there should be absolutely no link between the software version and the shared library version.

Please read the libtool manual's description carefully before your program approaches version `1.0.0`. You will find a careful 6-step procedure on how to handle shared library versions, which they mark as "`CURRENT:REVISION:AGE`". You will need to have these procedures firmly in place, and to practice them on pre-1.0.0 releases so you can claim that you are releasing stable software.

Chapter 7

Topics – project management tools

7.1 Bug and issue tracking

Many software projects begin with an informal approach to bug tracking, but it is important to move beyond that as soon as possible: your end users need to know which bugs have been fixed in a new release, so you need to set up your tools and process to make this clear.

Here is a basic checklist for bug tracking:

1. Assign the bug a unique identifier when you receive it.
2. Verify the bug against a specific release of the software.
3. Write a program (or develop some other procedure) to reproduce the bug and add it to your test suite, as demonstrated in Section 5.3.
4. Mark that the bug has been fixed in a few places: the `ChangeLog` file (and/or the version control commit message) for all tracked bugs, and the `NEWS` file for major bug fixes. The notes yout place should reference the bug tracking number.

It would be a valid process to receive bug reports by email and store all this information in a text file, but as projects grow in size it might become difficult to manage this process by hand.

There are software systems which automate the recording of bug reports, and which allow you to track which ones get resolved in various software releases. One that is often used is the `Trac` project (Trac team 2014). Their suite of web-based programs link bug reports (“tickets”) to the version control system, and offer some other features to facilitate project coordination. Many others bug tracking systems are available.

7.2 Continuous integration tools

As your project grows in complexity you will probably want to automate how you build and test your software, and make the tests happen regularly, possibly every night, so that you get an early warning if a bug has been introduced or re-introduced.

There are tools which help with this process; they are referred to as continuous integration (CI) tools. They typically clone your repository every night, build the software, and can run the test suite in several different configurations and circumstances.

The most widely used continuous integration tool in the free software world is probably Jenkins (Smart 2011), which presents a web interface allowing team members to configure tests and produce reports on these tests.

7.3 Public hosting

Hosting a project is the provision of servers to run the various collaborative tools. Although strictly one could use distributed version control and never need a central server, in practice many teams choose to host the following services in a single central server:

- Version control.
- Mailing lists.
- Documentation repositories (for documents that do not end up in the project distribution). This is often a WIKI system allowing media storage.
- Bug tracking.
- Continuous integration.

I have touched on most of these topics individually, but there are many other tools that can facilitate collaboration, such as blogging, microblogging, web-based forums, . . .

There are two aspects to hosting these services which are sometimes confused, but should not be:

- The software package which bundles together and coordinates all these services, and
- A web server, administered by some organization, which allows members of the public to set up pages for their own project using one of the aforementioned software packages.

If you want your project to be hosted you can either use your own web server and configure it to run the software you need, or you can use one of the public hosting servers. The first widely used ones were `https://sourceware.org/` (used internally at Cygnus and later RedHat), and `http://sourceforge.net` (which was the first to allow projects from the general public). The GNU project offers hosting for many of its projects at `https://savannah.gnu.org` and `https://savannah.nongnu.org`

The advent of distributed version control systems, such as Mercurial and Git, as well as general advances in client and server side web capabilities, prompted the development of a new generation of hosting servers such as `https://gitlab.org/` (git), `https://github.com/` (git) and `https://bitbucket.org/` (mercurial and git).

At this time I do not have specific recommendations on which hosting server to use. I only encourage the adoption of free software tools, and many of these are based on a proprietary web infrastructure, which would cause damage to projects in the long term since they would not have the freedom to transfer their work to another setup if their provider were to change its terms or disappear.

The more frequently used free software public hosting sites at this time have practical limitations (GNU Savannah depends on the limited financial resources of the GNU project, and gitlab allows git but not mercurial).

My advice here is to survey the current state of public hosting servers[1], as well as the options for hosting on your own servers, and make a choice as your project is starting. But do not commit too much to the specifics of one approach, and be prepared to migrate to a different host if necessary: many projects have had to do it.

[1] One place to start is `http://en.wikipedia.org/wiki/Comparison_of_open_source_software_hosting_facilities`

Chapter 8

Topics – more about the world of packaging

Software packaging is an important, ubiquitous and rapidly changing area of software engineering. I will discuss some of the issues that surround packaging, trying to avoid the rapid obsolescence that lurks around any statement on this subject.

In the "self referential HOWTO" that constitutes the first part of this paper, leading up to the release of version 0.5.1, I referred to packaging often, giving a recipe for preparing Red Hat (RPM) and Debian (deb) packages for `toy-releng`.

But I did not cover two important parts of the world of packaging.

First: there is a strong point of view that the author of a program should not be the same person who prepares the packages.

Second: it is important to understand a bit more about how packages fit together into large scale coordinated package repositories.

8.1 Who should do the packaging?

When a new major GNU/Linux distribution release is about to come out, or when a new version of an important package comes out, the packaging team for that distribution creates (or updates) the RPM spec file or the Debian packaging description files.

The people who do this are experts, they do it quickly and precisely, and they follow the careful guidelines of their GNU/Linux distribution.[1]

The packaging team might find your attempt at providing an RPM spec file to be misguided and not up to their standards, so they will either write one from scratch, or use yours as a starting point to write their own.

They also might apply some patches to the source code that you distribute.

They will then maintain their packaging specification and source patches *separately from the package source code!*

You might wonder whey I had you go to the trouble of creating and exercising packaging for `toy-releng`. There are a couple of reasons:

1. It is unlikely that your package will be picked up to be included in one of the major distributions, so the only package specs your end users will get will be the ones you provide.
2. If and when your package is picked up by a major distribution, you will still want to test the packaging in-house.
3. It doesn't hurt the distribution maintainers to have a package template to start from, even if they barely use it.
4. Testing your software as it gets installed by the package manager is a very good test of software robustness.

For these reasons I recommend continuing to maintain and distribute packaging specifications for RPM and Debian.

8.2 Coordinated package repositories (apt, yum)

When you install a Red Hat based system (such as Red Hat Enterprise Linux, CentOS, SUSE or Fedora), or when you install a Debian based system (such as Debian or Ubuntu GNU/Linux), you *almost never* hunt down the individual packages that you need and then run the RPM or dpkg commands directly to install them.

What you do, instead, is use a *coordinated package repository.*[2] Let us say, for example, that you have heard of the legendary program `rdiff-backup` and you want to install it on your system.

On RPM-based systems you will type

```
yum search all rdiff-backup
```

and on Debian-based systems you will type:

```
apt-cache search rdiff-backup
```

In both cases you will find out that the package is indeed called `rdiff-backup`, and you will be able to install it with, respectively:

```
sudo yum install rdiff-backup
sudo apt-get install rdiff-backup
```

[1]Note that other distributions, such as the Berkeley UNIX variants, also make such an effort. Proprietary operating systems do very little of this since they don't consider it their task to offer anything beyond the minimal operating system and a few bundled utilities.

[2]The word *repository* comes up in several contexts; we have now seen it used for version control repository (a collection of files) and package distribution repositories (a collection of RPM or deb packages, coordinated in some way.)

What the programs yum and apt-get are doing for Red Hat and Debian respectively is to use the staggeringly large effort by the distribution maintainers to *coordinate* a great number of packages for their distribution. The packages are all built against an identical infrastructure, so that the binaries will all work.

8.2.1 A tale from 1999

To demonstrate how important this is I will give you a bit of history about coordinated packaging. We will look at a snapshot of the GNU/Linux world around the year 1999. Both the Red Hat and Debian distributions were widely used. When you installed Red Hat Linux, at the time version 6.0, you got a certain number of packages with all the core functionality. These were officially supported by Red Hat, and were coordinated and binary-compatible with each other.

Beyond this basic officially supported set of packages there were several web sites which collected RPMs for many packages not directly supported by Red Hat, and there was never guarantee that a package would work well with other packages.

So what would happen if you released a package that relied on a library outside of the supported Red Hat core, for example a scientific library? You would probably find a third party RPM of that library, so you could build your own package against it and distribute your own RPM together with a copy of the third party library RPM.

Now imagine that someone else also built a program which needed that same library. They found a *different* user-provided RPM for that same library. Now if someone wanted to install both your binary RPM and this other one, they would not be able to because there would be incompatible binary versions of this shared library. We would be back in the world of DLL Hell mentioned above.

By contrast, back in 1999 the Debian GNU/Linux distribution followed a different policy: Debian has a large team of volunteer maintainers who coordinate a very large number of packages for the distribution. Even back in 1999 it would be very unlikely that you would need user-provided RPMs for anything, let alone an infrastructure library. Debian had also developed the `apt-get` tool which automatically pulled the other programs needed to support the program you request. Nothing like that was in widespread use for the RPM-based distributions.

8.2.2 How it is today

This story highlights the importance of having a very large core collection of coordinated packages. Since 1999 the RPM-based distributions have started using centrally coordinated package repositories. For example, at the time of writing, when you install a Fedora 20 (RPM-based) distribution, there is a large team of maintainers doing the same thing that the Debian maintainers do: they offer coordinated and up-to-date binary packages for a very large number of packages. As I have mentioned, in RPM-based distributions the tool used is called *yum*, which works analogously to Debian's *apt-get*.

The more commercial software distributions, like Red Hat Enterprise Linux, offer a much smaller collection of supported packages. This is because they offer commercial support, and are not willing to support too many packages. To get around this limited commercial offering, the Fedora project volunteers have put together the `EPEL` effort: additional repositories, coordinated with the Red Hat Enterprise base packages, which make for a much larger core of coordinated RPMs.

8.2.3 How does this affect our release engineering?

Apart from being very useful for end users and developers, the notion of a coordinated package repository can be very useful to us. One way to distribute your programs is to offer a yum repository for RPM-based distributions and an apt repository for Debian and Ubuntu.

This way your users do not have to hunt for the software: they will receive it automatically when they type `sudo yum update` or `sudo apt-get update; sudo apt-get \ dist-upgrade`.

It takes a small effort to set these repositories up, but once you have gone to that trouble the process of distributing your new releases is remarkably smooth.

The Fedora documentation team has a simple guide to creating a Yum repository (Fedora documentation team 2014), and the Debian team has an analogous guide (Debian wiki team 2014).

Chapter 9

Topics – miscellaneous notes and tricks

9.1 Adding version info to LaTeX

A mismatch between documentation and software can make for a poor user experience as well as obfuscating whether bug reports are due to incorrect use of the software.

One could address this by adding a footnote to the title that clarifies it:

```
 \title{A self-referential HOWTO on release \
     engineering\footnote{For
    use with software version 0.5.1}}
```

Listing 9.1: s/w version in doc title

but this is error-prone because it requires human input instead of being automatic. In hardware safety engineering we use the term "engineering interlock" to describe a system that guarantees the coupling of two separate things (in our case: the software version number and the documentation subtitle) and does not rely on a human remembering to take a step.

In Section 2.6 I showed how the program `toy-releng-sample.c` can use information from the build system. Here I show that many tricks are possible to take this interlock further. Since our LaTeX documentation does not have access to the C preprocessor symbols in `config.h.in` and `config.h`, we make a new file called `VERSION.input.in`

```
\def \PAPERVERSION {@VERSION@}
```

Listing 9.2: VERSION.input.in

If we then add to `configure.ac` the appropriate line:

```
# [...]
AC_CONFIG_FILES([
Makefile
toy-releng.spec
$srcdir/howto-paper/VERSION.input
])
AC_OUTPUT
```

Listing 9.3: configure.ac with VERSION.input

then `configure` will transform this into a file called `VERSION.input`:

```
\def \PAPERVERSION {0.5.1plus}
```

Listing 9.4: VERSION.input

which we can include into our paper by putting this line at the top of `toy-releng-howto.tex`:

```
\documentclass{report}
\input{VERSION.input}
% [...]
```

Listing 9.5: VERSION.input

9.2 Other build systems

In this paper I have used GNU autotools as a build system for all the examples. This is because the autotools currently have an edge over other options when it comes to packaging and release engineering.

There are several other approaches people have used to build their software. They all have advantages (which are sometimes just *apparent* advantages but give problems when projects get more complex) and disadvantages.

I will briefly mention some of them, but be aware that most of these sections are to give you background information – the only build system you should take seriously is cmake (Section 9.2.5).

Before describing these alternatives I will emphasize one of the more important aspects of the GNU-style release tarballs that we have been using in our toy project.

9.2.1 The beauty of the standard `make` targets

When you first learn to use make and build systems you think that you will only need a few targets when you compile, for example:

```
make
make doc
make install
```

But when you have built your first non-trivial project, and have had to think about how it will be packaged, as we have done in this self-referential HOWTO, you will find that you really need most or all of the exotic-seeming targets described in the GNU automake manual.

You also need to be able to compile with one prefix, and install with a different prefix, since this is crucial for the packaging system to prepare binary packages. And you need to perform a staged installation using the `DESTDIR` variable.

If you are not yet convinced of this you should look carefully at how RPM and Debian packaging prepares a directory tree with all the correct files, and then plucks those files out to put them into the package bundle.

The typical sequence used by the package builder looks like this:

```
./configure --prefix=/usr
# configuring like this, the program knows
# its files are under /usr
make
make DESTDIR=/var/tmp/DUMMY_TARGET install
```

Listing 9.6: build sequence for package preparation

after which the binaries will be in `/var/tmp/DUMMY_TARGET/usr/bin` and so forth, but all these files will be configured to work out of `/usr/bin`, `/usr/lib`, ...

The result is that the package manager is now ready to install these files in `/usr/...` on the target computer, and this procedure never trampled those directories on the build computer.

9.2.2 A shell script with a name like `compile.sh`

People often start with a single shell script which compiles all their code together into an executable. This is not an option for any project that goes beyond a couple of source files: re-compiling all the files takes too long, and there is no structure which would allow packaging systems to install and enumerate all the important files.

9.2.3 Clever hand-crafted Makefiles

For small projects you might feel some relief at using a simple Makefile to build and install your programs. This does not scale well to more complex projects, but the GNU version of make has several enhancements over the traditional UNIX make, and this has tempted some designers of large software systems to create include files for GNU make.

These "clever" Makefiles allow you to organize a set of directories, set some variables in your own Makefile, include a file that might be called something like `make_macros.mk`, and everything will build.

Clever Makefiles end up suffering from various problems when your project grows. Make was originally not aimed at enforcing a structure that works well for packaging, not to mention shared libraries and other facilities.

You would need to include a lot of extra structure into your Makefiles, and you would end up recreating a lot of the functionality that the autotools give you.

9.2.4 Drop-in build systems arranged by one of your infrastructure pieces

There are some large software systems with a collection of libraries that you are supposed to link to and which provide you with a custom build system that hides the details of linking to their libraries.

Examples of this are:

The *qt widget set* has a program `qmake` that does everything for you so that you do not have to find all the libraries you need to link to, nor do you need to figure out the paths to the .h files you are `#include`-ing.

MPI The message passing interface is the most common tool for massively parallel programming: MPI implementations offer a wrapper around the C/C++ compiler called `mpicc`.

Geant4 (Agostinelli et al. 2003) is one of the most important particle physics simulation programs, developed at CERN to support the Large Hadron Collider and other efforts, but now used throughout the particle physics community. Geant4 originally provided examples with a drop-in directory structure in which you would drop your C++ source code and the compilation would "just happen". Today the authors have moved away from that approach, both for their own development and for examples: Geant4 example code now comes with template CMake project files.

The problem with the approach of drop-in Makefiles is that *every one of these packages wants to own your application.* If you use more than one of them, you cannot coordinate their drop-in build approaches!

There is, of course, always a way to just find where the .h, .a and .so files provided by these libraries live and to come up with your own linking instructions.

My strong proposal here is to never use the drop-in build systems, but rather determine if the library providers have also offered a way of locating where their files are so that you can build your code in a portable way.

The most robust approach is to have a script called `myprogram-config` which takes options `--libs` and `--cflags` and so forth. To compile a program that uses the GNU Scientific Library, for example, you type

```
gcc `gsl-config --cflags` mymathprogram.c `gsl-config --libs`
```

Listing 9.7: compiling with GSL

and you can combine this with similar ways of finding the flags needed for all your other infrastructure libraries. There is also a framework for such `-config` programs called `pkg-config`, which allows you to write a descriptor file for your library.

Some programs that used to offer drop-in build systems have now moved to offering a `-config` script.

9.2.5 CMake

CMake (Martin et al. 2010) is a cross-platform build system which fills approximately the same niche as the GNU autotools.

Being of more recent design, CMake gets around some of the criticisms that are brought to the autotools, such as the confusing collection of intermediate supporting files and scripts that are created, as well as the several build steps.

On the other hand CMake does not provide one of autotools's most important outputs: a GNU-compliant tarball which can install on a computer that does not have advanced developer tools.

The lack of GNU-compliant tarballs also means that RPM and Debian packaging are not as straightforward to produce, although CMake provides rich enough Makefiles that the packaging specifications are not very difficult to produce.

At the time of writing CMake is beginning to provide tools to build packages: the new CPack program automates many of the packaging tasks.

CMake is free/open-source software and is certainly a valid alternative to the GNU autotools. At this time CMake is used in many large and complex free software projects. While autotools retains a small edge, I consider CMake viable for complex projects.

9.3 Virtual machines

Your release testing should be done on a great variety of computer configurations.

This will flush out subtle bugs and improve the user experience: fewer users will have those problems, since you already discovered them in testing.

In the last few years virtualization software has reached maturity, and CPU support for virtualization is now widespread. The result is that in a few minutes you can install several operating systems in virtual machines, configure them all differently, and thus simulate many possible end-user setups.

You might hear the point of view that it is a waste of time to release for anything other than your advertised platform – a manager, prompted by a marketing analysis, will say: "we only support Red Hat Enterprise Linux 6", or "we only support Windows XP". But independent of what platform you officially support, you should know that: *your software will be much more robust on all platforms if you test it widely.*

I always aim to test the software I release on the following GNU/Linux distributions: Debian unstable, Debian testing, the latest Fedora, Ubuntu (both the current and the latest Long Term Support releases), CentOS 7, 6 and 5 (CentOS is community-supported and binary compatible with Red Hat Enterprise Linux). It is straightforward to automate testing on virtual machines with all these operating systems. [1]

[1]If needed you can also test on Microsoft Windows, using the Cygwin or MinGW environments to provide POSIX compatibility. I periodically check the state of the ReactOS operating system to see if it has reached the point where I can try to compile my code on it.

Chapter 10

Acknowledgments

This work was supported by the U.S. Department of Energy's National Nuclear Security Administration as part of the DIORAMA project in the Satellite Nuclear Detonation Detection program.

I would like to thank collaborators from my entire career who have shown me how they release software, and have provided feedback on my procedures.

In particular I thank: Tom Tromey who introduced me to automake back in April 1996; Christopher Gabriel with whom I developed (in 1999) a *parody* of modern project development which proposed applying very careful release engineering approaches to vaporware - the release engineering approaches were quite serious and are similar to what I present in this paper; Michael Fischer with whom I developed (in 1990) my first release engineering approach for the Dominion world simulation game.

The Diorama project in Los Alamos is a very elaborate and sophisticated software project in which I have been able to explore advanced and subtle aspects of release engineering.

And finally my thanks to Laura Fortunato for suggestions on the pedagogical approach and for a remarkably careful job proofreading this paper, trying out all the examples, and correcting several mistakes in the step-by-step procedures.

Appendix A

Collected release checklists

A.1 Starting a project

1. Create a version control repository. Use a distributed version control system (such as Mercurial or Git). Add every file you author to the repository early, and commit changes to all your files often.
2. Create a project roadmap. This will accompany you through early releases. Eventually your project will diverge from the initial roadmap, but it is crucial to write down your design early on. In the project that accompanies the tutorial, this is in the file project-admin.org
3. Create your first source code files.
4. Set up a build system as soon as you have your first source code files.
5. Exercise "make" and "make check" immediately.
6. Write documentation immediately and develop it in step with the software.

A.2 Leading up to a release

For this example let us say that the release is 0.4.3, the previous one was 0.4.2, and currently `configure.ac` has the version set to 0.4.2plus.

1. Make sure you are conceptually happy with what you have done and you understand what goes into this release.
2. Make a release branch (or update to that branch if it already exists). For example: `hg branch \ branch-release-0.4` or `hg -v update branch-release-0.4`
3. If any changesets from the trunk are needed in this release, merge them into the release branch.
4. Make sure that the software performs well when you run it in your own development environment.
5. Update the NEWS file with release notes describing end-user-visible changes.
6. Make sure that everything builds and installs with "`make distcheck`"
7. Also confirm that the RPM spec file fits well by running "`rpmbuild -ta toy-releng-0.4.2plus.tar.gz`".
8. Commit to version control.
9. Set the version for your test release in `configure.ac`, for example `0.4.3~beta.1`. Remember that alpha releases are little more than snapshots, but once you put out a beta release you can only make changes to:
   (a) Fix critical bugs.
   (b) Update documentation.
10. Commit to version control.
11. Tag the repository with `hg tag release-0.4.3~beta.1`
12. Edit configure.ac and add a "plus" to the version: `0.4.3~beta.1plus`.
13. Commit to version control, for example with `hg \ commit -m 'returning to development snapshots'`

A.3 Putting out the final release

1. Update to the release branch with `hg -v update \ branch-release-0.4`
2. Do a final check that the NEWS file is complete.
3. Edit configure.ac to set the version to (for example) `0.4.3`
4. Tag the version control repository with

   ```
   hg tag release-0.4.3
   ```

   This tagging step is extremely important because it guarantees that you can always reproduce what went into version 0.4.3 from your version control repository. This *reproducibility* will become very important as your project grows, but even now you should make a habit of it.
5. You can now make the tarball with

   ```
   make distcheck
   ```

6. Prepare RPMs as shown in Section 3.1.4 and the Debian package as shown in Section 3.3.2
7. Make your source tarball as well as your binary packages available to your end users.

8. Edit configure.ac and add a “plus” to the version: `0.4.3~beta.1plus`.

9. Commit to version control, for example with

   ```
   hg commit -m 'returning to development snapshots'
   ```

10. Merge work on the release branch back in to the default branch.

Bibliography

Agostinelli, S et al. (2003). "GEANT4 - a simulation toolkit". In: *Nuclear instruments and methods in physics research section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 506.3, pp. 250–303.

Atalassian, Inc. (2015). *Bitbucket Project Hosting.* Accessed: 2015-11-24. URL: `https://bitbucket.org/`.

Debian wiki team (2014). *How To Setup A Debian Repository.* URL: `https://wiki.debian.org/HowToSetupADebianRepository`.

Fedora documentation team (2014). *Creating a Yum Rpository.* URL: `http://docs.fedoraproject.org/en-US/Fedora/14/html/Deployment_Guide/sec-Creating_a_Yum_Repository.html`.

Fedora team (2013a). *Fedora Packaging Guidelines.* URL: `http://fedoraproject.org/wiki/Packaging:Guidelines`.

— (2013b). *How to create an RPM package.* URL: `http://fedoraproject.org/wiki/How_to_create_an_RPM_package`.

Galassi, Mark, James Theiler, Brian Gough, et al. (2009). *GNU scientific library: reference manual.* A GNU manual. Bristol, UK: Network Theory. ISBN: 0-9546120-7-8.

Mackall, Matt (2006). "Towards a better SCM: Revlog and Mercurial". In: *Linux Symposium*, p. 83.

Martin, Ken et al. (2010). *Mastering CMake: A cross-platform build system.* Kitware Incorporated.

Matzigkeit, Gordon et al. (1996). *GNU Libtool.*

Mercurial wiki team (2014). *Branching and merging in Mercurial (and Git) explained.* URL: `http://mercurial.selenic.com/wiki/BranchingExplained`.

Munroe, Randall (2011). *XKCD #927: Standards.* URL: `https://xkcd.com/927/`.

Preston-Werner, Tom (2013). *Semantic Versioning 2.0.0.* URL: `http://semver.org/`.

Rodin, Josip and Osamu Aoki (1998-2013). *Debian New Maintainers' Guide.* URL: `https://www.debian.org/doc/manuals/maint-guide/index.en.html`.

Smart, John (2011). *Jenkins: The Definitive Guide.* Oreilly and Associate Series. O'Reilly Media. ISBN: 9781449305352.

Stallman, Richard (Mar. 1985). "The GNU Manifesto". In: *Dr. Dobb's Journal of Software Tools* 10.3, pp. 30–. ISSN: 1044-789X.

Stallman, Richard et al. (1992-2013). *GNU coding standards.* URL: `http://www.gnu.org/prep/standards/`.

The GCC team (2015). *GCC Development Plan.* Accessed: 2015-11-24. URL: `https://gcc.gnu.org/develop.html`.

Trac team (2014). *the Trac Open Source Project.* URL: `http://trac.edgewall.org/`.

Vaughan, GV et al. (2000). "GNU Autoconf, Automake, and Libtool: Expert insight into porting software and building large projects using GNU Autotools". In: *New Riders, Indianapolis.*

Wikipedia (2014a). *DLL Hell — Wikipedia, The Free Encyclopedia.* [Online; accessed 22-March-2014]. URL: `http://en.wikipedia.org/w/index.php?title=DLL_Hell&oldid=594200654`.

— (2014b). *Software versioning — Wikipedia, The Free Encyclopedia.* [Online; accessed 24-February-2014]. URL: `http://en.wikipedia.org/w/index.php?title=Software_versioning&oldid=596539144`.