Ubuntu Packaging Guide

Versão 1.0.0 bzr651 ubuntu14.04.1

Ubuntu Developers

16/03/2018

Sumário

Welcome to the Ubuntu Packaging and Development Guide! We are currently developing codename Bionic Beaver, which is to be released in April 2018 as Ubuntu 18.04 LTS.

This is the official place for learning all about Ubuntu Development and packaging. After reading this guide you will have:

- Heard about the most important players, processes and tools in Ubuntu development,
- Your development environment set up correctly,
- A better idea of how to join our community,
- Fixed an actual Ubuntu bug as part of the tutorials.

O Ubuntu não é somente um sistema operacional livre e de código-aberto, sua plataforma também é aberta e desenvolvida de maneira transparente. O código-fonte de cada um dos componentes pode ser obtido facilmente e toda mudança na plataforma do Ubuntu pode ser revisada.

Isto significa que você pode se envolver ativamente na melhoria dele e a comunidade de desenvolvedores da plataforma Ubuntu está sempre disposta a auxiliar seus pares a iniciar.

O Ubuntu também é uma comunidade de boas pessoas que acreditam no software livre e que este deve estar acessível a todos. Seus membro são amistosos e querem que você se envolva também. Queremos que você se envolva, faça perguntas, para fazer o Ubuntu melhor junto conosco.

Se você encontrar problemas, não entre em pânico! Confira o *artigo sobre comunicação* e você irá descobrir a maneira mais fácil de entrar em contato com outros desenvolvedores.

O guia é dividido em duas seções:

- Um lista de artigos baseada em tarefas, coisas que você quer fazer.
- Um conjunto de artigos de base de conhecimento que se aprofunda em partes específicas de nossas ferramentas e fluxos de trabalho.

CAPÍTULO 1

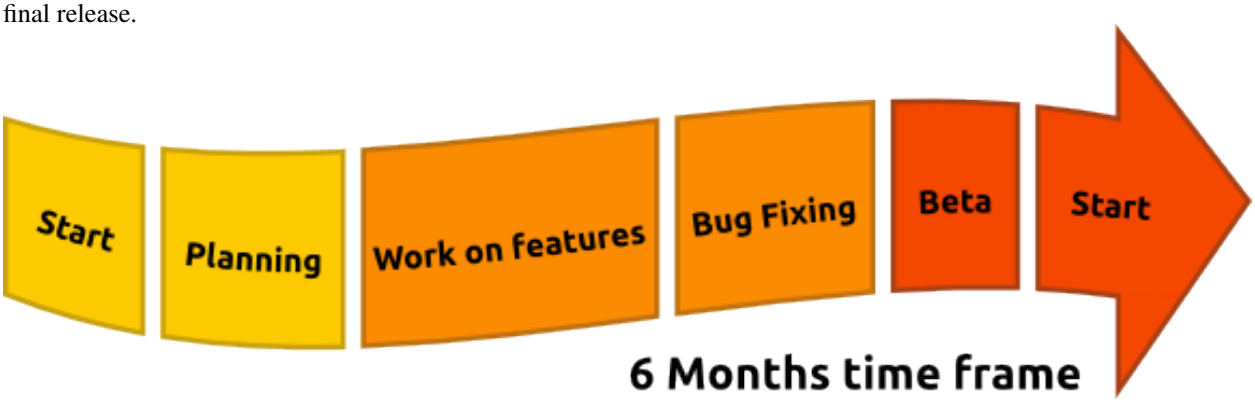
Artigos

1.1 Introdução ao desenvolvimento Ubuntu

Ubuntu is made up of thousands of different components, written in many different programming languages. Every component - be it a software library, a tool or a graphical application - is available as a source package. Source packages in most cases consist of two parts: the actual source code and metadata. Metadata includes the dependencies of the package, copyright and licensing information, and instructions on how to build the package. Once this source package is compiled, the build process provides binary packages, which are the .deb files users can install.

Every time a new version of an application is released, or when someone makes a change to the source code that goes into Ubuntu, the source package must be uploaded to Launchpad's build machines to be compiled. The resulting binary packages then are distributed to the archive and its mirrors in different countries. The URLs in `/etc/apt/sources.list` point to an archive or mirror. Every day images are built for a selection of different Ubuntu flavours. They can be used in various circumstances. There are images you can put on a USB key, you can burn them on DVDs, you can use netboot images and there are images suitable for your phone and tablet. Ubuntu Desktop, Ubuntu Server, Kubuntu and others specify a list of required packages that get on the image. These images are then used for installation tests and provide the feedback for further release planning.

Ubuntu's development is very much dependent on the current stage of the release cycle. We release a new version of Ubuntu every six months, which is only possible because we have established strict freeze dates. With every freeze date that is reached developers are expected to make fewer, less intrusive changes. Feature Freeze is the first big freeze date after the first half of the cycle has passed. At this stage features must be largely implemented. The rest of the cycle is supposed to be focused on fixing bugs. After that the user interface, then the documentation, the kernel, etc. are frozen, then the beta release is put out which receives a lot of testing. From the beta release onwards, only critical bugs get fixed and a release candidate release is made and if it does not contain any serious problems, it becomes the final release.



Thousands of source packages, billions of lines of code, hundreds of contributors require a lot of communication and planning to maintain high standards of quality. At the beginning and in the middle of each release cycle we have the

Ubuntu Developer Summit where developers and contributors come together to plan the features of the next releases. Every feature is discussed by its stakeholders and a specification is written that contains detailed information about its assumptions, implementation, the necessary changes in other places, how to test it and so on. This is all done in an open and transparent fashion, so you can participate remotely and listen to a videocast, chat with attendants and subscribe to changes of specifications, so you are always up to date.

Not every single change can be discussed in a meeting though, particularly because Ubuntu relies on changes that are done in other projects. That is why contributors to Ubuntu constantly stay in touch. Most teams or projects use dedicated mailing lists to avoid too much unrelated noise. For more immediate coordination, developers and contributors use Internet Relay Chat (IRC). All discussions are open and public.

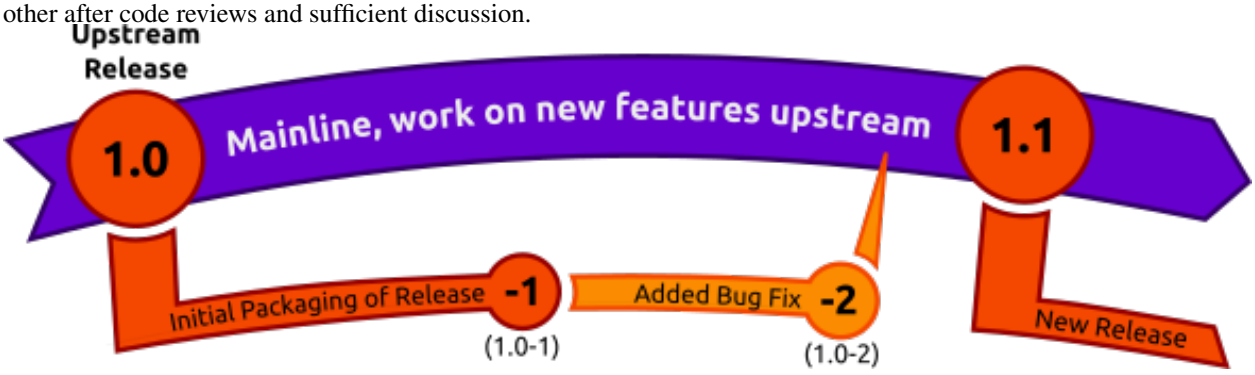
Uma outra importante ferramenta de comunicação são os relatórios de erros. Sempre que um defeito é encontrado em um pacote ou parte da infraestrutura, um relatório de erro deve ser registrado no Launchpad. Todas as informações são coletadas nesse relatório, a sua importância, o status e atualização de para quem foi atribuído, quando necessário. Isto faz com que seja uma ferramenta efetiva para ficar em cima dos erros em um pacote ou projeto e organizar a carga de trabalho.

Most of the software available through Ubuntu is not written by Ubuntu developers themselves. Most of it is written by developers of other Open Source projects and then integrated into Ubuntu. These projects are called “Upstreams”, because their source code flows into Ubuntu, where we “just” integrate it. The relationship to Upstreams is critically important to Ubuntu. It is not just code that Ubuntu gets from Upstreams, but it is also that Upstreams get users, bug reports and patches from Ubuntu (and other distributions).

The most important Upstream for Ubuntu is Debian. Debian is the distribution that Ubuntu is based on and many of the design decisions regarding the packaging infrastructure are made there. Traditionally, Debian has always had dedicated maintainers for every single package or dedicated maintenance teams. In Ubuntu there are teams that have an interest in a subset of packages too, and naturally every developer has a special area of expertise, but participation (and upload rights) generally is open to everyone who demonstrates ability and willingness.

Ser um novo colaborador do Ubuntu não é tão intimidador como parece e pode ser uma experiência bastante compensadora. Não é apenas aprender algo novo e excitante, mas também sobre compartilhar a solução e resolver um problema para milhões de usuários.

Open Source Development happens in a distributed world with different goals and different areas of focus. For example there might be the case that a particular Upstream is interested in working on a new big feature while Ubuntu, because of the tight release schedule, is interested in shipping a solid version with just an additional bug fix. That is why we make use of “Distributed Development”, where code is being worked on in various branches that are merged with each other after code reviews and sufficient discussion.



No exemplo mencionado acima faria sentido incluir a versão existente do projeto no Ubuntu, adicionar uma correção de erro, colocá-lo no upstream para o seu próximo lançamento (se for possível) e inclui-lo na próxima versão do Ubuntu. Seria a melhor solução, uma situação onde todos saem ganhando.

To fix a bug in Ubuntu, you would first get the source code for the package, then work on the fix, document it so it is easy to understand for other developers and users, then build the package to test it. After you have tested it, you can easily propose the change to be included in the current Ubuntu development release. A developer with upload rights will review it for you and then get it integrated into Ubuntu.

Quando estiver tentando encontrar uma solução, normalmente é uma boa ideia verificar como o uspstream se o problema (ou uma possível solução) já é conhecida e, se não, fazer de tudo para que a solução seja um esforço coordenado.

Passos adicionais podem envolver fazer o backport da alteração para uma versão mais velha ainda suportada ou repassá-la ao upstream.

Os requisitos mais importantes para o sucesso no desenvolvimento do Ubuntu são: ter a habilidade de "fazer as coisas voltarem a funcionar", não ter medo de ler documentação e fazer perguntas, jogar em equipe e gostar de trabalhar como detetive.

Good places to ask your questions are `ubuntu-motu@lists.ubuntu.com` and `#ubuntu-motu` on `freenode`.. You will easily find a lot of new friends and people with the same passion that you have: making the world a better place by making better Open Source software.

1.2 Guia de configuração

Há várias coisas que você precisa saber para começar a desenvolver para o Ubuntu. Esta artigo foi desenvolvido para deixar o seu computador preparado para que você possa começar a trabalhar com pacotes e enviá-los para a plataforma que hospeda o Ubuntu, o Launchpad. Eis o que vamos cobrir:

- Instalando pacotes de programas relacionados. Isto inclui:
  - Utilitários de empacotamento específicos do Ubuntu
  - Programas de criptografia para que seu trabalho possa ser verificado se foi feito por você
  - Programas adicionais de criptografia para que você possa transferir seus arquivos com segurança
- Criando e configurando sua conta no Launchpad
- Configurando o seu ambiente de desenvolvimento para ajudá-lo a fazer compilações de pacotes locais, interagir com outros desenvolvedores, e propor as alterações no Launchpad.

Nota: É recomendado fazer o trabalho de empacotamento utilizando a versão do Ubuntu que atualmente estiver em desenvolvimento. Isto irá permitir que você teste as alterações no mesmo ambiente em que elas serão aplicadas e utilizadas.

Don't want to install the latest development version of Ubuntu? Spin up an LXD container.

1.2.1 Instalar os programas de empacotamento básico

There are a number of tools that will make your life as an Ubuntu developer much easier. You will encounter these tools later in this guide. To install most of the tools you will need run this command:

```
$ sudo apt install gnupg pbuilder ubuntu-dev-tools apt-file
```

Este comando instalará os seguintes programas:

- `gnupg` – GNU Privacy Guard contains tools you will need to create a cryptographic key with which you will sign files you want to upload to Launchpad.

- `pbuilder` – uma ferramenta para fazer a reprodução da construção dos pacotes em um ambiente limpo e isolado.
- "ubuntu-dev-tools" (e "devscripts", uma dependência direta) – uma coleção de ferramentas para tornar mais fácil muitas tarefas de empacotamento.
- `apt-file` fornece um caminho fácil para pesquisar um pacote binário que contém o determinado arquivo.

Criar sua chave GPG

GPG stands for GNU Privacy Guard and it implements the OpenPGP standard which allows you to sign and encrypt messages and files. This is useful for a number of purposes. In our case it is important that you can sign files with your key so they can be identified as something that you worked on. If you upload a source package to Launchpad, it will only accept the package if it can absolutely determine who uploaded the package.

Para gerar uma nova chave GPG, execute:

```
$ gpg --gen-key
```

GPG will first ask you which kind of key you want to generate. Choosing the default (RSA and DSA) is fine. Next it will ask you about the keysize. The default (currently 2048) is fine, but 4096 is more secure. Afterwards, it will ask you if you want it to expire the key at some stage. It is safe to say "0", which means the key will never expire. The last questions will be about your name and email address. Just pick the ones you are going to use for Ubuntu development here, you can add additional email addresses later on. Adding a comment is not necessary. Then you will have to set a passphrase, choose a safe one (a passphrase is just a password which is allowed to include spaces).

Agora o GPG irá criar uma chave para você, o que pode levar uma pouco de tempo. Ele precisa de bytes aleatórios, então será bom se você fornecer ao sistema algum trabalho para fazer. Mova o cursor pela tela, digite alguns parágrafos de texto aleatório, carregue uma página da Internet.

uma vez feito isso, você receberá uma mensagem semelhante a esta:

```
pub   4096R/43CDE61D 2010-12-06
      Key fingerprint = 5C28 0144 FB08 91C0 2CF3  37AC 6F0B F90F 43CD E61D
uid                  Daniel Holbach <dh@mailempfang.de>
sub   4096R/51FBE68C 2010-12-06
```

Neste caso, `43CDE61D` é o *ID da chave*.

A seguir, você deve enviar a parte pública de sua chave para um servidor de chaves para que o mundo possa identificar mensagens e arquivos como sendo seus. Para fazer isso, digite:

```
$ gpg --send-keys --keyserver keyserver.ubuntu.com <KEY ID>
```

This will send your key to the Ubuntu keyserver, but a network of keyservers will automatically sync the key between themselves. Once this syncing is complete, your signed public key will be ready to verify your contributions around the world.

Criar sua chave SSH

SSH stands for *Secure Shell*, and it is a protocol that allows you to exchange data in a secure way over a network. It is common to use SSH to access and open a shell on another computer, and to use it to securely transfer files. For our purposes, we will mainly be using SSH to securely upload source packages to Launchpad.

Para gerar uma chave SSH, digite:

```
$ ssh-keygen -t rsa
```


O nome de arquivo padrão geralmente faz sentido, então você pode deixar como está. Por motivo de segurança, é altamente recomendado que você utilize uma frase secreta.

Configurando o pbuilder

`pbuilder` permite você construir pacotes localmente em sua máquina. Ele serve para vários propósitos:

- A construção será feita em um sistema mínimo e limpo. Isto ajuda a assegurar que suas construções sejam feitas com sucesso de uma maneira reprodutível, sem modificar seu sistema local
- Não há necessidade de instalar todas as *dependências de construção* localmente
- Você pode configurar múltiplas instâncias de várias versões do Ubuntu e do Debian

Configurar o `pbuilder` é muito fácil, execute:

```
$ pbuilder-dist <release> create
```

where <release> is for example *xenial*, *zesty*, *artful* or in the case of Debian maybe *sid* or *buster*. This will take a while as it will download all the necessary packages for a "minimal installation". These will be cached though.

1.2.2 Prepare-se para trabalhar com o Launchpad

Com uma configuração local básica estabelecida, seu próximo passo será configurar o seu sistema para trabalhar com o Launchpad. Esta seção irá se concentrar nos seguintes tópicos:

- O que o Launchpad é e criando uma conta nele
- Enviando suas chaves GPG e SSH para o Launchpad
- Configure your shell to recognize you (for putting your name in changelogs)

Sobre o Launchpad

O Launchpad é uma peça central da infraestrutura que usamos no Ubuntu. Ela não só armazena nossos pacotes e nossos códigos, mas também coisas como traduções, relatórios de erros, e informações sobre as pessoas que trabalham no Ubuntu e os seus grupos de membros. Você também irá usar o Launchpad para publicar propostas de correções, e conseguir outros desenvolvedores para revisão e orientação.

Você irá precisar se registrar no Launchpad e fornecer algumas informações mínimas. Isto irá permitir que você baixe e envie códigos, envie relatórios de erros e mais.

Além de hospedar o Ubuntu, o Launchpad pode hospedar qualquer projeto de Software Livre. Para mais informações, veja a wiki de ajuda do Launchpad.

Obter uma conta do Launchpad

If you don't already have a Launchpad account, you can easily create one. If you have a Launchpad account but cannot remember your Launchpad id, you can find this out by going to https://launchpad.net/~ and looking for the part after the ~ in the URL.

O processo de registro do Launchpad irá pedir que você escolha um nome de exibição. O uso do nome real é encorajado aqui, de modo que seus colegas desenvolvedores do Ubuntu sejam capazes de lhe conhecer melhor.

Quando você registrar uma nova conta, o Launchpad enviará um email com um link que precisa ser aberto no seu navegador para verificar o seu endereço de email. Se você não recebê-lo, verifique na sua pasta de spam.

The new account help page on Launchpad has more information about the process and additional settings you can change.

Enviar sua chave GPG para o Launchpad

First, you will need to get your fingerprint and key ID.

Para pesquisar sobre sua impressão digital GPG, execute:

```
$ gpg --fingerprint email@address.com
```

e será exibido alguma coisa como:

```
pub   4096R/43CDE61D 2010-12-06
      Key fingerprint = 5C28 0144 FB08 91C0 2CF3  37AC 6F0B F90F 43CD E61D
uid                  Daniel Holbach <dh@mailempfang.de>
sub   4096R/51FBE68C 2010-12-06
```

Then run this command to submit your key to Ubuntu keyserver:

```
$ gpg --keyserver keyserver.ubuntu.com --send-keys 43CDE61D
```

where `43CDE61D` should be replaced by your key ID (which is in the first line of output of the previous command). Now you can import your key to Launchpad.

Vá para https://launchpad.net/~/+editpgpkeys e copie a "Key fingerprint" na caixa de texto. No caso acima seria "5C28 0144 FB08 91C0 2CF3 37AC 6F0B F90F 43CD E61D". Agora clique em "Import Key".

O Launchpad irá usar a impressão digital para verificar a sua chave no servidor de chaves do Ubuntu e, se tudo ocorrer bem, enviar para você um e-mail criptografado pedindo para confirmar a importação da chave. Verifique a sua conta de e-mail e leia a mensagem que o Launchpad lhe enviou. "Se o seu cliente de e-mail tiver suporte a criptografia OpenPGP, ele irá pedir a senha que você escolheu para a chave quando o GPG a gerou. Digite a senha, e então clique no link para confirmar que a chave é sua".

Launchpad encrypts the email, using your public key, so that it can be sure that the key is yours. If you are using Thunderbird, the default Ubuntu email client, you can install the Enigmail plugin to easily decrypt the message. If your email software does not support OpenPGP encryption, copy the encrypted email's contents, type `gpg` in your terminal, then paste the email contents into your terminal window.

De volta ao site do Launchpad, clique no botão Confirm e o Launchpad irá completar a importação da sua chave OpenPGP.

Encontre mais informações em https://help.launchpad.net/YourAccount/ImportingYourPGPKey

Enviar sua chave SSH para o Launchpad

Open https://launchpad.net/~/+editsshkeys in a web browser, also open `~/.ssh/id_rsa.pub` in a text editor. This is the public part of your SSH key, so it is safe to share it with Launchpad. Copy the contents of the file and paste them into the text box on the web page that says "Add an SSH key". Now click "Import Public Key".

For more information on this process, visit the creating an SSH keypair page on Launchpad.

Configurar seu shell

The Debian/Ubuntu packaging tools need to learn about you as well in order to properly credit you in the changelog. Simply open your *~/.bashrc* in a text editor and add something like this to the bottom of it:


```
export DEBFULLNAME="Bob Dobbs"
export DEBEMAIL="subgenius@example.com"
```

Agora salve o arquivo e reinicie o seu terminal ou execute:

```
$ source ~/.bashrc
```

(se você não usa o shell padrão, que o é "bash", favor editar devidamente o arquivo de configuração para o seu shell.)

1.3 Corrigindo um erro no Ubuntu

1.3.1 Introdução

Se você seguiu as instruções do *prepare-se para o Desenvolvimento do Ubuntu*, você deve estar pronto para começar.

Como você pode observar na imagem acima, não há surpresas no processo de correção de erros no Ubuntu: você encontra um problema, obtém o código, trabalha para corrigi-lo, testa, envia as alterações para o Launchpad e pede para que sejam revisadas e incorporadas. Neste guia nós passaremos por todas as etapas necessárias, uma por uma.

1.3.2 Encontrando o problema

Há várias maneiras diferentes de encontrar coisas nas quais trabalhar. Pode ser um relatório de erro que você mesmo encontrou (o que lhe dá uma boa oportunidade para testar a correção), ou um problema que você encontrou em outro lugar, talvez num relatório de erro.

Take a look at the bitesize bugs in Launchpad, and that might give you an idea of something to work on. It might also interest you to look at the bugs triaged by the Ubuntu One Hundred Papercuts team.

1.3.3 Descobrindo o que corrigir

Se você não sabe qual o pacote fonte que contém o código que tem o problema, mas você sabe o caminho para o programa afetado no seu sistema, você pode descobrir o pacote fonte no qual você precisa trabalhar.

Let's say you've found a bug in Bumprace, a racing game. The Bumprace application can be started by running `/usr/bin/bumprace` on the command line. To find the binary package containing this application, use this command:

```
$ apt-file find /usr/bin/bumprace
```

Isso deverá imprimir:

```
bumprace: /usr/bin/bumprace
```


Note that the part preceding the colon is the binary package name. It's often the case that the source package and binary package will have different names. This is most common when a single source package is used to build multiple different binary packages. To find the source package for a particular binary package, type:

```
$ apt-cache showsrc bumprace | grep ^Package:
Package: bumprace
$ apt-cache showsrc tomboy | grep ^Package:
Package: tomboy
```

`apt-cache` é parte da instalação padrão do Ubuntu.

1.3.4 Confirmando o problema

Once you have figured out which package the problem is in, it's time to confirm that the problem exists.

Digamos que o pacote "bumprace" não tenha uma página na Internet na sua descrição de pacote. Como primeiro passo, você iria verificar se o problema já esta solucionado. Isto é fácil de verificar, dê uma olhado na Central de Programas ou execute:

```
apt-cache show bumprace
```

A saída deve ser similar a isto:

```
Package: bumprace
Priority: optional
Section: universe/games
Installed-Size: 136
Maintainer: Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>
XNBC-Original-Maintainer: Christian T. Steigies <cts@debian.org>
Architecture: amd64
Version: 1.5.4-1
Depends: bumprace-data, libc6 (>= 2.4), libsdl-image1.2 (>= 1.2.10),
         libsdl-mixer1.2, libsdl1.2debian (>= 1.2.10-1)
Filename: pool/universe/b/bumprace/bumprace_1.5.4-1_amd64.deb
Size: 38122
MD5sum: 48c943863b4207930d4a2228cedc4a5b
SHA1: 73bad0892be471bbc471c7a99d0b72f0d0a4babc
SHA256: 64ef9a45b75651f57dc76aff5b05dd7069db0c942b479c8ab09494e762ae69fc
Description-en: 1 or 2 players race through a multi-level maze
 In BumpRacer, 1 player or 2 players (team or competitive) choose among 4
 vehicles and race through a multi-level maze. The players must acquire
 bonuses and avoid traps and enemy fire in a race against the clock.
 For more info, see the homepage at http://www.linux-games.com/bumprace/
Description-md5: 3225199d614fba85ba2bc66d5578ff15
Bugs: https://bugs.launchpad.net/ubuntu/+filebug
Origin: Ubuntu
```

Um contra-exemplo seria o "gedit", que tem uma página na Internet definida:

```
$ apt-cache show gedit | grep ^Homepage
Homepage: http://www.gnome.org/projects/gedit/
```

Algumas vezes você ira descobrir que um problema em particular que você está analisando já está corrigido. Para evitar desperdiçar esforço e duplicar trabalho, faz sentido fazer um trabalho de detetive antes.

1.3.5 Pesquisar situação da falha

Primeiro temos que verificar um relatório de erro para o problema já existe no Ubuntu. Talvez alguém já esteja trabalhando em uma correção, ou que possamos contribuir para solução de alguma maneira. Para o Ubuntu, nós demos uma rápida olhada em https://bugs.launchpad.net/ubuntu/+source/bumprace e não há um relatório de erro ara o problema lá.

Nota: Para o Ubuntu a URL `https://bugs.launchpad.net/ubuntu/+source/<pacote>` deve sempre levar até a página de falhas do pacote fonte em questão.

Para o Debian, que é a maior fonte para os pacotes do Ubuntu, nós demos uma olhada em http://bugs.debian.org/src:bumprace e não achamos um relatório para o nosso problema também.

Nota: Para o Debian a URL `http://bugs.debian.org/src:<pacote>` deve sempre levar até a página de falhas do pacote fonte em questão.

The problem we are working on is special as it only concerns the packaging-related bits of `bumprace`. If it was a problem in the source code it would be helpful to also check the Upstream bug tracker. This is unfortunately often different for every package you have a look at, but if you search the web for it, you should in most cases find it pretty easily.

1.3.6 Oferecendo ajuda

Se você encontrou um relatório de erro aberto, se ele não estiver atribuído a alguém e você puder corrigi-lo, você deverá comentar no relatório a sua solução. Certifique-se de incluir toda a informação que puder: em que circunstâncias o erro acontece? Como você corrigiu o problema? Você testou a sua solução?

Se nenhum relatório de erro foi arquivado, você pode arquivá-lo. O que você deve ter em mente é: o problema é tão pequeno que pedir para alguém para repará-lo seria suficiente? Você conseguiu corrigir apenas parcialmente o problema e você quer ao menos compartilhar a sua parte na correção?

Isso é ótimo se você pode oferecer ajuda e certamente será apreciado.

1.3.7 Obtendo o código

Once you know the source package to work on, you will want to get a copy of the code on your system, so that you can debug it. The ubuntu-dev-tools package has a tool called `pull-lp-source` that a developer can use to grab the source code for any package. For example, to grab the source code for the tomboy package in `xenial`, you can type this:

```
$ pull-lp-source bumprace xenial
```

If you do not specify a release such as `xenial`, it will automatically get the package from the development version.

Once you've got a local clone of the source package, you can investigate the bug, create a fix, generate a debdiff, and attach your debdiff to a bug report for other developers to review. We'll describe specifics in the next sections.

1.3.8 trabalhar em uma correção

Há livros inteiros escritos sobre encontrar erros, corrigi-los, testá-los, etc. Se você for um programador novato, tente corrigir erros fáceis primeiro, como erros de digitação óbvios. Tente alterar o mínimo possível, e documente sua alteração e suposições com clareza.

Antes de trabalhar na correção você mesmo, tenha certeza de investigar se ninguém já corrigiu ou está trabalhando na correção para a falha. Boas fontes para verificar são:

- Rastreamento de erros upstream (e Debian) de erros abertos e fechados,
- Histórico de revisão do upstream (ou lançamento recente) poderia ter corrigido o problema,
- bugs ou envio de pacotes do Debian ou outras distribuições.

You may want to create a patch which includes the fix. The command `edit-patch` is a simple way to add a patch to a package. Run:

```
$ edit-patch 99-new-patch
```

Isso copiará o empacotamento para um diretório temporário. Agora você pode editar arquivos com um editor de textos ou aplicar patches do upstream, por exemplo:

```
$ patch -p1 < ../bugfix.patch
```

Após editar o tipo de arquivo `exit` ou pressione `control-d` para sair do shell temporário. O novo patch será adicionado em `debian/patches`.

You must then add a header to your patch containing meta information so that other developers can know the purpose of the patch and where it came from. To get the template header that you can edit to reflect what the patch does, type this:

```
$ quilt header --dep3 -e
```

This will open the template in a text editor. Follow the template and make sure to be thorough so you get all the details necessary to describe the patch.

In this specific case, if you just want to edit `debian/control`, you do not need a patch. Put `Homepage: http://www.linux-games.com/bumprace/` at the end of the first section and the bug should be fixed.

Documentando a correção

É muito importante documentar a sua alteração suficientemente de maneira que os desenvolvedores que olharem o código no futuro não terão que adivinhar qual foi o seu raciocínio e quais foram a suposições. Todo pacote fonte do Debian e do Ubuntu inclui o "debian/changelog", onde alterações em cada pacote enviado são rastreadas.

A maneira mais fácil de atualizar isto é executar:

```
$ dch -i
```

Isto irá adicionar um modelo entrada de registro de alteração para você e iniciar um editor onde você possa preencher os espaços em branco. Um exemplo disso poderia ser:

```
specialpackage (1.2-3ubuntu4) trusty; urgency=low

  * debian/control: updated description to include frobnicator (LP: #123456)

 -- Emma Adams <emma.adams@isp.com>  Sat, 17 Jul 2010 02:53:39 +0200
```

`dch` should fill out the first and last line of such a changelog entry for you already. Line 1 consists of the source package name, the version number, which Ubuntu release it is uploaded to, the urgency (which almost always is 'low'). The last line always contains the name, email address and timestamp (in **RFC 5322** format) of the change.

Com isso fora do caminho, vamos nos concentrar na entrada do registro de alteração propriamente dita: é muito importante documentar:

1. Where the change was done.

2. What was changed.
3. Where the discussion of the change happened.

In our (very sparse) example the last point is covered by `(LP: #123456)` which refers to Launchpad bug 123456. Bug reports or mailing list threads or specifications are usually good information to provide as a rationale for a change. As a bonus, if you use the `LP: #<number>` notation for Launchpad bugs, the bug will be automatically closed when the package is uploaded to Ubuntu.

In order to get it sponsored in the next section, you need to file a bug report in Launchpad (if there isn't one already, if there is, use that) and explain why your fix should be included in Ubuntu. For example, for tomboy, you would file a bug here (edit the URL to reflect the package you have a fix for). Once a bug is filed explaining your changes, put that bug number in the changelog.

1.3.9 Testando a correção

Para construir um pacote de teste com suas alterações, e execute estes comandos:

```
$ debuild -S -d -us -uc
$ pbuilder-dist <release> build ../<package>_<version>.dsc
```

This will create a source package from the branch contents (`-us -uc` will just omit the step to sign the source package and `-d` will skip the step where it checks for build dependencies, pbuilder will take care of that) and `pbuilder-dist` will build the package from source for whatever `release` you choose.

Nota: If `debuild` errors out with "Version number suggests Ubuntu changes, but Maintainer: does not have Ubuntu address" then run the `update-maintainer` command (from ubuntu-dev-tools) and it will automatically fix this for you. This happens because in Ubuntu, all Ubuntu Developers are responsible for all Ubuntu packages, while in Debian, packages have maintainers.

In this case with bumprace, run this to view the package information:

```
$ dpkg -I ~/pbuilder/*_result/bumprace_*.deb
```

As expected, there should now be a `Homepage:` field.

Nota: In a lot of cases you will have to actually install the package to make sure it works as expected. Our case is a lot easier. If the build succeeded, you will find the binary packages in `~/pbuilder/<release>_result`. Install them via `sudo dpkg -i <package>.deb` or by double-clicking on them in your file manager.

1.3.10 Submitting the fix and getting it included

With the changelog entry written and saved, run `debuild` one more time:

```
$ debuild -S -d
```

and this time it will be signed and you are now ready to get your diff to submit to get sponsored.

In a lot of cases, Debian would probably like to have the patch as well (doing this is best practice to make sure a wider audience gets the fix). So, you should submit the patch to Debian, and you can do that by simply running this:

```
$ submittodebian
```

Isto lhe levará por uma série de passos para assegurar que o erro vá parar no lugar correto. Assegure-se de revisar o diff novamente para ter certeza de que não inclui alterações aleatórias que você tenha feito antes.

Comunicação é importante, então quando você adicionar alguma descrição para solicitar a inclusão, seja amigável, explique ela bem.

Se tudo deu certo você deverá receber um e-mail do sistema de rastreamento de erros do Debian com mais informações. Isto pode levar alguns minutos.

It might be beneficial to just get it included in Debian and have it flow down to Ubuntu, in which case you would not follow the below process. But, sometimes in the case of security updates and updates for stable releases, the fix is already in Debian (or ignored for some reason) and you would follow the below process. If you are doing such updates, please read our *Security and stable release updates* article. Other cases where it is acceptable to wait to submit patches to Debian are Ubuntu-only packages not building correctly, or Ubuntu-specific problems in general.

But if you're going to submit your fix to Ubuntu, now it's time to generate a "debdiff", which shows the difference between two Debian packages. The name of the command used to generate one is also `debdiff`. It is part of the `devscripts` package. See `man debdiff` for all the details. To compare two source packages, pass the two dsc files as arguments:

```
$ debdiff <package_name>_1.0-1.dsc <package_name>_1.0-1ubuntu1.dsc
```

In this case, `debdiff` the dsc you downloaded with `pull-lp-source` and the new dsc file you generated. This will generate a patch that your sponsor can then apply locally (by using `patch -p1 < /path/to/debdiff`). In this case, pipe the output of the debdiff command to a file that you can then attach to the bug report:

```
$ debdiff <package_name>_1.0-1.dsc <package_name>_1.0-1ubuntu1.dsc > 1-1.0-1ubuntu1.debdiff
```

The format shown in `1-1.0-1ubuntu1.debdiff` shows:

1. `1-` tells the sponsor that this is the first revision of your patch. Nobody is perfect, and sometimes follow-up patches need to be provided. This makes sure that if your patch needs work, that you can keep a consistent naming scheme.
2. `1.0-1ubuntu1` shows the new version being used. This makes it easy to see what the new version is.
3. `.debdiff` is an extension that makes it clear that it is a debdiff.

While this format is optional, it works well and you can use this.

Next, go to the bug report, make sure you are logged into Launchpad, and click "Add attachment or patch" under where you would add a new comment. Attach the debdiff, and leave a comment telling your sponsor how this patch can be applied and the testing you have done. An example comment can be:

```
This is a debdiff for Artful applicable to 1.0-1. I built this in pbuilder
and it builds successfully, and I installed it, the patch works as intended.
```

Make sure you mark it as a patch (the Ubuntu Sponsors team will automatically be subscribed) and that you are subscribed to the bug report. You will then receive a review anywhere between several housr from submitting the patch to several weeks. If it takes longer than that, please join `#ubuntu-motu` on `freenode` and mention it there. Stick around until you get an answer from someone, and they can guide you as to what to do next.

Once you have received a review, your patch was either uploaded, your patch needs work, or is rejected for some other reason (possibly the fix is not fit for Ubuntu or should go to Debian instead). If your patch needs work, follow the same steps and submit a follow-up patch on the bug report, otherwise submit to Debian as shown above.

Remember: good places to ask your questions are `ubuntu-motu@lists.ubuntu.com` and `#ubuntu-motu` on freenode. You will easily find a lot of new friends and people with the same passion that you have: making the world a better place by making better Open Source software.

1.3.11 Considerações adicionais

Se você encontrar um pacote e achar que há algumas coisas pequenas que você pode corrigir ao mesmo tempo, faça isso. Isto irá acelerar a revisão e a inclusão.

Se há várias coisas grandes que você queira corrigir, é prudente que você envie patches individuais ou propostas de mesclagem. Se houver erros individuais já arquivados para esses problemas, isto facilita ainda mais.

1.4 Empacotando um novo software

Mesmo que haja milhares de pacotes no arquivo do Ubuntu, ainda há os que ninguém pegou ainda. Se houver um programa que você acha que deva ganhar uma exposição maior, talvez você queira colocar a mão na massa e criar um pacote para o Ubuntu ou um PPA. Este guia irá mostrar passo-a-passo o processo de empacotamento e um novo programa.

Você deverá ler o artigo *Preparando* para preparar o seu ambiente de desenvolvimento.

1.4.1 Verificando o programa

O primeiro estágio no empacotamento é obter o tar da versão a partir do upstream (nós chamamos os autores dos aplicativos de "upstream") e verificar se ele compila e executa.

Este guia irá mostrar como empacotar um aplicativo simples chamado GNU Hello que foi postado em GNU.org.

Download GNU Hello:

```
$ wget -O hello-2.10.tar.gz "http://ftp.gnu.org/gnu/hello/hello-2.10.tar.gz"
```

Now uncompress it:

```
$ tar xf hello-2.10.tar.gz
$ cd hello-2.10
```

Este aplicativo utiliza um sistema de compilação de autoconfiguração, então nós devemos executar "./configure" para preparar para a compilação.

This will check for the required build dependencies. As `hello` is a simple example, `build-essential` should provide everything we need. For more complex programs, the command will fail if you do not have the needed libraries and development files. Install the needed packages and repeat until the command runs successfully.:

```
$ ./configure
```

Agora você pode compilar o fonte:

```
$ make
```

Se a compilação concluir com sucesso você poderá instalar e executar o programa:

```
$ sudo make install
$ hello
```

1.4.2 Iniciando um pacote

`bzr-builddeb` includes a plugin to create a new package from a template. The plugin is a wrapper around the `dh_make` command. Run the command providing the package name, version number, and path to the upstream tarball:


```
$ sudo apt-get install dh-make bzr-builddeb
$ cd ..
$ bzr dh-make hello 2.10 hello-2.10.tar.gz
```

Quando for perguntado o tipo de pacote, digite "s" para um único binário. Isto irá importar o código para um ramo e adicionar o diretório de empacotamento "debian/". Dê uma olhada no conteúdo. A maioria dos arquivos adicionados são necessários somente para pacotes especializados (como módulos Emacs), então você pode começar removendo os arquivos de exemplo opcionais:

```
$ cd hello/debian
$ rm *ex *EX
```

Agora você deve customizar cada um dos arquivos.

In `debian/changelog` change the version number to an Ubuntu version: `2.10-0ubuntu1` (upstream version 2.10, Debian version 0, Ubuntu version 1). Also change `unstable` to the current development Ubuntu release such as `trusty`.

Much of the package building work is done by a series of scripts called `debhelper`. The exact behaviour of `debhelper` changes with new major versions, the compat file instructs `debhelper` which version to act as. You will generally want to set this to the most recent version which is `9`.

`control` contains all the metadata of the package. The first paragraph describes the source package. The second and following paragraphs describe the binary packages to be built. We will need to add the packages needed to compile the application to `Build-Depends:`. For `hello`, make sure that it includes at least:

```
Build-Depends: debhelper (>= 9)
```

Você também precisará preencher a descrição do programa no campo "Description:".

"copyright" deve ser preenchido de acordo com a licença do código upstream. De acordo com o arquivo hello/COPYING, ela é GNU GPL 3 ou mais recente.

"docs" contém todos os arquivos de documentação do upstream que você acha que devam ser incluídos no pacote final.

"README.source" e "README.Debian" são necessários somente se o seu pacote tiver quaisquer características fora do padrão. Nós não temos, você pode excluí-los.

"source/format" pode ser deixado como está, ele descreve o formato da versão do pacote fonte e deve ser "3.0 (quilt)".

O "rules" é o arquivo mais complexo. É um makefile que compila o código e o transforma em um pacote binário. Felizmente, a maior parte do trabalho hoje em dia é feita automaticamente pelo "debhelper 7" para que o alvo makefile universal "%" apenas execute o script "dh", que irá executar tudo que for necessário.

Todos estes arquivos são explicados mais detalhadamente no artigo *resumo do diretório debian*.

Finalmente submeta o código para seu ramo de empacotamento:

```
$ bzr add debian/source/format
$ bzr commit -m "Initial commit of Debian packaging."
```

1.4.3 Construindo o pacote

Agora precisamos verificar se nosso empacotamento compila o pacote com sucesso e constrói o pacote binário .deb:

```
$ bzr builddeb -- -us -uc
$ cd ../../
```


"bzr builddeb" é um comando para construir o pacote no local atual. As opções "-us -uc" dizem que não é preciso que o GPG assine o pacote. O resultado será colocado em "..".

Você pode visualizar o conteúdo do pacote com:

```
$ lesspipe hello_2.10-0ubuntu1_amd64.deb
```

Install the package and check it works (later you will be able to uninstall it using `sudo apt-get remove hello` if you want):

```
$ sudo dpkg --install hello_2.10-0ubuntu1_amd64.deb
```

You can also install all packages at once using:

```
$ sudo debi
```

1.4.4 Próximos passos

Even if it builds the .deb binary package, your packaging may have bugs. Many errors can be automatically detected by our tool `lintian` which can be run on the source .dsc metadata file, .deb binary packages or .changes file:

```
$ lintian hello_2.10-0ubuntu1.dsc
$ lintian hello_2.10-0ubuntu1_amd64.deb
```

To see verbose description of the problems use `--info` lintian flag or `lintian-info` command.

For Python packages, there is also a `lintian4python` tool that provides some additional lintian checks.

Depois de efetuar uma correção no empacotamento, você pode reconstruí-lo usando "-nc" (no clean) não precisando construir tudo novamente:

```
$ bzr builddeb -- -nc -us -uc
```

Having checked that the package builds locally you should ensure it builds on a clean system using `pbuilder`. Since we are going to upload to a PPA (Personal Package Archive) shortly, this upload will need to be *signed* to allow Launchpad to verify that the upload comes from you (you can tell the upload will be signed because the `-us` and `-uc` flags are not passed to `bzr builddeb` like they were before). For signing to work you need to have set up GPG. If you haven't set up `pbuilder-dist` or GPG yet, *do so now*:

```
$ bzr builddeb -S
$ cd ../build-area
$ pbuilder-dist trusty build hello_2.10-0ubuntu1.dsc
```

Quando você estiver feliz com seu pacote você irá querer que outros revisem-o. Você pode enviar para o ramo no Launchpad para revisão:

```
$ bzr push lp:~<lp-username>/+junk/hello-package
```

Enviá-lo ao PPA irá assegurar que ele pode ser construído e irá fornecer uma maneira fácil para você e os outros testarem os pacotes binários. Você deverá configurá um PPA no Launchpad e então enviar o pacote com "dput":

```
$ dput ppa:<lp-username>/<ppa-name> hello_2.10-0ubuntu1.changes
```

You can ask for reviews in `#ubuntu-motu` IRC channel, or on the MOTU mailing list. There might also be a more specific team you could ask such as the GNU team for more specific questions.

1.4.5 Submetendo para inclusão

Há várias maneira pelas quais um pacote pode entrar no Ubuntu. Na maioria dos casos, ir através do Debian primeiro pode ser a melhor maneira. Isto assegurará que o seu pacote alcançará o maior número de usuários, porque estará disponível não somente no Debian e no Ubuntu, mas também em todas as distribuições derivadas. Aqui estão alguns links úteis para enviar novos pacotes para o Debian:

- Debian Mentors FAQ - debian-mentors is for the mentoring of new and prospective Debian Developers. It is where you can find a sponsor to upload your package to the archive.
- Work-Needing and Prospective Packages - Information on how to file "Intent to Package" and "Request for Package" bugs as well as list of open ITPs and RFPs.
- Debian Developer's Reference, 5.1. New packages - The entire document is invaluable for both Ubuntu and Debian packagers. This section documents processes for submitting new packages.

In some cases, it might make sense to go directly into Ubuntu first. For instance, Debian might be in a freeze making it unlikely that your package will make it into Ubuntu in time for the next release. This process is documented on the "New Packages" section of the Ubuntu wiki.

1.4.6 Screenshots

Once you have uploaded a package to debian, you should add screenshots to allow propective users to see what the program is like. These should be uploaded to http://screenshots.debian.net/upload .

1.5 Atualizações de segurança de versão estável

1.5.1 Corrigindo uma falha de segurança no Ubuntu

Introdução

Fixing security bugs in Ubuntu is not really any different than *fixing a regular bug in Ubuntu*, and it is assumed that you are familiar with patching normal bugs. To demonstrate where things are different, we will be updating the dbus package in Ubuntu 12.04 LTS (Precise Pangolin) for a security update.

Obtendo o fonte

In this example, we already know we want to fix the dbus package in Ubuntu 12.04 LTS (Precise Pangolin). So first you need to determine the version of the package you want to download. We can use the `rmadison` to help with this:

```
$ rmadison dbus | grep precise
dbus | 1.4.18-1ubuntu1   | precise          | source, amd64, armel, armhf, i386, powerpc
dbus | 1.4.18-1ubuntu1.4 | precise-security | source, amd64, armel, armhf, i386, powerpc
dbus | 1.4.18-1ubuntu1.4 | precise-updates  | source, amd64, armel, armhf, i386, powerpc
```

Typically you will want to choose the highest version for the release you want to patch that is not in -proposed or -backports. Since we are updating Precise's dbus, you'll download 1.4.18-1ubuntu1.4 from precise-updates:

```
$ bzr branch ubuntu:precise-updates/dbus
```


Aplicando um patch no código fonte

Now that we have the source package, we need to patch it to fix the vulnerability. You may use whatever patch method that is appropriate for the package, but this example will use `edit-patch` (from the ubuntu-dev-tools package). `edit-patch` is the easiest way to patch packages and it is basically a wrapper around every other patch system you can imagine.

Para criar seu patch usando `edit-patch`:

```
$ cd dbus
$ edit-patch 99-fix-a-vulnerability
```

Isto irá aplicar os patches existentes e colocar o pacote em um diretório temporário. Agora edite os arquivos necessários para corrigir a vulnerabilidade. Frequentemente, o upstream terá fornecido um patch, então você pode aplicá-lo:

```
$ patch -p1 < /home/user/dbus-vulnerability.diff
```

Depois de fazer as mudanças necessárias, é só pressionar Ctrl-D ou digitar "exit" para sair temporariamente do shell.

Formatando o registro de alteração e os patches

After applying your patches you will want to update the changelog. The `dch` command is used to edit the `debian/changelog` file and `edit-patch` will launch `dch` automatically after un-applying all the patches. If you are not using `edit-patch`, you can launch `dch -i` manually. Unlike with regular patches, you should use the following format (note the distribution name uses precise-security since this is a security update for Precise) for security updates:

```
dbus (1.4.18-2ubuntu1.5) precise-security; urgency=low

  * SECURITY UPDATE: [DESCRIBE VULNERABILITY HERE]
    - debian/patches/99-fix-a-vulnerability.patch: [DESCRIBE CHANGES HERE]
    - [CVE IDENTIFIER]
    - [LINK TO UPSTREAM BUG OR SECURITY NOTICE]
    - LP: #[BUG NUMBER]
...
```

Atualize o seu patch para que use tags apropriadas. Seu patch deve ter no mínimo a orígem, descrição e tags bug-Ubuntu. Por exemplo, edite debian/patches/99-fix-a-vulnerability.patch para ter algo como:

```
## Description: [DESCRIBE VULNERABILITY HERE]
## Origin/Author: [COMMIT ID, URL OR EMAIL ADDRESS OF AUTHOR]
## Bug: [UPSTREAM BUG URL]
## Bug-Ubuntu: https://launchpad.net/bugs/[BUG NUMBER]
Index: dbus-1.4.18/dbus/dbus-marshal-validate.c
...
```

Várias vulnerabilidades podem ser corrigidas no mesmo envio de segurança; apenas certifique-se de usar patches diferentes para vulnerabilidades diferentes.

Testar e enviar o seu trabalho

Neste ponto, o processo é o mesmo que para *corrigir um erro comum no Ubuntu*. Especificamente, você deve:

1. Construir seu pacote e verificar se compilou sem erros e sem quaisquer avisos de compilador
2. Atualizar para a nova versão do pacote de uma versão anterior
3. Verifique se o novo pacotes corrige a vulnerabilidade e não introduz nenhuma regressão

4. Envie seu trabalho via proposta de mesclagem do Launchpad, registre um erro no Launchpad assegurando-se de marcar o erro como erro de segurança e inscreva-se no "ubuntu-security-sponsors".

Se a vulnerabilidade de segurança não é pública ainda então não adicione a proposta de mesclagem e certifique-se de que você marcou a falha como privada.

The filed bug should include a Test Case, i.e. a comment which clearly shows how to recreate the bug by running the old version then how to ensure the bug no longer exists in the new version.

The bug report should also confirm that the issue is fixed in Ubuntu versions newer than the one with the proposed fix (in the above example newer than Precise). If the issue is not fixed in newer Ubuntu versions you should prepare updates for those versions too.

1.5.2 Atualizações de versão estável

We also allow updates to releases where a package has a high impact bug such as a severe regression from a previous release or a bug which could cause data loss. Due to the potential for such updates to themselves introduce bugs we only allow this where the change can be easily understood and verified.

O processo para atualizações de versões estáveis é o mesmo que o processo para falhas de segurança, exceto que você deve registrar "ubuntu-sru" no relatório de erro.

The update will go into the `proposed` archive (for example `precise-proposed`) where it will need to be checked that it fixes the problem and does not introduce new problems. After a week without reported problems it can be moved to `updates`.

See the Stable Release Updates wiki page for more information.

1.6 Patches para pacotes

Sometimes, Ubuntu package maintainers have to change the upstream source code in order to make it work properly on Ubuntu. Examples include, patches to upstream that haven't yet made it into a released version, or changes to the upstream's build system needed only for building it on Ubuntu. We could change the upstream source code directly, but doing this makes it more difficult to remove the patches later when upstream has incorporated them, or extract the change to submit to the upstream project. Instead, we keep these changes as separate patches, in the form of diff files.

Há várias maneiras diferentes de lidar com patches em pacotes Debian, felizmente estamos padronizando um sistema, Quilt, que agora é utilizado pela maioria dos pacotes.

Let's look at an example package, `kamoso` in Trusty:

```
$ bzr branch ubuntu:trusty/kamoso
```

The patches are kept in `debian/patches`. This package has one patch `kubuntu_01_fix_qmax_on_armel.diff` to fix a compile failure on ARM. The patch has been given a name to describe what it does, a number to keep the patches in order (two patches can overlap if they change the same file) and in this case the Kubuntu team adds their own prefix to show the patch comes from them rather than from Debian.

A ordem dos patches para aplicar é mantida em `debian/patches/series`.

1.6.1 Patches com o Quilt

Antes de trabalhar com o Quilt você precisa informar a ele onde encontrar os patches. Adicione isso em seu "~/.bashrc":


```
export QUILT_PATCHES=debian/patches
```

E forneça a origem do arquivo para aplicar a nova exportação:

```
$ . ~/.bashrc
```

Por padrão, todos os patches são aplicados diretamente nos checkouts de UDD ou pacotes baixados. Você pode verificar isto com:

```
$ quilt applied
kubuntu_01_fix_qmax_on_armel.diff
```

Se você quisesse remover o patch poderia executar `pop`:

```
$ quilt pop
Removing patch kubuntu_01_fix_qmax_on_armel.diff
Restoring src/kamoso.cpp

No patches applied
```

E para aplicar um patch você usa `push`:

```
$ quilt push
Applying patch kubuntu_01_fix_qmax_on_armel.diff
patching file src/kamoso.cpp

Now at patch kubuntu_01_fix_qmax_on_armel.diff
```

1.6.2 Adicionando um novo patch

Para adicionar um novo patch você deve dizer ao Quilt para criar um novo patch, dizer a ele quais arquivos o patch irá alterar, editar os arquivos e então atualizar o patch:

```
$ quilt new kubuntu_02_program_description.diff
Patch kubuntu_02_program_description.diff is now on top
$ quilt add src/main.cpp
File src/main.cpp added to patch kubuntu_02_program_description.diff
$ sed -i "s,Webcam picture retriever,Webcam snapshot program,"
src/main.cpp
$ quilt refresh
Refreshed patch kubuntu_02_program_description.diff
```

O passo `quilt add` é importante, se você esquecê-lo os arquivos não vão terminar no patch.

A mudança agora estará em `debian/patches/kubuntu_02_program_description.diff` e o arquivo `series` terá um novo patch adicionado a ele. Você deve adicionar o novo arquivo para o empacotamento:

```
$ bzr add debian/patches/kubuntu_02_program_description.diff
$ bzr add .pc/*
$ dch -i "Add patch kubuntu_02_program_description.diff to improve the program description"
$ bzr commit
```

O Quilt mantém os seus metadados no diretório ".pc/", então atualmente você precisa adicioná-lo ao empacotamento também. Isto deverá ser aprimorado no futuro.

As a general rule you should be careful adding patches to programs unless they come from upstream, there is often a good reason why that change has not already been made. The above example changes a user interface string for example, so it would break all translations. If in doubt, do ask the upstream author before adding a patch.

1.6.3 Patch Headers

We recommend that you tag every patch with DEP-3 headers by putting them at the top of patch file. Here are some headers that you can use:

Description Description of what the patch does. It is formatted like `Description` field in `debian/control`: first line is short description, starting with lowercase letter, the next lines are long description, indented with a space.

Author Who wrote the patch (i.e. "Jane Doe <packager@example.com>").

Origin Where this patch comes from (i.e. "upstream"), when *Author* is not present.

Bug-Ubuntu A link to Launchpad bug, a short form is preferred (like *https://bugs.launchpad.net/bugs/XXXXXXX*). If there are also bugs in upstream or Debian bugtrackers, add *Bug* or *Bug-Debian* headers.

Forwarded Whether the patch was forwarded upstream. Either "yes", "no" or "not-needed".

Last-Update Date of the last revision (in form "YYYY-MM-DD").

1.6.4 Atualizando para novas versões do upstream

To upgrade to the new version, you can use `bzr merge-upstream` command:

```
$ bzr merge-upstream --version 2.0.2 https://launchpad.net/ubuntu/+archive/primary/+files/kamoso_2.0
```

When you run this command, all patches will be unapplied, because they can become out of date. They might need to be refreshed to match the new upstream source or they might need to be removed altogether. To check for problems, apply the patches one at a time:

```
$ quilt push
Applying patch kubuntu_01_fix_qmax_on_armel.diff
patching file src/kamoso.cpp
Hunk #1 FAILED at 398.
1 out of 1 hunk FAILED -- rejects in file src/kamoso.cpp
Patch kubuntu_01_fix_qmax_on_armel.diff can be reverse-applied
```

Se a aplicação puder ser revertida, isto significa que o patch já foi aplicado pelo upstream, então podemos excluir o patch:

```
$ quilt delete kubuntu_01_fix_qmax_on_armel
Removed patch kubuntu_01_fix_qmax_on_armel.diff
```

Então continue:

```
$ quilt push
Applied kubuntu_02_program_description.diff
```

É bom executar uma atualização do patch relativo ao fonte do upstream alterado:

```
$ quilt refresh
Refreshed patch kubuntu_02_program_description.diff
```

Então submeta como sempre:

```
$ bzr commit -m "new upstream version"
```


1.6.5 Fazendo um pacote usar o Quilt

Pacotes modernos usam o Quilt por padrão, que é construído dentro do formato do pacote. Verifique `debian/source/format` para assegurar que ele diz `3.0 (quilt)`.

Pacotes antigos usando o formato de fonte 1.0 terão que explicitar o uso do Quilt, normalmente incluindo um makefile dentro de `debian/rules`.

1.6.6 Configuring Quilt

You can use `~/.quiltrc` file to configure quilt. Here are some options that can be useful for using quilt with debian/packages:

```
# Set the patches directory
QUILT_PATCHES="debian/patches"
# Remove all useless formatting from the patches
QUILT_REFRESH_ARGS="-p ab --no-timestamps --no-index"
# The same for quilt diff command, and use colored output
QUILT_DIFF_ARGS="-p ab --no-timestamps --no-index --color=auto"
```

1.6.7 Outros sistemas de patch

Other patch systems used by packages include `dpatch` and `cdbs simple-patchsys`, these work similarly to Quilt by keeping patches in `debian/patches` but have different commands to apply, un-apply or create patches. You can find out which patch system is used by a package by using the `what-patch` command (from the `ubuntu-dev-tools` package). You can use `edit-patch`, shown in *previous chapters*, as a reliable way to work with all systems.

In even older packages changes will be included directly to sources and kept in the `diff.gz` source file. This makes it hard to upgrade to new upstream versions or differentiate between patches and is best avoided.

Não altere um sistema de patches de um pacote sem discutir isto com o mantenedor Debian ou time Ubuntu responsável. Se não houver sistema de patches, então sinta-se à vontade para adicionar o Quilt.

1.7 Fixing FTBFS packages

Before a package can be used in Ubuntu, it has to build from source. If it fails this, it will probably wait in -proposed and will not be available in the Ubuntu archives. You can find a complete list of packages that are failing to build from source at http://qa.ubuntuwire.org/ftbfs/. There are 5 main categories shown on the page:

- Package failed to build (F): Something actually went wrong with the build process.
- Cancelled build (X): The build has been cancelled for some reason. These should probably be avoided to start with.
- Package is waiting on another package (M): This package is waiting on another package to either build, get updated, or (if the package is in main) one of it's dependancies is in the wrong part of the archive.
- Failure in the chroot (C): Part of the chroot failed, this is most likely fixed by a rebuild. Ask a developer to rebuild the package and that should fix it.
- Failed to upload (U): The package could not upload. This is usually just a case of asking for a rebuild, but check the build log first.

1.7.1 First steps

The first thing you'll want to do is see if you can reproduce the FTBFS yourself. Get the code either by running `bzr branch lp:ubuntu/PACKAGE` and then getting the tarball or running `dget PACKAGE_DSC` on the .dsc file from the launchpad page. Once you have that, build it in a schroot.

You should be able to reproduce the FTBFS. If not, check if the build is downloading a missing dependency, which means you just need to make that a build-dependency in debian/control. Building the package locally can also help find if the issue is caused by a missing, unlisted, dependency (builds locally but fails on a schroot).

1.7.2 Checking Debian

Once you have reproduced the issue, it's time to try and find a solution. If the package is in Debian as well, you can check if the package builds there by going to http://packages.qa.debian.org/PACKAGE. If Debian has a newer version, you should merge it. If not, check the buildlogs and bugs linked on that page for any extra information on the ftbfs or patches. Debian also maintains a list of command FTBFSs and how to fix them which can be found at https://wiki.debian.org/qa.debian.org/FTBFS, you will want to check it for solutions too.

1.7.3 Other causes of a package to FTBFS

If a package is in main and missing a dependency that is not in main, you will have to file a MIR bug. https://wiki.ubuntu.com/MainInclusionProcess explains the procedure.

1.7.4 Corrigindo o problema

Once you have found a fix to the problem, follow the same process as any other bug. Make a patch, add it to a bzr branch or bug, subscribe ubuntu-sponsors, then try to get it included upstream and/or in Debian.

1.8 Bibliotecas compartilhadas

Bibliotecas compartilhadas são códigos compilados para serem compartilhados entre diversos programas diferentes. Eles são distribuídos como arquivos ".so" em "/usr/lib/".

Uma biblioteca exporta símbolos que são versões compiladas de funções, classes e variáveis. Uma biblioteca tem nome chamado de SONAME que inclui o número de versão. Esta versão SONAME não corresponde necessariamente ao número de versão de lançamento público. Um programa é compilado com uma versão SONAME da biblioteca. Se qualquer dos símbolos for removido ou alterado, então o número da versão precisa ser alterado, o que força todos os pacotes que usam aquela biblioteca a serem recompilados com a nova versão. A numeração de versão é geralmente definida pelo uspstream e nós a seguimos em nossos nomes de pacotes binários chamados numeração ABI. Mas às vezes o upstream não utiliza uma numeração de versão prática e os empacotadores têm que manter uma numeração de versão separada.

As bibliotecas são normalmente distribuídas pelo upstream como versões independentes. Às vezes elas são distribuídas como parte de um programa. Neste caso elas podem ser incluídas no pacote binário junto com o programa se você não espera que outros programas utilizem a biblioteca. Geralmente, elas devem ser divididas em pacotes binários separados.

As bibliotecas em si são colocadas em um pacote binário chamado "libfoo1" onde "foo" é o nome da biblioteca e "1" é a versão do SONAME. Arquivos de desenvolvimento do pacote, como os arquivos de cabeçalhos, necessários para compilar programas com a biblioteca, serão colocados em um pacote chamado "libfoo-dev".

1.8.1 Um exemplo

Vamos usar libnova como um exemplo:

```
$ bzr branch ubuntu:trusty/libnova
$ sudo apt-get install libnova-dev
```

Para encontra o SONAME de uma biblioteca, execute:

```
$ readelf -a /usr/lib/libnova-0.12.so.2 | grep SONAME
```

O SONAME é "libnova-0.12.so.2", que corresponde ao nome do arquivo (geralmente é o caso, mas nem sempre). Aqui o upstream colocou o número da versão upstream como parte do SONAME e deu a ele a versão ABI "2". Nomes de pacotes de bibliotecas devem seguir de acordo como SONAME das bibliotecas que contêm. O pacote binário de biblioteca "libnova-0.12-2" onde "libnova-0.12" é o nome da biblioteca e "2" é o nosso número ABI.

Se o upstream fizer alterações incompatíveis às suas bibliotecas, eles terão que reversionar o SONAME e nós teremos que renomear nossa biblioteca. Quaisquer outros pacotes usando o nosso pacote de biblioteca terão que ser recompilados com a nova versão, o que é chamado "transição" e pode exigir algum trabalho. Com sorte, nosso número ABI continuará a corresponder ao SONAME do uspstream, mas algumas vezes eles introduzem incompatibilidades sem alterar o seu número de versão e nós teremos que alterar o nosso.

Olhando em debian/libnova-0.12-2.install, vemos que inclui dos arquivos:

```
usr/lib/libnova-0.12.so.2
usr/lib/libnova-0.12.so.2.0.0
```

A última é a biblioteca real, completa com numeração de versão. A primeira é um link simbólico que aponta para a verdadeira biblioteca. O link simbólico é o que os programas que utilizam a biblioteca irão procurar, os programas em execução não se importam com alterações de versão menores.

`libnova-dev.install` includes all the files needed to compile a program with this library. Header files, a config binary, the `.la` libtool file and `libnova.so` which is another symlink pointing at the library, programs compiling against the library do not care about the major version number (although the binary they compile into will).

`.la` libtool files are needed on some non-Linux systems with poor library support but usually cause more problems than they solve on Debian systems. It is a current Debian goal to remove .la files and we should help with this.

1.8.2 Bibliotecas estáticas

The -dev package also ships `usr/lib/libnova.a`. This is a static library, an alternative to the shared library. Any program compiled against the static library will include the code directory into itself. This gets round worrying about binary compatibility of the library. However it also means that any bugs, including security issues, will not be updated along with the library until the program is recompiled. For this reason programs using static libraries are discouraged.

1.8.3 Arquivos de símbolos

When a package builds against a library the `shlibs` mechanism will add a package dependency on that library. This is why most programs will have `Depends: ${shlibs:Depends}` in `debian/control`. That gets replaced with the library dependencies at build time. However shlibs can only make it depend on the major ABI version number, `2` in our libnova example, so if new symbols get added in libnova 2.1 a program using these symbols could still be installed against libnova ABI 2.0 which would then crash.

Para fazer as dependências de uma biblioteca mais precisas, nós mantemos os arquivos ".symbols" que listam todos os símbolos em uma biblioteca e a versão em que eles aparecem.

libnova não tem arquivo de símbolos, então podemos criar um. Inicie compilando o pacote:


```
$ bzr builddeb -- -nc
```

O "-nc" irá fazer terminar no final da compilação sem remover os arquivos de compilação. Vá para o diretório de compilação e execute "dpkg-gensymbols" para o pacote de biblioteca:

```
$ cd ../build-area/libnova-0.12.2/
$ dpkg-gensymbols -plibnova-0.12-2 > symbols.diff
```

Isto gera um arquivo diff que você pode auto-aplicar:

```
$ patch -p0 < symbols.diff
```

Which will create a file named similar to `dpkg-gensymbolsnY_WWI` that lists all the symbols. It also lists the current package version. We can remove the packaging version from that listed in the symbols file because new symbols are not generally added by new packaging versions, but by the upstream developers:

```
$ sed -i s,-0ubuntu2,, dpkg-gensymbolsnY_WWI
```

Agora mova o arquivo para a seu novo local, submeta e faça uma construção de teste:

```
$ mv dpkg-gensymbolsnY_WWI ../../libnova/debian/libnova-0.12-2.symbols
$ cd ../../libnova
$ bzr add debian/libnova-0.12-2.symbols
$ bzr commit -m "add symbols file"
$ bzr builddeb
```

Se a compilação for bem-sucedida, o arquivo de símbolos está correto. Com a próxima versão usptream do libnova, se você executar dpkg-gensymbols novamente ele irá gerar um diff para atualizar o arquivo de símbolos.

1.8.4 Arquivos de símbolos de bibliotecas C++

C++ has even more exacting standards of binary compatibility than C. The Debian Qt/KDE Team maintain some scripts to handle this, see their Working with symbols files page for how to use them.

1.8.5 Leitura adicional

Junichi Uekawa's Debian Library Packaging Guide goes into this topic in more detail.

1.9 Backport de atualização de programas

Sometimes you might want to make new functionality available in a stable release which is not connected to a critical bug fix. For these scenarios you have two options: either you upload to a PPA or prepare a backport.

1.9.1 Arquivo de pacotes pessoal (PPA)

Usar um PPA tem várias vantagens. É bastante simples e você não precisa da aprovação de ninguém, mas o lado negativo disso é que seus usuários terão que ativá-lo manualmente. É um software de fonte não-padrão.

The PPA documentation on Launchpad is fairly comprehensive and should get you up and running in no time.

1.9.2 Ubuntu Backports oficial

O Projeto Backports é uma maneira de fornecer novos recursos aos usuários. Por causa dos riscos intrínsecos dos pacotes de backports, usuários não obtêm pacotes de backports sem algumas ações explícitas de suas partes. Isto geralmente faz com que backports sejam uma via inapropriada para correção de erros. Se um pacote em uma versão do Ubuntu possui um erro, ele deve ser corrigido através do *processo de atualização de segurança ou de atualização de versão estável*, como deve ser.

Uma vez que você tenha decidido que quer que se faça um backport de um pacote para uma versão estável, você precisará construí-lo e testá-lo nessa dada versão estável. "pbuider-dist" (no pacote "ubuntu-dev-tools") é uma ferramenta muito útil para se fazer isto facilmente.

Para relatar um pedido de backport e tê-lo processado polo time de Backport, você pode utilizar a ferramenta "request-backport" (também no pacote "ubuntu-dev-tools"). Isso irá determinar que as versões intermediárias para as quais será feito um backport do pacote, listar todas as dependências reversas, e arquivar o pedido de backport. Irá também incluir uma lista de verificação de teste no rastreador de erros.

CAPÍTULO 2

Base de conhecimento

2.1 Comunicação no desenvolvimento do Ubuntu

Em um projeto aonde milhares de linhas de código são alterados, muitas decisões são tomadas e centenas de pessoas interagem todos os dias, é importante se comunicar de forma eficaz.

2.1.1 Listas de e-mails

As listas de discussão são ferramentas muito importantes, se você quiser divulgar suas idéias para uma ampla equipe e ter certeza de que você alcance todos, até mesmo através de fusos horários diferentes.

Em termos de desenvolvimento, estas são as mais importantes:

- https://lists.ubuntu.com/mailman/listinfo/ubuntu-devel-announce (apenas anúncios, os mais importantes anúncios de desenvolvimento aqui)
- https://lists.ubuntu.com/mailman/listinfo/ubuntu-devel (discussão geral de desenvolvimento do Ubuntu)
- https://lists.ubuntu.com/mailman/listinfo/ubuntu-motu (Time de discussão MOTU, obter ajuda com empacotamento)

2.1.2 Canais de IRC

Para discussões em tempo real, por favor conecte-se na rede irc.freenode.net e entre em qualquer um desses canais:

- #ubuntu-devel (para discussão geral de desenvolvimento)
- #ubuntu-motu (Time de discussão MOTU e obtenção de ajuda em geral)

2.2 Visão básica do diretório `debian/`

Este artigo explicará sucintamente sobre os diferentes arquivos contidos no diretório `debian/`, importantes ao empacotamento Ubuntu. Os mais importantes são `changelog`, `control`, `copyright` e `rules`. Eles são obrigatórios para todos os pacotes. Diversos arquivos adicionais dentro de `debian/` podem ser usados para personalizar e configurar o comportamento do pacote. Alguns desses arquivos são abordados neste artigo, o que não significa que esta seja a lista completa.

2.2.1 O registro de mudanças

Este arquivo é, como seu nome implica, uma listagem das mudanças feitas em cada versão. Tem um formato específico que dá o nome do pacote, versão, distribuição, mudanças e quem fez as mudanças em um momento determinado. Se você tiver uma chave GPG (see: *Getting set up*), certifique-se de usar no `changelog` o mesmo nome e endereço de e-mail da chave. O documento a seguir é um modelo de `changelog`:

```
package (version) distribution; urgency=urgency

 * change details
  - more change details
 * even more change details

-- maintainer name <email address>[two spaces]  date
```

O formato (principalmente da data) é importante. A data deve estar no formato da **RFC 5322**, o qual pode ser obtido pelo uso do comando `date -R`. Por conveniência, o comando `dch` pode ser usado para editar o arquivo de mudanças. Ele atualizará a data automaticamente.

Pontos menos importantes são indicados por um traço "-", enquanto os pontos mais importantes usam um asterisco "*".

Se você estiver empacotando do zero, `dch --create` (`dch` está no pacote `devscripts`) criará um `debian/changelog` padrão para você.

Aqui está um exemplo de arquivo "changelog" para hello:

```
hello (2.8-0ubuntu1) trusty; urgency=low

  * New upstream release with lots of bug fixes and feature improvements.

-- Jane Doe <packager@example.com>  Thu, 21 Oct 2013 11:12:00 -0400
```

Observe que a versão tem um `-0ubuntu1` acrescentado a ela, isso é a revisão da distro, usada para que o empacotamento possa ser atualizado (para corrigir falhas, por exemplo) com novos envios para a mesma versão de lançamento do fonte.

Ubuntu e Debian são ligeiramente diferentes nos esquemas de versionamento de pacotes para evitar pacotes conflitantes com a mesma versão de código fonte. Se um pacote Debian foi mudado no Ubuntu, ele terá `ubuntuX` (onde `X` é o número da revisão Ubuntu) acrescentado ao final da versão Debian. Então, se o pacote Debian hello `2.6-1` sofresse alterações pelo Ubuntu, sua versão seria `2.6-1ubuntu1`. Se um pacote para determinada aplicação não existir no Debian, então a revisão Debian será `0` (exemplo, `2.6-0ubuntu1`).

For further information, see the changelog section (Section 4.4) of the Debian Policy Manual.

2.2.2 O arquivo de controle

O arquivo `control` contém a informação que os gerenciadores de pacotes (tais como `apt-get`, `synaptic` e `adept`) usam, como dependências em tempo de construção de pacote, informações sobre o mantenedor e muito mais.

Para o pacote `hello` do Ubuntu, o arquivo `control` parece alguma coisa assim:

```
Source: hello
Section: devel
Priority: optional
Maintainer: Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>
XSBC-Original-Maintainer: Jane Doe <packager@example.com>
Standards-Version: 3.9.5
Build-Depends: debhelper (>= 7)
```



```
Vcs-Bzr: lp:ubuntu/hello
Homepage: http://www.gnu.org/software/hello/

Package: hello
Architecture: any
Depends: ${shlibs:Depends}
Description: The classic greeting, and a good example
 The GNU hello program produces a familiar, friendly greeting. It
 allows non-programmers to use a classic computer science tool which
 would otherwise be unavailable to them. Seriously, though: this is
 an example of how to do a Debian package. It is the Debian version of
 the GNU Project's 'hello world' program (which is itself an example
 for the GNU Project).
```

O primeiro parágrafo descreve o pacote fonte, incluindo a lista de pacotes necessários para construir o pacote a partir do fonte no campo `Build-Depends`. Também contém algumas meta-informações tais como o nome do mantenedor, a versão de Política Debian que cumpre tal pacote, a localização do repositório de controle de versão de empacotamento e a página de internet do programa original.

Note that in Ubuntu, we set the `Maintainer` field to a general address because anyone can change any package (this differs from Debian where changing packages is usually restricted to an individual or a team). Packages in Ubuntu should generally have the `Maintainer` field set to `Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>`. If the Maintainer field is modified, the old value should be saved in the `XSBC-Original-Maintainer` field. This can be done automatically with the `update-maintainer` script available in the `ubuntu-dev-tools` package. For further information, see the Debian Maintainer Field spec on the Ubuntu wiki.

Cada parágrafo adicional descreve um pacote binário a ser construído.

For further information, see the control file section (Chapter 5) of the Debian Policy Manual.

2.2.3 O arquivo de direitos autorais

This file gives the copyright information for both the upstream source and the packaging. Ubuntu and Debian Policy (Section 12.5) require that each package installs a verbatim copy of its copyright and license information to `/usr/share/doc/$(package_name)/copyright`.

Geralmente, informações sobre direitos autorais são encontradas no arquivo `COPYING` do diretório de arquivos fonte do programa. Este arquivo deve incluir informações como os nomes dos autores e do empacotador, a URL da qual o arquivo fonte foi baixado, uma linha com o ano e o titular do direito autoral e o próprio texto de direito autoral. Um exemplo de padrão seria:

```
Format: http://www.debian.org/doc/packaging-manuals/copyright-format/1.0/
Upstream-Name: Hello
Source: ftp://ftp.example.com/pub/games

Files: *
Copyright: Copyright 1998 John Doe <jdoe@example.com>
License: GPL-2+

Files: debian/*
Copyright: Copyright 1998 Jane Doe <packager@example.com>
License: GPL-2+

License: GPL-2+
 This program is free software; you can redistribute it
 and/or modify it under the terms of the GNU General Public
```



```
License as published by the Free Software Foundation; either
version 2 of the License, or (at your option) any later
version.
.
This program is distributed in the hope that it will be
useful, but WITHOUT ANY WARRANTY; without even the implied
warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.  See the GNU General Public License for more
details.
.
You should have received a copy of the GNU General Public
License along with this package; if not, write to the Free
Software Foundation, Inc., 51 Franklin St, Fifth Floor,
Boston, MA  02110-1301 USA
.
On Debian systems, the full text of the GNU General Public
License version 2 can be found in the file
`/usr/share/common-licenses/GPL-2'.
```

This example follows the Machine-readable debian/copyright format. You are encouraged to use this format as well.

2.2.4 O arquivo de regras

O último arquivo que devemos olhar é o `rules`. Ele faz todo trabalho de criar nosso pacote. Ele é um Makefile com regras para compilar e instalar a aplicação, então criar o arquivo `.deb` a partir dos arquivos instalados. Ele também tem uma regra para limpar todos os arquivos de construção para que você possa apenas com os arquivos fonte.

Aqui é uma versão simplificada do arquivo de regras criado pelo `dh_make` (que pode ser encontrado no pacote `dh-make`):

```
#!/usr/bin/make -f
# -*- makefile -*-

# Uncomment this to turn on verbose mode.
#export DH_VERBOSE=1

%:
	dh $@
```

Vamos passar com mais detalhes sobre este arquivo. O que ele faz é passar por todas as regras de construção que `debian/rules` é chamado com argumentos para `/usr/bin/dh`, no qual ele mesmo chamará todos os comandos `dh_*` necessários.

"dh" executa uma sequência de comandos debhelper. As sequências suportadas correspondem às regras de um arquivo "debian/rules": "build", "clean", "install", "binary-arch", "binary-indep" e "binary". Para ver quais comandos são executados em cada regra, execute:

```
$ dh binary-arch --no-act
```

Comandos na sequência binary-indep são passados com a opção "-i" para garantir que funcionarão apenas em pacotes binário independentes e comandos na sequência binary-arch são passados com a opção "-a" para garantir que funcionarão apenas em pacotes dependentes de arquiteturas.

Cada comando debhelper gravará em `debian/package.debhelper.log` quando executado com sucesso. (O qual dh_clean apaga.) então dh pode dizer quais comandos foram executados, para quais pacotes e assim evitar rodar os mesmos comandos novamente.

Cada vez que o dh roda, ele examina o registro e procura pelo último comando registrado que está numa sequência especificada. Então, ele continua com o comando seguinte. As opções *–until'*, --before, --after e --remaining podem sobrescrever este comportamento.

Se o "debian/rules" contiver um alvo com um nome como "override_dh_command", então quando ele receber este comando o "dh" executará aquele alvo a partir do arquivo rules, em vez de executar o comando real. A substituição do alvo pode então executar o comando com opções adicionais, ou executar comandos totalmente diferentes. (Note que para usar este recurso, você deve construir as dependências com o debhelper 7.0.50 ou superior.)

Veja em /usr/share/doc/debhelper/examples/ e man dh para mais exemplos. Veja também a seção de regras (Seção 4.9) do Manual de Políticas do Debian.

2.2.5 Arquivos adicionais

O arquivo de instalação

O arquivo install' é usado por ''dh_install para instalar arquivos no pacote binário. Ele possui dois casos de uso padrão:

- Para arquivos de instalação dentro do seu pacote que não são manipulados pelo sistema de construção do upstream.
- Dividindo um grande e único pacote fonte em múltiplos pacotes binários.

No primeiro caso, o arquivo install deveria ter uma linha por arquivo instalado, especificando ambos arquivos e o local da instalação. Por exemplo, o seguinte arquivo install instalaria o script foo na fonte do pacote do diretório raiz para usr/bin e um atalho no diretório debian para usr/share/applications:

```
foo usr/bin
debian/bar.desktop usr/share/applications
```

When a source package is producing multiple binary packages dh will install the files into debian/tmp rather than directly into debian/<package>. Files installed into debian/tmp can then be moved into separate binary packages using multiple $package_name.install files. This is often done to split large amounts of architecture independent data out of architecture dependent packages and into Architecture: all packages. In this case, only the name of the files (or directories) to be installed are needed without the installation directory. For example, foo.install containing only the architecture dependent files might look like:

```
usr/bin/
usr/lib/foo/*.so
```

Enquanto foo-common.install contendo apenas o arquivo independente da arquitetura pode parecer como:

```
/usr/share/doc/
/usr/share/icons/
/usr/share/foo/
/usr/share/locale/
```

Isso criará dois pacotes binários, foo e foo-common. Ambos necessitam de seu próprio parágrafo em debian/control.

Veja "man dh_install" e a "seção de arquivo de instalação (Seção 5,11) <http://www.debian.org/doc/manuals/maint-guide/dother.en.html#install>"_ do Novo Guia dos Mantenedores do Debian para detalhes adicionais.

O arquivo watch

O arquivo "debian/watch" nos permite verificar automaticamente a existência de novas versões no upstream usando a ferramenta "uscan" encontrada no pacote "devscripts". A primeira linha do arquivo watch deve ser a versão do formato

(3, quando este texto foi escrito), enquanto que as linhas seguintes contém quaisquer URLs a serem analisadas. Por exemplo:

```
version=3

http://ftp.gnu.org/gnu/hello/hello-(.*).tar.gz
```

Ao executar `uscan` no diretório raiz dos fontes irá comparar o número da versão no upstream em `debian/changelog` com a última versão disponível no upstream. Se uma nova versão upstream for encontrada, ela será baixada automaticamente. Por exemplo:

```
$ uscan
hello: Newer version (2.7) available on remote site:
  http://ftp.gnu.org/gnu/hello/hello-2.7.tar.gz
  (local version is 2.6)
hello: Successfully downloaded updated package hello-2.7.tar.gz
    and symlinked hello_2.7.orig.tar.gz to it
```

If your tarballs live on Launchpad, the `debian/watch` file is a little more complicated (see Question 21146 and Bug 231797 for why this is). In that case, use something like:

```
version=3
https://launchpad.net/flufl.enum/+download http://launchpad.net/flufl.enum/.*/flufl.enum-(.+).tar.gz
```

Para maiores informações, veja `man uscan` e a seção do arquivo watch (Seção 4.11) do Manual de políticas do Debian.

Para uma lista dos pacotes cujo arquivo "watch" relata não estarem sincronizados com o upstream, consulte Ubuntu External Health Status.

O arquivo source/format

Este arquivo indica o formato do pacote fonte. Ele deve conter uma única linha indicando o formato desejado:

- `3.0 (native)` para pacotes nativos do Debian (sem versão do upstream)
- `3.0 (quilt)` para pacotes com um tarball separado do upstream
- `1.0` para pacotes que desejam declarar explicitamente o formato padrão

Atualmente, o formato fonte do pacote será padronizada para 1.0 se o arquivo não existir. Você pode fazer isso explicitamente no arquivo/formato fonte. Se você escolher não utilizar este arquivo para definir o formato fonte, o Lintian irá lhe avisar sobre o arquivo ausente. Este aviso é apenas uma informação e pode ser ignorado com segurança.

Você é encorajado a usar o mais recente formato de fonte, o 3.0. Ele fornece uma quantia de novos recursos:

- Suporte para formatos de compressão adicionais: bzip2, lzma e xz
- Suporte para múltiplos tarballs do upstream
- Não é necessário reempacotar o tarball do upstream para retirar o diretório debian
- Mudanças específicas do Debian não são armazenadas em um único .diff.gz, mas em múltiplos patches compatíveis com o quilt em `debian/patches/`

https://wiki.debian.org/Projects/DebSrc3.0 summarizes additional information concerning the switch to the 3.0 source package formats.

See `man dpkg-source` and the source/format section (Section 5.21) of the Debian New Maintainers' Guide for additional details.

2.2.6 Recursos adicionais

In addition to the links to the Debian Policy Manual in each section above, the Debian New Maintainers' Guide has more detailed descriptions of each file. Chapter 4, "Required files under the debian directory" further discusses the control, changelog, copyright and rules files. Chapter 5, "Other files under the debian directory" discusses additional files that may be used.

2.3 ubuntu-dev-tools: Tools for Ubuntu developers

`ubuntu-dev-tools` package is a collection of 30 tools created for making packaging work much easier for Ubuntu developers. It's similar in scope to Debian `devscripts` package.

2.3.1 Setting up packaging environment

`setup-packaging-environment` command allows to interactively set up packaging environment, including setting environment variables, installing required packages and ensuring that required repositories are enabled.

2.3.2 Environment variables

Introducing yourself

`ubuntu-dev-tools` configurations can be set using environment variables. It's used for example in changelogs. For example, to set e-mail address (and full name), use `UBUMAIL` variable. It overrides the `DEBEMAIL` and `DEBFULLNAME` variables used by `devscripts`. To learn `ubuntu-dev-tools` about you, open *~/.bashrc* in text editor and add something like this:

```
export UBUMAIL="Marcin Mikołajczak <marcin@example.org>"
```

Now, save this file and restart your terminal or use `source ~/.bashrc`.

Changing preferred builder

Default builder is specified by `UBUNTUTOOLS_BUILDER` variable. To set between *pbuilder* (default), *pbuilder-dist*, and *sbuild*, change this variable. Example:

```
export UBUNTUTOOLS_BUILDER=sbuild
```

Save file and restart terminal.

You can also check whether to update the builder every time before building, by changing `UBUNTUTOOLS_UPDATE_BUILDER` from `no` (default) to `yes`.

2.3.3 Downloading source packages

`ubuntu-dev-tools` comes with `pull-lp-source` command, allowing to download source packages from Launchpad. Its usage is simple. To download latest source package for ubuntu-settings, use:

```
$ pull-lp-source ubuntu-settings
```

You can also specify release from which you want to download source or specify version of source package. `-d` option allows to download source package without extracting. A slightly more complex example would look like this:


```
$ pull-lp-source brisk-menu 0.5.0-1 -d
```

`pull-debian-source` package allows to do the same for Debian source packages. It has similar syntax.

2.3.4 Backporting packages

`ubuntu-dev-tools` provides `backportpackage` allowing us to backport a package from specified release of Ubuntu or Debian. For example, to backport `bzr` package from latest development release for your installed Ubuntu version, simply:

```
$ backportpackage -w . bzr
```

This command allows to use more options. To specify Ubuntu release for which you are going to backport a package, use `-d dest` or `--destination=DEST` parameter, where `DEST` is Ubuntu release, for example `xenial`. You can specify more than one destination. In turn, `-s SOURCE` and `--source=SOURCE` specifies the Ubuntu or Debian release from which you are going to backport a package. `-w DIR` and `--workdir=DIR` specifies directory, where package files will be downloaded, unpacked and built. By default, it will create temporary directory that will be automatically deleted. `-U` or `--update` allows to update build environment before building package. `-u` or `--upload` allows to upload package after building (for example to PPAs) using `dput`.

2.3.5 Requesting backports

`requestbackport` command makes creating backports through Launchpad bugs much easier. It creates testing checklist that will be included in the bug. For example, to request backporting libqt5webkit5 from latest development branch to current stable release (without optional parameters):

```
$ requestbackport libqt5webkit5
```

You should fulfill the checklist if you have already tested the backport.

Additional options allows to specify destination of backport and its source, by using `-d DEST` or `--destination=DEST` and `s SRC` or `--source=SRC`.

2.3.6 Other simple commands

`ubuntu-dev-tools` also includes small utilities allowing to do simple tasks like checking whether .iso file is an Ubuntu installation media.

ubuntu-iso

To do this, use `ubuntu-iso <pathtoiso>`, for example:

```
$ ubuntu-iso ~/Downloads/ubuntu.iso
```

bitesize

“Bitesize” tag is used on Launchpad to describe tasks that are suitable for begineers who want to contribute to one of the projects. `bitesize` command allows to add “bitesize” tag to Launchpad bug with just simple command, by providing its number, like:

```
$ bitesize 1735410
```


404main

`404main` allows to check whether all of package build dependencies are included in main repository of specified Ubuntu distribution. Example:

```
$ 404main libqt5webkit5 xenial
```

If any of the required packages isn't part of Ubuntu main repository, you can check whether the package fulfill Ubuntu main inclusion requirements and request it.

Further reading

`ubuntu-dev-tools` manpages are covering more about usage of this package.

2.4 autopkgtest: Teste automático para pacotes

The DEP 8 specification defines how automatic testing can very easily be integrated into packages. To integrate a test into a package, all you need to do is:

- adicione um arquivo chamado `debian/tests/control` que especifica os requisitos para o testbed,
- adicione os testes em `debian/tests/`.

2.4.1 Requisitos do Testbed

In `debian/tests/control` you specify what to expect from the testbed. So for example you list all the required packages for the tests, if the testbed gets broken during the build or if `root` permissions are required. The DEP 8 specification lists all available options.

Abaixo estamos vendo o pacote fonte do `glib2.0`. Em um caso muito simples, o arquivo se pareceria com isto:

```
Tests: build
Depends: libglib2.0-dev, build-essential
```

Para o teste em `debian/tests/build` isto deverá garantir que os pacotes `libglib2.0-dev` e `build-essential` estão instalados.

Nota: Você pode usar `@` na linha `Depends` para indicar que você quer que todos os pacotes instalados sejam compilados pelo pacote fonte em questão.

2.4.2 Os testes reais

O teste que deve acompanhar o exemplo acima poderia ser:

```
#!/bin/sh
# autopkgtest check: Build and run a program against glib, to verify that the
# headers and pkg-config file are installed correctly
# (C) 2012 Canonical Ltd.
# Author: Martin Pitt <martin.pitt@ubuntu.com>

set -e

WORKDIR=$(mktemp -d)
```



```
trap "rm -rf $WORKDIR" 0 INT QUIT ABRT PIPE TERM
cd $WORKDIR
cat <<EOF > glibtest.c
#include <glib.h>

int main()
{
    g_assert_cmpint (g_strcmp0 (NULL, "hello"), ==, -1);
    g_assert_cmpstr (g_find_program_in_path ("bash"), ==, "/bin/bash");
    return 0;
}
EOF

gcc -o glibtest glibtest.c `pkg-config --cflags --libs glib-2.0`
echo "build: OK"
[ -x glibtest ]
./glibtest
echo "run: OK"
```

Aqui um pedaço simples de código em C é escrito em um diretório temporário. Então este é compilado com bibliotecas do sistema (usando flags e caminhos de biblioteca providos por *pkg-config*). Em seguida o binário compilado, que apenas utiliza partes da funcionalidade do núcleo glib, é executado.

While this test is very small and simple, it covers quite a lot: that your -dev package has all necessary dependencies, that your package installs working pkg-config files, headers and libraries are put into the right place, or that the compiler and linker work. This helps to uncover critical issues early on.

2.4.3 Executando o teste

While the test script can be easily executed on its own, it is strongly recommended to actually use `autopkgtest` from the `autopkgtest` package for verifying that your test works; otherwise, if it fails in the Ubuntu Continuous Integration (CI) system, it will not land in Ubuntu. This also avoids cluttering your workstation with test packages or test configuration if the test does something more intrusive than the simple example above.

The README.running-tests (online version) documentation explains all available testbeds (schroot, LXD, QEMU, etc.) and the most common scenarios how to run your tests with `autopkgtest`, e. g. with locally built binaries, locally modified tests, etc.

The Ubuntu CI system uses the QEMU runner and runs the tests from the packages in the archive, with `-proposed` enabled. To reproduce the exact same environment, first install the necessary packages:

```
sudo apt install autopkgtest qemu-system qemu-utils autodep8
```

Now build a testbed with:

```
autopkgtest-buildvm-ubuntu-cloud -v
```

(Please see its manpage and `--help` output for selecting different releases, architectures, output directory, or using proxies). This will build e. g. `adt-trusty-amd64-cloud.img`.

Then run the tests of a source package like `libpng` in that QEMU image:

```
autopkgtest libpng --- qemu adt-trusty-amd64-cloud.img
```

The Ubuntu CI system runs packages with only selected packages from `-proposed` available (the package which caused the test to be run); to enable that, run:


```
autopkgtest libpng -U --apt-pocket=proposed=src:foo --- qemu adt-release-amd64-cloud.img
```

or to run with all packages from `-proposed`:

```
autopkgtest libpng -U --apt-pocket=proposed --- qemu adt-release-amd64-cloud.img
```

The `autopkgtest` manpage has a lot more valuable information on other testing options.

2.4.4 Outros exemplos

Esta lista não é completa, mas pode ajudá-lo a ter uma ideia melhor de como testes automatizados são implementados e usados no Ubuntu.

- The libxml2 tests are very similar. They also run a test-build of a simple piece of C code and execute it.
- The gtk+3.0 tests also do a compile/link/run check in the "build" test. There is an additional "python3-gi" test which verifies that the GTK library can also be used through introspection.
- In the ubiquity tests the upstream test-suite is executed.
- The gvfs tests have comprehensive testing of their functionality and are very interesting because they emulate usage of CDs, Samba, DAV and other bits.

2.4.5 Infraestrutura do Ubuntu

Packages which have `autopkgtest` enabled will have their tests run whenever they get uploaded or any of their dependencies change. The output of automatically run autopkgtest tests can be viewed on the web and is regularly updated.

Debian also uses `autopkgtest` to run package tests, although currently only in schroots, so results may vary a bit. Results and logs can be seen on http://ci.debian.net. So please submit any test fixes or new tests to Debian as well.

2.4.6 Obtendo o teste no Ubuntu

O processo de envio de um autopkgtest para um pacote é muito semelhante a *corrigindo um erro no Ubuntu*. Essencialmente, você simplesmente:

- execute `bzr branch ubuntu:<nomedopacote>`,
- edite `debian/control` para habilitar os testes,
- adiciona o diretório `debian/tests`,
- write the `debian/tests/control` based on the DEP 8 Specification,
- adicione seu(s) caso(s) de teste em `debian/tests`,
- submeta suas mudanças, envie elas para o Launchpad, proponha a mesclagem e obtenha revisões apenas como qualquer outra melhoria no pacote fonte.

2.4.7 O que você pode fazer

The Ubuntu Engineering team put together a list of required test-cases, where packages which need tests are put into different categories. Here you can find examples of these tests and easily assign them to yourself.

If you should run into any problems, you can join the #ubuntu-quality IRC channel to get in touch with developers who can help you.

2.5 Usando o chroot

Se você está executando uma versão do Ubuntu mas os pacotes que está trabalhando são de outras versões você pode criar um ambiente da outra versão com um `chroot`.

Um `chroot` lhe permite ter um sistema de arquivos completo a partir de outra distribuição, no qual se pode trabalhar normalmente. Isto evita a sobrecarga de uma máquina virtual inteira.

2.5.1 Criando um chroot

Use o comando `debootstrap` para criar um novo chroot:

```
$ sudo debootstrap trusty trusty/
```

This will create a directory `trusty` and install a minimal trusty system into it.

If your version of `debootstrap` does not know about Trusty you can try upgrading to the version in `backports`.

Você pode então trabalhar dentro do chroot:

```
$ sudo chroot trusty
```

Onde você pode instalar ou remover qualquer pacote que desejar sem afetar seu sistema principal.

Você pode querer copiar suas chaves GPG/SSH e configuração do Bazaar para dentro do chroot, então você pode acessar e assinar pacotes diretamente:

```
$ sudo mkdir trusty/home/<username>
$ sudo cp -r ~/.gnupg ~/.ssh ~/.bazaar trusty/home/<username>
```

Para impedir que o apt e outros programas reclamem de falta de localizações, você deve instalar o pacote de linguagem relevante para você:

```
$ apt-get install language-pack-en
```

Se você quer executar programas X será necessário vincular o diretório /tmp com o chroot, de fora do chroot execute:

```
$ sudo mount -t none -o bind /tmp trusty/tmp
$ xhost +
```

Alguns programas podem necessitar que os diretórios /dev ou /proc estejam vinculados com o chroot.

For more information on chroots see our Debootstrap Chroot wiki page.

2.5.2 Alternativas

SBuild is a system similar to PBuilder for creating an environment to run test package builds in. It closer matches that used by Launchpad for building packages but takes some more setup compared to PBuilder. See the Security Team Build Environment wiki page for a full explanation.

Full virtual machines can be useful for packaging and testing programs. TestDrive is a program to automate syncing and running daily ISO images, see the TestDrive wiki page for more information.

Você também pode configurar o pbuilder para pausar quando ele vem através de uma falha de construção. Copie C10shell de /usr/share/doc/pbuilder/examples para dentro do diretório e use o argumento `--hookdir=` para apontá-lo.

Amazon's EC2 cloud computers allow you to hire a computer paying a few US cents per hour, you can set up Ubuntu machines of any supported version and package on those. This is useful when you want to compile many packages at the same time or to overcome bandwidth restraints.

2.6 Setting up sbuild

`sbuild` simplifies building Debian/Ubuntu binary package from source in clean environment. It allows to try debugging packages in environment similar (as opposed to `pbuild`) to builders used by Launchpad.

It works on different architectures and allows to build packages for other releases. It needs kernel supporting overlayfs.

2.6.1 Installing sbuild

To use sbuild, you need to install sbuild and other required packages and add yourself to the `sbuild` group:

```
$ sudo apt install debhelper sbuild schroot ubuntu-dev-tools
$ sudo adduser $USER sbuild
```

Create `.sbuildrc` in your home directory with following content:

```
# Name to use as override in .changes files for the Maintainer: field
# (mandatory, no default!).
$maintainer_name='Your Name <user@example.org>';

# Default distribution to build.
$distribution = "bionic";
# Build arch-all by default.
$build_arch_all = 1;

# When to purge the build directory afterwards; possible values are "never",
# "successful", and "always".  "always" is the default. It can be helpful
# to preserve failing builds for debugging purposes.  Switch these comments
# if you want to preserve even successful builds, and then use
# "schroot -e --all-sessions" to clean them up manually.
$purge_build_directory = 'successful';
$purge_session = 'successful';
$purge_build_deps = 'successful';
# $purge_build_directory = 'never';
# $purge_session = 'never';
# $purge_build_deps = 'never';

# Directory for writing build logs to
$log_dir=$ENV{HOME}."/ubuntu/logs";

# don't remove this, Perl needs it:
1;
```

Replace "Your Name <user@example.org>" with your name and e-mail address. Change default distribution if you want, but remember that you can specify target distribution when executing command.

If you haven't restarted your session after adding yourself to the `sbuild` group, enter:

```
$ sg sbuild
```

Generate GPG keypair for sbuild and create chroot for specified release:


```
$ sbuild-update --keygen
$ mk-sbuild bionic
```

This will create chroot for your current architecture. You might want to specify another architecture. For this, you can use `--arch` option. Example:

```
$ mk-sbuild xenial --arch=i386
```

2.6.2 Using schroot

Entering schroot

You can use `schroot -c <release>-<architecture> [-u <USER>]` to enter newly created chroot, but that's not exactly the reason why you are using sbuild:

```
$ schroot -c bionic-amd64 -u root
```

Using schroot for package building

To build package using sbuild chroot, we use (surprisingly) the `sbuild` command. For example, to build `hello` package from x86_64 chroot, after applying some changes:

```
apt source hello
cd hello-*
sed -i -- 's/Hello/Goodbye/g' src/hello.c   # some
sed -i -- 's/Hello/Goodbye/g' tests/hello-1 #
dpkg-source --commit
dch -i                                      #
update-maintainer                           # changes
sbuild -d bionic-amd64
```

To build package from source package (`.dsc`), use location of the source package as second parameter:

```
sbuild -d bionic-amd64 ~/packages/goodbye_*.dsc
```

To make use of all power of your CPU, you can specify number of threads used for building using standard `-j<threads>`:

```
sbuild -d bionic-amd64 -j8
```

2.6.3 Maintaining schroots

Listing chroots

To get list of all your sbuild chroots, use `schroot -l`. The `source:` chroots are used as base of new schroots. Changes here aren't recommended, but if you have specific reason, you can open it using something like:

```
$ schroot -c source:bionic-amd64
```

Updating schroots

To upgrade the whole schroot:


```
$ sbuild-update -ubc bionic-amd64
```

Expiring active schroots

If because of any reason, you haven't stopped your schroot, you can expire all active schroots using:

```
$ schroot -e --all-sessions
```

2.6.4 Further reading

There is Debian wiki page covering sbuild usage.

Ubuntu Wiki also has article about basics of sbuild.

`sbuild` manpages are covering details about sbuild usage and available features.

2.7 Empacotamento do KDE

Packaging of KDE programs in Ubuntu is managed by the Kubuntu and MOTU teams. You can contact the Kubuntu team on the Kubuntu mailing list and `#kubuntu-devel` Freenode IRC channel. More information about Kubuntu development is on the Kubuntu wiki page.

Our packaging follows the practices of the Debian Qt/KDE Team and Debian KDE Extras Team. Most of our packages are derived from the packaging of these Debian teams.

2.7.1 Politica de patches

O Kubuntu não adiciona patches aos programas do KDE a menos que venham dos autores do upstream ou submetidos ao upstream com a expectativa de que sejam mesclados logo ou que tenhamos consultado a questão com os autores upstream.

O Kubuntu não altera o nome dos pacotes exceto quando o upstream o requer (como o logo no canto superior esquerdo do menu Kickoff) ou para simplificar (como remover telas de inicialização).

2.7.2 debian/rules

Os pacotes do Debian incluem algumas adições ao uso básico do Debhelper. Elas ficam no pacote "pkg-kde-tools".

Pacotes que usam o Debhelper 7 devem adicionar a opção "–with=kde". Isto irá assegurar que os sinalizadores de construção corretos estão sendo usados e adicionar opções como manuseio de stubs do kdeinit e traduções:

```
%:
    dh $@ --with=kde
```

Alguns pacotes KDE mais novos usam o sistema "dhmk", uma alternativa ao "dh" feito pelo time do Debian Qt/KDE. Você pode ler sobre ele em /usr/share/pkg-kde-tools/qt-kde-team/2/README. Pacotes que o utilizam irão "incluir /usr/share/pkg-kde-tools/qt-kde-team/2/debian-qt-kde.mk" em vez de executar "dh".

2.7.3 Traduções

Os pacotes do main tem as suas traduções importadas no Launchpad e exportadas do Launchpad para o os pacotes de idioma do Ubuntu.

Então todos os pacotes KDE devem gerar modelos de tradução, incluir ou deixar disponível traduções upstream e manipular traduções de arquivos ".desktop".

Para gerar modelos de tradução o pacote precisa incluir um arquivo "Messages.sh". Reclame ao upstream caso isto não aconteça. Você pode verificar se ele funciona executando "extract-messages.sh" que deve produzir um ou mais arquivos ".pot" em "po/". Isto será feito automaticamente durante a construção se você usar a opção "–with=kde" para "dh".

Upstream will usually have also put the translation `.po` files into the `po/` directory. If they do not, check if they are in separate upstream language packs such as the KDE SC language packs. If they are in separate language packs Launchpad will need to associate these together manually, contact David Planella to do this.

Se um pacote é movido de "universe" para "main", ele precisará ser reenviado antes que as traduções sejam importadas para o Launchpad.

`.desktop` files also need translations. We patch KDELibs to read translations out of `.po` files which are pointed to by a line `X-Ubuntu-Gettext-Domain=` added to `.desktop` files at package build time. A .pot file for each package is be generated at build time and .po files need to be downloaded from upstream and included in the package or in our language packs. The list of .po files to be downloaded from KDE's repositories is in `/usr/lib/kubuntu-desktop-i18n/desktop-template-list`.

2.7.4 Símbolos de bibliotecas

Library symbols are tracked in `.symbols` files to ensure none go missing for new releases. KDE uses C++ libraries which act a little differently compared to C libraries. Debian's Qt/KDE Team have scripts to handle this. See Working with symbols files for how to create and keep these files up to date.

CAPÍTULO 3

Leitura adicional

You can read this guide offline in different formats, if you install one of the binary packages.

If you want to learn more about building Debian packages, here are some Debian resources you may find useful:

- How to package for Debian;
- Debian Policy Manual;
- Debian New Maintainers' Guide — available in many languages;
- Packaging tutorial (also available as a package);
- Guide for Packaging Python Modules.

We are always looking to improve this guide. If you find any problems or have some suggestions, please report a bug on Launchpad. If you'd like to help work on the guide, grab the source there as well.