Guida per il nuovo Maintainer

COLLABORATORI

	TITOLO : Guida per il nuovo Maintainer	

AZIONE	*NOME*	*DATA*	*FIRMA*
A CURA DI	Josip Rodin, Osamu Aoki, Calogero Lo Leggio, Jacopo Reggiani, e Francesco P. Lovergine	3 luglio 2020	
Traduzione italiana		3 luglio 2020	
Traduzione italiana		3 luglio 2020	
Traduzione italiana		3 luglio 2020	

CRONOLOGIA DELLE REVISIONI

POSIZIONE	DATA	DESCRIZIONE	NOME

Indice

1 Partire nel modo giusto **1**

1.1 Dinamiche sociali di Debian . . . 1
1.2 Programmi necessari per lo sviluppo . . . 3
1.3 Documentazione necessaria per lo sviluppo . . . 4
1.4 Dove chiedere aiuto . . . 5

2 Primi passi **6**

2.1 Flusso di lavoro per la costruzione dei pacchetti Debian . . . 6
2.2 Scegliamo il programma . . . 7
2.3 Prendere il programma e provarlo . . . 9
2.4 Sistemi di compilazione semplici . . . 10
2.5 Sistemi di compilazione portabili più utilizzati . . . 10
2.6 Nome e versione del pacchetto . . . 11
2.7 Configurare **dh_make** . . . 12
2.8 Il primo pacchetto non nativo per Debian . . . 12

3 Modificare i sorgenti **14**

3.1 Configurare **quilt** . . . 14
3.2 Correggere i bug nel sorgente originale . . . 14
3.3 Installazione dei file nei loro percorsi . . . 15
3.4 Distinguere le librerie . . . 17

4 File richiesti nella directory `debian` **19**

4.1 `control` . . . 19
4.2 `copyright` . . . 23
4.3 `changelog` . . . 24
4.4 Il file `rules` . . . 25
4.4.1 Target del file `rules` . . . 25
4.4.2 Il file `rules` predefinito . . . 26
4.4.3 Personalizzazione del file `rules` . . . 29

5 Altri file nella directory `debian` 32

5.1 `README.Debian` . . . 32
5.2 `compat` . . . 33
5.3 `conffiles` . . . 33
5.4 *`pacchetto`*`.cron.*` . . . 33
5.5 `dirs` . . . 34
5.6 *`pacchetto`*`.doc-base` . . . 34
5.7 `docs` . . . 34
5.8 `emacsen-*` . . . 34
5.9 *`pacchetto`*`.examples` . . . 35
5.10 *`pacchetto`*`.init` e *`pacchetto`*`.default` . . . 35
5.11 `install` . . . 35
5.12 *`pacchetto`*`.info` . . . 35
5.13 *`pacchetto`*`.links` . . . 36
5.14 `{`*`pacchetto`*`.,source/}lintian-overrides` . . . 36
5.15 `manpage.*` . . . 36
5.15.1 `manpage.1.ex` . . . 36
5.15.2 `manpage.sgml.ex` . . . 37
5.15.3 `manpage.xml.ex` . . . 37
5.16 *`pacchetto`*`.manpages` . . . 37
5.17 `NEWS` . . . 37
5.18 `{pre,post}{inst,rm}` . . . 38
5.19 *`package`*`.symbols` . . . 38
5.20 `TODO` . . . 38
5.21 `watch` . . . 38
5.22 `source/format` . . . 39
5.23 `source/local-options` . . . 39
5.24 `source/options` . . . 40
5.25 `patches/*` . . . 40

6 Costruzione del pacchetto 41

6.1 (ri)Creazione completa . . . 41
6.2 Auto-costruzione . . . 42
6.3 Il comando **debuild** . . . 43
6.4 Il pacchetto `pbuilder` . . . 43
6.5 Il comando **git-buildpackage** ed altri simili . . . 45
6.6 Ricostruzione veloce . . . 45
6.7 Struttura gerarchica del comando . . . 46

7 Controllare il pacchetto per errori **47**

7.1 Modifiche sospette . . . 47
7.2 Verifica dell'installazione di un pacchetto . . . 47
7.3 Verifica degli script del manutentore di un pacchetto . . . 47
7.4 Utilizzare `lintian` . . . 48
7.5 Il comando **debc** . . . 49
7.6 Il comando **debdiff** . . . 49
7.7 Il comando **interdiff** . . . 49
7.8 Il comando **mc** . . . 49

8 Aggiornamento del pacchetto **50**

8.1 Nuova revisione Debian . . . 50
8.2 Controllo della nuova distribuzione . . . 51
8.3 Nuova distribuzione . . . 51
8.4 Aggiornare lo stile di pacchettizzazione . . . 52
8.5 Conversione UTF-8 . . . 53
8.6 Note per l'aggiornamento dei pacchetti . . . 53

9 Caricamento del pacchetto **54**

9.1 Caricamento nell'archivio Debian . . . 54
9.2 Includere `orig.tar.gz` per il caricamento . . . 55
9.3 Aggiornamenti scartati . . . 55

A Pacchettizzazione avanzata **56**

A.1 Librerie condivise . . . 56
A.2 Gestire `debian/`*`package`*`.symbols` . . . 57
A.3 Multiarch . . . 58
A.4 Costruzione del pacchetto della libreria condivisa . . . 59
A.5 Pacchetto nativo Debian . . . 60

Capitolo 1

Partire nel modo giusto

Questo documento cerca di descrivere la costruzione di un pacchetto Debian GNU/Linux, sia per un normale utente Debian che per un aspirante sviluppatore. Utilizzando un linguaggio immediato, non tecnico e con l'ausilio di esempi concreti. C'è un detto latino che dice *Longum iter est per preaecepta, breve et efficax per exempla!* (La via è lunga usando la teoria, ma breve ed efficiente con gli esempi!).

È disponibile la riscrittura di questo tutorial, con contenuti aggiornati e con esempi più pratici, denominato Guide for Debian Maintainers (https://www.debian.org/doc/devel-manuals#debmake-doc) . Si prega di utilizzare il nuovo tutorial come documento primario.

Questo documento sarà disponibile fino alle release `Buster` di Debian, dato che è stato tradotto in molte lingue. Dato che i contenuti non sono aggiornati, questo documento verrà eliminato nelle release successive. [1]

Una delle cose che rende Debian una delle distribuzioni più importanti, è il suo sistema di pacchettizzazione. Sebbene ci sia una vasta quantità di software disponibile sotto forma di pacchetto Debian, qualche volta è necessario installare del software per il quale non è stato ancora creato il pacchetto. Si potrebbe pensare che creare dei pacchetti sia un compito molto difficile. In effetti, se si è alle prime armi con GNU/Linux è dura, ma se si ha esperienza non si può non leggere questo documento :-) Servirà conoscere dei rudimenti di programmazione Unix, ma di sicuro non sarà necessario essere un mago della programmazione. [2]

Una cosa però è certa: per creare in maniera corretta e manutenere dei pacchetti Debian serviranno svariate ore di lavoro. Per far funzionare il nostro sistema, i maintainer devono stare attenti a non commettere errori, essere scrupolosi e tecnicamente competenti.

Se è necessario qualche aiuto sulla pacchettizzazione, si consiglia la lettura di Sezione 1.4.

Le versioni aggiornate di questo documento dovrebbero sempre essere disponibile online all'indirizzo http://www.debian.org/-doc/maint-guide/ e nel pacchetto `maint-guida`. Le traduzioni saranno disponibili in pacchetti come `maint-guide-es` . Si noti che questa documentazione può essere leggermente obsoleta.

Poiché si tratta di un tutorial, si è scelto di spiegare dettagliatamente ogni passo per alcuni argomenti importanti. Alcuni passi possono apparire irrilevanti per alcuni di voi. Si prega di essere pazienti. Si sono anche volutamente evitati alcuni casi particolari fornendo solo dei riferimenti, per mantenere questo semplice documento.

1.1 Dinamiche sociali di Debian

Qui sono presenti alcune osservazioni delle dinamiche sociali di Debian, nella speranza che siano di aiuto per le interazioni con Debian:

[1] Nel documento si suppone l'utilizzo di un sistema `jessie` o superiore. Se si intende seguire questo documento per un vecchio sistema (incluse le vecchie versioni di Ubuntu), bisognerebbe almeno installare delle versioni backport (n.d.t. versioni dei programmi presenti nei pacchetti di testing e unstable, compilate per poter funzionare in un sistema stable) di `dpkg` e `debhelper`.

[2] Si possono avere più informazioni sulla gestione base di un sistema Debian dal documento Debian Reference (http://www.debian.org/doc/manuals/debian-reference/) . Questo documento contiene anche alcune riferimenti utili per approfondire la programmazioni Unix.

- Siamo tutti volontari.
  - Non si può imporre agli altri cosa fare.
  - Si dovrebbe essere motivati a fare le cose da soli.
- La coooperazione amichevole è la forza motrice.
  - Il vostro contributo non deve affaticare gli altri.
  - Il vostro contributo è prezioso solo quando gli altri lo apprezzano.
- Debian non è la vostra scuola in cui si ottiene in automatico l'attenzione degli insegnanti.
  - Dovreste essere in grado di imparare molte cose da soli.
  - L'attenzione degli altri volontari è una risorsa molto scarsa.
- Debian è in costante miglioramento.
  - Ci si aspetta che vengano creati pacchetti di alta qualità.
  - Si consiglia di adattarsi al cambiamento.

Ci sono molti tipi di persone, che interagiscono nel mondo Debian, con ruoli diversi:

- **autore originale (upstream author)**: La persona che ha iniziato lo sviluppo del programma.
- **responsabile del programma (upstream maintainer)**: La persona che attualmente sviluppa il programma.
- **responsabile del pacchetto (maintainer)**: La persona che ha creato il pacchetto Debian del programma.
- **sponsor**: La persona che aiuta i responsabili dei pacchetti a verificarne la correttezza dei contenuti ed a depositarli nell'archivio Debian ufficiale.
- **mentore**: La persona che aiuta i responsabili dei pacchetti alle prime armi nelle operazioni di creazione del pacchetto, ecc.
- **sviluppatore Debian (Debian Developer)** (DD): membro del progetto Debian con i permessi di depositare i pacchetti all'interno dell'archivio Debian ufficiale.
- **responsabile Debian (Debian Maintainer)** (DM): persona con permessi di accesso limitati all'archivio ufficiale dei pacchetti di Debian.

Si noti che non è possibile diventare uno **sviluppatore Debian** (DD) ufficiale dal giorno alla notte, poiché questo richiede più che delle semplici conoscenze tecniche. Ma non ci si deve lasciare scoraggiare da questo. Se il lavoro che si è fatto è utile a qualcun altro, si può sempre depositare il proprio pacchetto sia come **maintainer** attraverso uno **sponsor** che come **maintainer Debian**.

Si noti che non è necessario creare un nuovo pacchetto per diventare uno sviluppatore Debian ufficiale. Lo si può diventare anche semplicemente contribuendo alla manutenzione di pacchetti già esistenti. Ci sono molti pacchetti che aspettano solo dei bravi responsabili (vedere Sezione 2.2).

Dal momento che in questo documento si trattano solo gli aspetti tecnici della pacchettizzazione, si prega di fare riferimento a quanto segue per conoscere come relazionarsi con Debian e come partecipare:

- Debian: 17 anni di Software Libero, "do-ocracy", e la democrazia (http://upsilon.cc/~zack/talks/2011/20110321-taipei.pdf) (Slides introduttive)
- Come aiutare Debian? (http://www.debian.org/intro/help) (Ufficiale)
- Debian GNU/Linux FAQ, capitolo 13 - "Contribuire al Progetto Debian" (https://www.debian.org/doc/manuals/debian-faq/-contributing.en.html) (Semi-ufficiale)
- Il Wiki di Debian, HelpDebian (http://wiki.debian.org/HelpDebian) (supplementare)
- Sito del New Member Debian (https://nm.debian.org/) (ufficiale)
- Debian Mentors FAQ (http://wiki.debian.org/DebianMentorsFaq) (supplementare)

1.2 Programmi necessari per lo sviluppo

Prima di iniziare, bisogna assicurarsi di avere installato correttamente alcuni pacchetti aggiuntivi, necessari per lo sviluppo. Da notare che la lista non contiene nessun pacchetto etichettato come `essenziale` o `richiesto` - ci aspettiamo che siano già installati nel sistema.

I seguenti pacchetti fanno parte dell'installazione standard di Debian, per cui probabilmente sono già presenti nel sistema (insieme ai pacchetti aggiuntivi dai quali dipendono). Si può effettuare un controllo con `aptitude show` *`pacchetto`* o con `dpkg -s` *`pacchetto`*.

Il pacchetto più importante da installare in un sistema in cui si ha intenzione di sviluppare è `build-essential`. Questo *includerà* altri pacchetti necessari per avere un ambiente di base per la compilazione dei programmi.

Per alcuni tipi di pacchetti questo è tutto quello che serve, ci sono però una serie di pacchetti che, pur non essendo essenziali per lo sviluppo, vengono in aiuto allo sviluppatore o possono essere richiesti dal pacchetto su cui si lavora:

- `autoconf`, `automake` e `autotools-dev` - diversi programmi recenti usano script di configurazione e `Makefile` pre-processati con l'aiuto di programmi come questi. (vedere `info autoconf`, `info automake`) `autotools-dev` tiene aggiornate le versioni di alcuni file di automazione e contiene la documentazione che spiega il modo migliore per utilizzare questi file.
- `debhelper` e `dh-make` - `dh-make` è necessario per creare lo scheletro del pacchetto, utilizza alcuni strumenti di `debhelper` per creare i pacchetti. Non sono essenziali per la creazione di pacchetti, ma sono *fortemente* consigliati per i nuovi maintainer. Questo rende l'intero processo molto più semplice da iniziare e controllare successivamente. (vedere dh_make(8), debhelper(1).) [3]

  Il nuovo `debmake` può essere utilizzato in alternativa a `dh-make`. Ha più funzionalità e comprende la documentazione HTML con esempi dettagliati di pacchettizzazione, presenti in `debmake-doc`.
- `devscripts` - questo pacchetto contiene alcuni pratici script che possono essere utili ai maintainer, anche se non sono strettamente necessari per la creazione dei pacchetti. I pacchetti consigliati o suggeriti da questo pacchetto andrebbero presi in considerazione. (vedere `/usr/share/doc/devscripts/README.gz`.)
- `fakeroot` - questo programma permette di fingere di essere root, dato che è necessario per l'esecuzione di alcune parti del processo di creazione del pacchetto. (vedere fakeroot(1).)
- `file` - questo semplice programma determina la tipologia di un file. (vedere file(1).)
- `gfortran` - il compilatore GNU Fortran 95, necessario se il programma è scritto in Fortran. (vedere gfortran(1).)
- `git` - questo pacchetto contiene un famoso sistema per il controllo delle versioni, progettato per gestire progetti molto grandi con velocità ed efficienza; è utilizzato da molti progetti open source, tra cui il kernel Linux. (Vedere git(1), Manuale di git (`/usr/share/doc/git-doc/index.html`).)
- `gnupg` - questo programma consente di *firmare* elettronicamente i pacchetti. Questo è importante soprattutto se si vuole distribuirli ad altre persone, e verrà sicuramente fatto quando un pacchetto sarà pronto per essere incluso nella distribuzione Debian. (vedere gpg(1).)
- `gpc` - il compilatore GNU Pascal, necessario se il programma è scritto in Pascal. Un sostituto degno di nota è `fp-compiler`, il Compilatore Free Pascal. (vedere gpc(1), ppc386(1).)
- `lintian` - questo è l'analizzatore dei pacchetti Debian, una volta costruito il pacchetto, permette di scoprire gli errori più comuni, cercando di spiegarli. (vedere lintian(1), Lintian User's Manual (https://lintian.debian.org/manual/index.html) .)
- `patch` - questo utile programma usa un file contenente una serie di differenze (prodotta dal programma diff) e le applica al file originale, per produrre una versione modificata. (vedere patch(1).)
- `patchutils` - questo pacchetto contiene dei programmi che lavorano con le patch, come **lsdiff**, **interdiff** e **filterdiff**.
- `pbuilder` - questo pacchetto contiene i programmi che vengono usati per creare e manutenere un ambiente **chroot**. Creare pacchetti Debian nell'ambiente **chroot** permette di verificare le dipendenze appropriate ed evitare bug di tipo FTBFS (Fails To Build From Source, non compila da sorgente). (vedere pbuilder(8) e pdebuild(1))

[3] Ci sono un paio di pacchetti specializzati ma simili, come `dh-make-perl`, `dh-make-php`, ecc.

- `perl` - Perl è uno dei linguaggi di scripting interpretato più utilizzati sugli odierni sistemi Unix e derivati, spesso definito come il coltellino svizzero di Unix. (vedere perl(1).)
- `python` - Python è un altro linguaggio di scripting interpretato molto utilizzato sui sistemi Debian, combina una notevole potenza con una sintassi molto chiara. (vedere python(1).)
- `quilt` - questo pacchetto aiuta a gestire una gran numero di patch, tenendo traccia dei cambiamenti apportati. Le patch sono organizzate in maniera logica come una pila, è possibile applicare(=push) le modifiche apportate dalla patch, oppure annullarle(=pop), semplicemente muovendosi attraverso la pila. (vedere quilt(1) e `/usr/share/doc/quilt/quilt.pdf.gz`.)
- `xutils-dev` - alcuni programmi, generalmente quelli fatti per X11, usano questi strumenti per generare i `Makefile` da una serie di funzioni macro. (vedere imake(1), xmkmf(1).)

Le brevi note elencate qui sopra servono solo ad accennare lo scopo di ogni pacchetto. Prima di continuare, è opportuno leggere la documentazione di ogni programma rilevante, compresi quelli installati per via delle dipendenze del pacchetto come **make**, almeno per un utilizzo di base. Può sembrare molto pesante farlo adesso, ma in seguito ci si renderà conto che sarà stato *utilissimo*. Se si hanno dubbi specifici potrebbe essere utile rileggere i documenti di cui sopra.

1.3 Documentazione necessaria per lo sviluppo

Segue una serie di documenti *molto importanti* che è consigliabile leggere insieme a questo documento:

- `debian-policy` - il manuale delle policy Debian (http://www.debian.org/doc/devel-manuals#policy) comprende le spiegazioni riguardanti la struttura e il contenuto dell'archivio Debian, numerose problematiche inerenti la progettazione del sistema operativo, lo Standard della Gerarchia del Filesystem (http://www.debian.org/doc/packaging-manuals/fhs/fhs-3.0.html) (ndr. Filesystem Hierarchy Standard), (che indica la posizione prevista per ogni file e directory), ecc. In questo contesto, l'argomento più importante è la descrizione dei requisiti che ogni pacchetto deve soddisfare per essere incluso nella distribuzione. (vedere le copie locali di `/usr/share/doc/debian-policy/policy.pdf.gz` and `/usr/share/doc/debian-policy/fhs/fhs-3.0.pdf.gz`.)
- `developers-reference` - la Debian Developer's Reference (http://www.debian.org/doc/devel-manuals#devref) descrive tutte le questioni non prettamente legate alle tecniche di pacchettizzazione, come la struttura dell'archivio, come rinominare, abbandonare o adottare un pacchetto, come fare gli NMUs, come gestire i bug, suggerimenti pratici di pacchettizzazione, quando e dove fare i caricamenti, ecc. (Si veda la copia locale di `/usr/share/doc/developers-reference/developers-reference.pdf`.)

Segue una serie di documenti *importanti* che è consigliabile leggere insieme a questo documento:

- Autotools Tutorial (http://www.lrde.epita.fr/~adl/autotools.html) contiene un ottimo tutorial su the GNU Build System known as the GNU Autotools i cui componenti più importanti sono Autoconf, Automake, Libtool, e gettext.
- `gnu-standards` - questo pacchetto contiene due documenti provenienti dal progetto GNU: GNU Coding Standards (http://www.gnu prep/standards/html_node/index.html) , and Information for Maintainers of GNU Software (http://www.gnu.org/prep/maintain/-html_node/index.html) . Nonostante Debian non necessiti del loro utilizzo, sono comunque utili come linee guida e buona prassi. (vedere le copie locali di `/usr/share/doc/gnu-standards/standards.pdf.gz` e `/usr/share/doc/gnu-standards/maintain.pdf.gz`.)

Se questo documento contraddice uno dei documenti cui sopra, si considerino corretti quest'ultimi. Si prega di inviare una segnalazione di bug relativa al pacchetto `maint-guide` usando **reportbug**.

Segue una serie di tutorial alternativi che si possono leggere insieme a questo documento:

- Debian Packaging Tutorial (http://www.debian.org/doc/packaging-manuals/packaging-tutorial/packaging-tutorial)

1.4 Dove chiedere aiuto

Prima di decidere di fare una domanda in qualche luogo pubblico, si prega di leggere la documentazione:

- i file in `/usr/share/doc/`*`pacchetto`* per tutti i pacchetti pertinenti
- il contenuto di **`man`***`comando`* per tutti i comandi pertinenti
- il contenuto di **`info`***`comando`* per tutti i comandi pertinenti
- il contenuto dell'archivio della listadebian-mentors@lists.debian.org (http://lists.debian.org/debian-mentors/)
- il contenuto dell'archivio della listadebian-devel@lists.debian.org (http://lists.debian.org/debian-devel/)

È possibile utilizzare i motori di ricerca web più efficacemente includendo stringhe di ricerca come `sito:lists.debian.org` per limitare le ricerca ad un determinato dominio.

Creare un piccolo pacchetto di test è un ottimo metodo per imparare i dettagli della pacchettizzazione. Analizzare dei pacchetti già esistenti e ben manutenuti, è invece, il metodo migliore per capire come creano i pacchetti le altre persone.

Se si hanno ancora domande sulla pacchettizzazione alle quale non si trova risposta nella documentazione disponibile e le risorse web, si può chiedere aiuto nei seguenti luoghi:

- debian-mentors@lists.debian.org mailing list (http://lists.debian.org/debian-mentors/) . (Questa è la mailing list per i principianti.)
- debian-devel@lists.debian.org mailing list (http://lists.debian.org/debian-devel/) . (Questa è la mailing list per gli esperti.)
- su IRC (http://www.debian.org/support#irc) in canali come `#debian-mentors`.
- Team concentrati su uno specifico insieme di pacchetti. (Lista completa su https://wiki.debian.org/Teams (https://wiki.debian.org/-Teams))
- Mailing list specifiche per lingua come debian-devel-{french,italian,portuguese,spanish}@lists.debian.org o debian-devel@debian.or.j (Elenco completo su https://lists.debian.org/devel.html (https://lists.debian.org/devel.html) e https://lists.debian.org/users.html (https://lists.debian.org/users.html))

Gli sviluppatori di Debian più esperti saranno lieti di aiutare, se viene chiesto correttamente e dopo aver fatto gli sforzi necessari.

Quando si ricevono delle segnalazioni di bug (sì, proprio quelle!), si dovrà approfondire l'utilizzo del Sistema di tracciamento dei bug di Debian (http://www.debian.org/Bugs/) e leggere la relativa documentazione, per essere in grado di gestire le segnalazioni in maniera efficiente. È vivamente consigliato leggere Debian Developer's Reference, 5.8. "Handling bugs" (http://www.debian.org/doc/manuals/developers-reference/pkgs.html#bug-handling) .

Anche se tutto è andato per il meglio, è arrivato il momento di pregare. Perché? Perché in poche ore (o giorni) utenti da tutto il mondo cominceranno ad usare il vostro pacchetto, e se si è commesso qualche errore grave, la propria email sarà inondata da messaggi di molti utenti Debian incavolati⋯Si scherza ovviamente. :-)

Ci si deve rilassare ed essere pronti per le segnalazioni di bug, perché c'è molto lavoro prima che un pacchetto sia completamente conforme alle policy di Debian (ancora una volta, si legga la *vera documentazione* per i dettagli). In bocca al lupo!

Capitolo 2

Primi passi

Iniziamo a creare un pacchetto (o, meglio ancora, adottiamone uno già esistente).

2.1 Flusso di lavoro per la costruzione dei pacchetti Debian

Se si sta facendo un pacchetto Debian con un programma, il flusso di lavoro tipico per la costruzione di pacchetti Debian comporta la generazione di diversi file indicati in modo specifico per ogni passo come segue:

- Procuriamoci una copia del programma, di solito in formato tar compresso.
  - *pacchetto-versione*`.tar.gz`
- Aggiungiamo le modifiche specifiche per il pacchetto Debian del programma, nella directory `debian`, e creiamo un archivio sorgente non nativo (ovvero con l'insieme di file di input utilizzati per la creazione del pacchetto) in formato `3.0 (quilt)`.
  - *pacchetto_versione*`.orig.tar.gz`
  - *pacchetto_versione-revisione*`.debian.tar.gz`[1]
  - *pacchetto_versione-revisione*`.dsc`
- Costruiamo i pacchetti binari Debian, che sono normali pacchetti installabili nel formato `.deb` (o nel formato `.udeb`, usato dall'installer Debian) dal sorgente del pacchetto Debian.
  - *pacchetto_versione-revisione_arch*`.deb`

Si prega di notare che il carattere di separazione tra *pacchetto* e *versione* è stato modificato da `-` (trattino) nel nome dell'archivio, a _ (trattino basso) nel nome del pacchetto Debian.

Nel file di nomi di cui sopra, sostituire la parte relativa al *pacchetto* con il **nome del pacchetto**, la *versione* con la **versione originale**, la *revisione* con la **revisione Debian**, e l' *architettura* con l'**architettura del pacchetto**, come definito nel manuale delle Policy di Debian. [2]

Ogni passo di questo schema è spiegato con esempi dettagliati nelle sezioni successive.

[1] Per gli archivi non nativi nel vecchio formato `1.0`, viene utilizzato il nome *pacchetto_versione-revisione*`.diff.gz`.

[2] Vedere 5.6.1 "Sorgente" (http://www.debian.org/doc/debian-policy/ch-controlfields.html#s-f-Source) , 5.6.7 "Pacchetto" (http://www.debian.org/doc/debian-policy/ch-controlfields.html#s-f-Package) , e 5.6.12 "Versione" (http://www.debian.org/doc/debian-policy/ch-controlfields.html#s-f-Version) . L'**architettura del pacchetto** è conforme al manuale delle Policy di Debian: 5.6.8 "Architettura" (http://www.debian.org/doc/debian-policy/ch-controlfields.html#s-f-Architecture) ed è assegnata automaticamente al processo di creazione del pacchetto.

2.2 Scegliamo il programma

Probabilmente avete già scelto il pacchetto da creare. La prima cosa da fare è controllare se il pacchetto è già presente negli archivi della distribuzione, utilizzando i seguenti strumenti:

- il comando **aptitude**
- la pagina web Pacchetti Debian (http://www.debian.org/distrib/packages)
- the Debian Package Tracker (https://tracker.debian.org/) web page

Se il pacchetto esiste già, bene, basta installarlo! :-) Se dovesse essere **orfano** (cioè, se il maintainer è Debian QA Group (http://qa.debian.org/)), lo si può prendere in carico se è ancora disponibile. Si può adottare anche un pacchetto per il quale il suo manutentore ha presentato una richiesta di adozione (**RFA**).[3]

Ci sono diverse risorse per controllare lo stato di appartenenza dei pacchetti:

- Il comando **wnpp-alert** dal pacchetto `devscripts`
- Work-Needing and Prospective Packages (http://www.debian.org/devel/wnpp/)
- Registri di Debian delle segnalazioni dei bug: bug nello pseudo-pacchetto `wnpp` in `unstable` (http://bugs.debian.org/wnpp)
- Pacchetti Debian che hanno bisogno d'amore (http://wnpp.debian.net/)
- Sfogliare i bug di `wnpp` basati su debtags (http://wnpp-by-tags.debian.net/)

Come nota a margine, è importante sottolineare che Debian ha già i pacchetti per la maggior parte dei programmi e che il numero di pacchetti presenti nell'archivio Debian è molto più grande di quello dei contribuenti con permessi di caricamento. Pertanto, i contributi sui pacchetti già presenti in archivio sono molto più apprezzati dagli altri sviluppatori (ed è molto più probabile che si ottenga una sponsorizzazione). [4]. Si può contribuire in diversi modi:

- adottando dei pacchetti orfani, ma ancora attivamente utilizzati
- entrando a far parte dei team di pacchettizzazione (http://wiki.debian.org/Teams)
- aiutando nella risoluzione dei bug di pacchetti molto popolari
- preparando QA o caricando NMU (http://www.debian.org/doc/developers-reference/pkgs.html#nmu-qa-upload)

Appena ci si sente in grado di adottare un pacchetto, bisogna scaricare i sorgenti (con qualcosa tipo `apt-get source` *`pacchetto`*) ed esaminarli. Questo documento purtroppo non include tutte le informazioni che riguardano l'adozione dei pacchetti. Fortunatamente non sarà difficile capire come funziona il pacchetto dal momento che qualcuno avrà già effettuato la configurazione iniziale. Continua comunque a leggere, molti dei suggerimenti qui di seguito saranno utili per il nostro scopo.

Se il pacchetto è nuovo, e si pensa che sarebbe bello entrare a far parte di Debian, ecco come procedere:

- Prima di tutto bisogna capire se il programma funziona in modo corretto, e averlo provato per almeno un po' di tempo e dimostrarne l'utilità.
- Bisogna controllare nella lista dei pacchetti sui quali si lavora (http://www.debian.org/devel/wnpp/being_packaged) che nessun altro stia lavorando sullo stesso pacchetto. Se nessuno ci sta lavorando, si può segnalare un bug di tipo ITP (Intent To Package) allo pseudo-pacchetto `wnpp` usando il programma **reportbug**. Se qualcuno ci sta lavorando e si ritiene necessario si potrebbe contattare il maintainer. Altrimenti —si potrebbe trovare un altro programma interessante che è non manutenuto.
- Il programma **deve avere una licenza**.

[3] Per maggiori informazioni, consulta la Guida di riferimento per lo sviluppatore Debian 5.9.5. "Adottare un pacchetto" (http://www.debian.org/doc/devel-manuals#devref) .

[4] Detto questo, ovviamente ci sarà sempre qualche nuovo programma che vale la pena pacchettizzare.

  - I programmi nella sezione `main`, **devono essere completamente compatibili con le Linee Guida per il Software Libero Debian (DFSG, Debian Free Software Guidelines)** (vedere DSFG (http://www.debian.org/social_contract#guidelines)) e **non devono richiede nessun pacchetto che non sia presente nella sezione `main`**, per la compilazione o l'esecuzione. Questo è il caso preferito.
  - I programmi nella sezione `contrib`, devono essere conformi alle DFSG, ma potrebbero richiedere, per la compilazione o l'esecuzione, un pacchetto che non è presente nella sezione `main`.
  - I programmi nella sezione `non-free`, possono non essere conformi alle DFSG, ma **devono poter essere distribuibili**.
  - Se non si è sicuri su quale sezione il pacchetto dovrebbe essere incluso, si può mandare il testo della licenza alla mailing list debian-legal@lists.debian.org (http://lists.debian.org/debian-legal/) e chiedere consigli.

- Il programma **non** dovrebbe introdurre problemi si sicurezza e di manutenzione al il sistema Debian.
  - Il programma dovrebbe essere ben documentato e il suo codice facilmente comprensibile (ad es. non offuscato).
  - Si dovrebbe contattare l'autore o gli autori del programma per verificare che siano d'accordo con la sua pacchettizzazione. È importante essere in grado di consultarsi con l'autore/i sul programma nel caso di problemi specifici del programma, per questo è meglio non provare a pacchettizzare programmi non più manutenuti.
  - Il programma **non** dovrebbe certamente girare come setuid root, o meglio, non dovrebbe per niente richiedere di essere setuid o setgid.
  - Il programma non dovrebbe essere un daemon, o essere installato nelle directory `*/sbin`, o aprire una porta come root.

Ovviamente queste sono solo misure di sicurezza, fatte per salvarti dall'ira degli utenti se si commette qualche errore in qualche daemon setuid···Una volta acquisita esperienza nella pacchettizzazione, sarai in grado di creare pure quel tipo di pacchetti.

Visto che si è alle prime armi come maintainer, si consiglia di acquisire un po' d'esperienza creando dei pacchetti semplici cercando di evitare quelli più complicati.

- Pacchetti semplici
  - pacchetto binario singolo, arch = all (collezione di dati, come le immagini di sfondo)
  - pacchetto binario singolo, arch = all (eseguibili scritti in un linguaggio interpretato come lo shell POSIX)
- Pacchetti di media difficoltà
  - pacchetto binario singolo, arch = any (binari ELF eseguibili compilati da linguaggi come C e C++)
  - pacchetti binari multipli, arch = any + all (pacchetti per binari ELF eseguibili + documentazione)
  - sorgente originale in un formato diverso da `tar.gz` o `tar.bz2`
  - l'archivio dei sorgenti originale ha contenuti non distribuibili
- Pacchetti complessi
  - pacchetto di un modulo di un interprete utilizzato da altri pacchetti
  - libreria ELF generica utilizzata da altri pacchetti
  - pacchetti binari multipli che includono un pacchetto di una libreria ELF
  - pacchetto con molteplici sorgenti originali
  - pacchetti di moduli del kernel
  - pacchetti di patch del kernel
  - ogni pacchetto con degli script di manutenzione non banali

Creare pacchetti complessi non è troppo difficile, ma richiede un po' più di conoscenza. Si dovrebbe cercare una guida specifica per ogni caratteristica complessa. Ad esempio, alcuni linguaggi hanno dei loro documenti con le loro linee guida:

- Perl policy (http://www.debian.org/doc/packaging-manuals/perl-policy/)
- Python policy (http://www.debian.org/doc/packaging-manuals/python-policy/)

- Java policy (http://www.debian.org/doc/packaging-manuals/java-policy/)

C'è un altro vecchio detto latino che dice: *fabricando fit faber* (la pratica rende perfetti). Si tratta di una pratica *vivamente* consigliata, sperimentate tutte le fasi di pacchettizzazione Debian con un pacchetto semplice durante la lettura di questo tutorial. Un archivio compresso banale come `hello-sh-1.0.tar.gz` creato come segue, può offrire un buon punto di partenza:[5]

```
$ mkdir -p hello-sh/hello-sh-1.0; cd hello-sh/hello-sh-1.0
$ cat > hello <<EOF
#!/bin/sh
# (C) 2011 Foo Bar, GPL2+
echo "Hello!"
EOF
$ chmod 755 hello
$ cd ..
$ tar -cvzf hello-sh-1.0.tar.gz hello-sh-1.0
```

2.3 Prendere il programma e provarlo

La prima cosa da fare è trovare e scaricare il codice sorgente originale del programma. Supponendo che si è recuperato il file dal sito web dell'autore. Generalmente il codice sorgente dei programmi liberi per Unix e derivati sono in formato **tar**+**gzip** con estensione `.tar.gz`, oppure in formato **tar**+**bzip2** con estensione `.tar.bz2`. Di solito, questi file, contengono la sottodirectory dal nome *`pacchetto-versione`* con tutti i sorgenti.

Se è presente un sistema di controllo di versione (VCS) come Git, Subversion o CVS, è possibile scaricare l'ultima versione del codice sorgente con `git clone`, `svn co`, o `cvs co` e comprimerlo in formato **tar**+**gzip** utilizzando l'opzione `--exclude-vcs`.

Se il codice sorgente è in qualche altro formato di archiviazione (per esempio, con estensione `.Z` o `.zip`[6]), scompattarlo con i programmi appropriati, e ricomprimerlo.

Se il sorgente del programma viene fornito con alcuni contenuti che non sono conformi con il DFSG, si dovrebbe scompattarlo, rimuovere questi contenuti e ricomprimerlo con una versione modificata dei sorgenti originali contenenti `dfsg` .

A titolo di esempio, verrà utilizzato il programma **gentoo**, un gestore file grafico basato su GTK+.[7]

È buona regola creare una sottodirectory nella directory home e nominarla `debian` o `deb` o qualsiasi altro nome appropriato (ad es. in questo caso `~/gentoo` andrebbe più che bene). Scaricare l'archivio e scompattarlo (con il comando `tar xzf gentoo-0.9.12.tar.gz`). Bisogna assicurarsi che non ci siano errori, per quanto in apparenza *irrilevanti*, perché potrebbero causare problemi nell'estrazione dell'archivio sul sistema di altre persone, alcuni strumenti di estrazione a volte ignorano queste anomalie. Nella console dovrebbe esserci quanto segue:

```
$ mkdir ~/gentoo ; cd ~/gentoo
$ wget http://www.example.org/gentoo-0.9.12.tar.gz
$ tar xvzf gentoo-0.9.12.tar.gz
$ ls -F
gentoo-0.9.12/
gentoo-0.9.12.tar.gz
```

A questo punto si avrà un'altra sottodirectory, dal nome `gentoo-0.9.12`. Spostarsi in questa directory e leggere *attentamente* la documentazione fornita. Di solito si avranno dei file come `README*`, `INSTALL*`, `*.lsm` o `*.html`. È necessario trovare istruzioni su come compilare e installare correttamente il programma (si potrebbe supporre di installare il programma nella directory `/usr/local/bin`, ma questo non è il comportamento corretto, tratteremo l'argomento più avanti Sezione 3.3).

Si dovrebbe iniziare la pacchettizzazione con la directory dei sorgenti completamente ripulita, o semplicemente partendo da una nuova estrazione dall'archivio dei sorgenti.

[5] Non ci si preoccupi della mancanza dei `Makefile`. È possibile installare il programma **hello** semplicemente usando il comando **debhelper** come in Sezione 5.11, oppure modificando il sorgente originale aggiungendo un nuovo `Makefile` con la destinazione dell' `install` come in Capitolo 3 .

[6] Si può utilizzare il comando **file** per scoprire il formato di archiviazione.

[7] Il programma in questione è già stato pacchettizzato. La versione corrente (http://packages.qa.debian.org/g/gentoo.html) usa gli Autotools e la sua costruzione e struttura è molto sostanzialmente differente dagli esempi seguenti, che sono basati sulla versione 0.9.12.

2.4 Sistemi di compilazione semplici

I programmi più semplici sono dotati di un file `Makefile`, e possono essere compilati semplicemente con il comando `make`.[8] Alcuni supportano `make check`, che esegue dei controlli automatici. Per installare il programma nella directory di destinazione, di solito basta eseguire `make install`.

Adesso si provi a compilare ed eseguire il programma, assicurandosi che funzioni correttamente e che niente sia andato storto durante l'installazione o l'esecuzione.

Di solito, per ripulire la directory di compilazione, si usa il comando `make clean` (o meglio ancora `make distclean`). Talvolta c'è anche il comando `make uninstall` che serve a rimuovere tutti i file installati.

2.5 Sistemi di compilazione portabili più utilizzati

Molti programmi liberi sono scritti utilizzando i linguaggi di programmazione C e C++. Molti di questi utilizzano Autotools o CMake per essere portabili su diverse piattaforme. Questi strumenti vengono utilizzati per generare il `Makefile` e altri file sorgenti necessari. Dopo di questo, i programmi vengono compilati utilizzando il solito `make; make install`.

Autotools è il sistema di compilazione della GNU, che comprendere Autoconf, Automake, Libtool, e gettext. Per capire meglio ciò che avviene, si possono leggere i seguenti file sorgenti: `configure.ac`, `Makefile.am`, e `Makefile.in`. [9]

Il primo passo del flusso di lavoro degli Autotools consiste solitamente nell'esecuzione del comando `autoreconf -i -f` per i sorgenti che verranno successivamente distribuiti insieme ai file generati.

```
configure.ac-----+-> autoreconf -+-> configure
Makefile.am -----+        |      +-> Makefile.in
src/Makefile.am -+        |      +-> src/Makefile.in
                          |      +-> config.h.in
                      automake
                      aclocal
                      aclocal.m4
                      autoheader
```

La modifica dei file `configure.ac` e `Makefile.am` richiede una minima conoscenza di **autoconf** e **automake**. Vedere `info autoconf` e `info automake`.

Il passo successivo da compiere con Autotools, di solito, consiste nel procurarsi il sorgente del programma e nel compilarlo nella directory ***binary*** tramite l'esecuzione dei comandi `./configure && make` all'interno della directory dei sorgenti stessi.

```
Makefile.in -----+               +-> Makefile -----+-> make -> binary
src/Makefile.in -+-> ./configure -+-> src/Makefile -+
config.h.in -----+               +-> config.h -----+
                 |
  config.status -+
  config.guess --+
```

Si possono modificare molte cose nel file `Makefile`, come la posizione predefinita di installazione dei file utilizzando l'opzione **./configure --prefix=/usr**.

Nonostante non sia richiesto, l'aggiornamento di `configure` e degli altri file con `autoreconf -i -f` può migliorare la compatibilità del sorgente. [10]

CMake è un alternativo sistema di compilazione. Per conoscerlo meglio si può leggere il file `CMakeLists.txt`.

[8] Molti programmi moderni sono dotati di uno script `configure` che, una volta eseguito, crea un `Makefile` personalizzato per il proprio sistema.

[9] Autotools è troppo vasto per essere approfondito in questo piccolo tutorial. Questa sezione ha lo scopo di fornire solo i concetti base ed i riferimenti. Assicurarsi di leggere il tutorial Autotools (http://www.lrde.epita.fr/~adl/autotools.html) e la copia locale di `/usr/share/doc/autotools-dev/README.Debian.gz`, se si intende usarlo.

[10] Si possono automatizzare queste operazioni utilizzando il pacchetto `dh-autoreconf`. Vedere Sezione 4.4.3.

2.6 Nome e versione del pacchetto

Se il sorgente del programma originale è nominato `gentoo-0.9.12.tar.gz`, si può usare `gentoo` come **nome pacchetto** e `0.9.12` come **versione del programma originale**. Queste stringhe saranno utilizzate nel file `debian/changelog` come vedremo più avanti nel Sezione 4.3.

Sebbene questo semplice approccio il più delle volte funzioni, può essere necessario modificare il **nome pacchetto** e la **versione del programma originale** rinominando il sorgente originale in qualcosa conforme alla policy di Debian ed alle convenzioni esistenti.

È necessario scegliere il **nome del pacchetto** utilizzando solo lettere minuscole (`a-z`), cifre (`0-9`), il segno più (`+`) e il segno meno (`-`), e il punto (`.`). Il nome deve essere di almeno due caratteri, deve iniziare con un carattere alfanumerico, e non deve essere la stesso di quelli già esistenti. È consigliabile mantenere la lunghezza intorno ai 30 caratteri. [11]

Se il sorgente originale usa un nome troppo generico come `test-suite` è consigliabile rinominarlo, in modo da identificare meglio il suo contenuto e per non rischiare di sporcare il namespace. [12]

Si consiglia di scegliere un nome della **versione del programma originale** contenente solo caratteri alfanumerici (`0-9A-Za-z`), il segno più (`+`), il simbolo tilde (`~`), e il punto (`.`). Il nome deve iniziare con una cifra (`0-9`). [13] È consigliabile, se possibile, mantenere una lunghezza entro gli 8 caratteri. [14]

Se il programma originale non utilizza un normale sistema di versioning, come ad esempio `2.30.32`, ma utilizza qualche tipo di data, come `11Apr29`, un stringa con un codice casuale, o un valoro di hash di un VCS, bisogna assicurarsi di rimuoverli dalla **versione originale**. Queste informazioni possono essere registrate nel file `debian/changelog`. Se si ha bisogno di inventare una stringa di versione, bisogna utilizzare il formato `YYYYMMDD`, ad esempio `20110429` come versione originale. Questo fa in modo che **dpkg** interpreti in modo corretto le versioni successive del programma, per gli aggiornamenti. Se ci si vuole assicurare, in futuro, una transazione indolore ad un normale sistema di versioning, come `0.1`, si utilizzi il formato `0~YYMMDD`, ed esempio `0~110429`, anzichè la versione originale.

Le stringhe di versione [15] possono essere confrontate usando dpkg(1) come segue:

```
$ dpkg --compare-versions ver1 op ver2
```

Le regole per il confronto delle versioni possono essere riassunte in questo modo:

- Le stringhe sono confrontate dall'inizio alla fine (da sinistra verso destra).
- Le lettere hanno più priorità delle cifre.
- I numeri sono confrontati come interi.
- Le lettere sono confrontate in ordine di codice ASCII.
- Ci sono regole speciali per il punto (`.`), il segno più (`+`), e il carattere tilde (`~`), eccole descritte:
  `0.0` < `0.5` < `0.10` < `0.99` < `1` < `1.0~rc1` < `1.0` < `1.0+b1` < `1.0+nmu1` < `1.1` < `2.0`

Un esempio di caso intricato si ha, ad esempio, quando una è presente una pre-release RC come `gentoo-0.9.12-ReleaseCandida tar.gz` per il programma `gentoo-0.9.12.tar.gz`. In questo caso è necessario assicurarsi che l'aggiornamento funzioni correttamente, rinominando il sorgente originale `gentoo-0.9.12~rc99.tar.gz`.

[11] Il campo di default di **aptitude** del nome del pacchetto è di 30 caratteri. Oltre il 90% dei pacchetti ha un nome più piccolo di 24 caratteri.

[12] Se si segue la guida dello sviluppatore Debian 5.1. "New packages" (http://www.debian.org/doc/developers-reference/pkgs.html#newpackage) , il processo di ITP di solito dovrebbe risolvere questo tipo di problemi.

[13] Questa regola dovrebbe aiutare ad evitare confusione con i nome dei file.

[14] La lunghezza predefinita del campo "versione" di **aptitude** è di 10 caratteri. Di solito le revisioni Debian precedute da un trattino ne utilizzano 2. Per più dell'80% dei pacchetti, la lunghezza della versione dei sorgenti originali è più piccola di 10 caratteri e la revisione Debian è inferiore a 3 caratteri.

[15] La stringa della versione può essere costituita da **versione del programma originale** (*`versione`*), **revisione Debian** (*`revisione`*), oppure da **versione** (*`versione-revisione`*). Vedere Sezione 8.1 per conoscere come incrementare il numero della **revisione Debian**.

2.7 Configurare dh_make

Una delle prime cose da fare è impostare le variabili d'ambiente della shell `$DEBEMAIL` e `$DEBFULLNAME` visto che molti strumenti di gestione di Debian usano queste variabili per recuperare il nome e l'email da utilizzare nei pacchetti. [16]

```
$ cat >>~/.bashrc <<EOF
DEBEMAIL="your.email.address@example.org"
DEBFULLNAME="Firstname Lastname"
export DEBEMAIL DEBFULLNAME
EOF
$ . ~/.bashrc
```

2.8 Il primo pacchetto non nativo per Debian

I pacchetti classici per Debian sono pacchetti non-nativi, ovvero non specificamente pensati per debian, come ad esempio qualche programma di manutenzione per Debian. Se si desidera creare un pacchetto Debian non-nativo del programma con il sorgente `gentoo-0.9.12.tar.gz` si può utilizzare il programma **dh_make** come segue:

```
$ cd ~/gentoo
$ wget http://example.org/gentoo-0.9.12.tar.gz
$ tar -xvzf gentoo-0.9.12.tar.gz
$ cd gentoo-0.9.12
$ dh_make -f ../gentoo-0.9.12.tar.gz
```

Ovviamente, si deve sostituire il nome del file con il nome dell'archivio dei sorgenti originali. [17] Vedere dh_make(8) per i dettagli.

Verranno visualizzate alcune informazioni e verrà chiesto che tipo di pacchetto creare. Gentoo è un pacchetto binario singolo — crea un solo binario, e quindi un solo file `.deb` —per cui si dovrà selezionare la prima opzione (con il tasto `s`), controllare le informazioni sullo schermo e confermare la scelta con *`ENTER`*. [18]

L'esecuzione di **dh_make**, creerà una copia dell'archivio originale del programma, come nome `gentoo_0.9.12.orig.tar.gz`, nella directory superiore, per consentire, più avanti, la creazione di un pacchetto Debian sorgente non-nativo con nome `debian.tar.gz`:

```
$ cd ~/gentoo ; ls -F
gentoo-0.9.12/
gentoo-0.9.12.tar.gz
gentoo_0.9.12.orig.tar.gz
```

Si presti attenzione a due caratteristiche chiave presenti nel nome del file `gentoo_0.9.12.orig.tar.gz`:

- Il nome del pacchetto e la versione sono separati da _ (trattino basso).
- La stringa `.orig` è inserita prima di `.tar.gz`.

Si dovrebbe aver notato che nella sottodirectory dei sorgenti `debian` sono stati creati molti modelli di file. Questo verrà trattato in Capitolo 4 e Capitolo 5. Dovreste aver capito che la pacchettizzazione non è un processo automatico. È necessario modificare il sorgente originale per Debian (come descritto in Capitolo 3). Dopo di questo, è necessario creare i pacchetti Debian in maniera appropriata (Capitolo 6), provarli Capitolo 7, e caricarli (Capitolo 9). Tutti i passaggi verranno approfonditi in seguito.

[16] Il seguente testo assume che stiate utilizzando Bash come shell di login. Se si utilizza un'altra shell di login, come la Z shell, bisognerà usare i suoi file di configurazione al posto di `~/.bashrc`.

[17] Se i sorgenti originali contengono già la directory `debian` e il suo contenuto, si deve eseguire il comando **dh_make** con l'opzione `--addmissing`. Il nuovo formato dei sorgenti `3.0 (quilt)` è abbastanza maturo da non danneggiare questi pacchetti. Potrebbe essere necessario aggiornare i contenuti forniti nei sorgenti originali per il pacchetto Debian.

[18] Ecco le varie opzioni: `s` che sta per binario Singolo, `i` per Indipendente dall'architettura, `m` per binario Multiplo, `l` per Libreria, `k` per modulo del Kernel, `n` per patch del kernel e `b` per `cdbs`. Questo documento si basa sull'uso del comando **dh** (contenuto nel pacchetto `debhelper`) per la creazione di un pacchetto contenente un singolo binario, verrà trattato anche il funzionamento dei pacchetti indipendenti dall'architettura e i pacchetti con binari multipli. Il pacchetto `cdbs` offre un'infrastruttura di script alternativa al comando **dh** e non rientra nell'ambito di questo documento.

Se accidentalmente viene cancellato qualche modello di file mentre ci si lavora, è possibile recuperarlo eseguendo **dh_make** con l'opzione `--addmissing` nella directory dei sorgenti del pacchetto Debian.

L'aggiornamento di un pacchetto già esistente può diventare complicato, perché è possibile che si siano usate vecchie tecniche di pacchettizzazione. Per adesso, è consigliabile, concentrarsi sulla creazione di nuovi pacchetti per imparare le basi. Si tornerà ad approfondire l'argomento più avanti su Capitolo 8.

Si tenga presente che non è necessario che il file dei sorgenti contenga qualsiasi sistema di creazione, discusso in Sezione 2.4 and Sezione 2.5. Potrebbe essere solo una collezione di file grafici, ecc. L'installazione di questi file può essere effettuata utilizzando solamente i file di configurazione di `debhelper`, come `debian/install` (vedere Sezione 5.11).

Capitolo 3

Modificare i sorgenti

Non c'è spazio qui per approfondire *tutti* i dettagli su come modificare i sorgenti originali, ma verranno trattati alcuni passaggi fondamentali e le problematiche più comuni.

3.1 Configurare quilt

Il programma **quilt** offre un metodo semplice per registrare le modifiche dei sorgenti originali per la pacchettizzazione Debian. È comodo averlo leggermente personalizzato per la pacchettizzazione Debian, utilizzando l'alias **dquilt**, aggiungendo le seguenti linee al file `~/.bashrc`. La seconda riga fornisce la stessa funzionalità di shell completion del comando **quilt** al comando **dquilt**:

```
alias dquilt="quilt --quiltrc=${HOME}/.quiltrc-dpkg"
complete -F _quilt_completion -o filenames dquilt
```

Si crei il file `~/.quiltrc-dpkg` come segue:

```
d=. ; while [ ! -d $d/debian -a $(readlink -e $d) != / ]; do d=$d/..; done
if [ -d $d/debian ] && [ -z $QUILT_PATCHES ]; then
    # if in Debian packaging tree with unset $QUILT_PATCHES
    QUILT_PATCHES="debian/patches"
    QUILT_PATCH_OPTS="--reject-format=unified"
    QUILT_DIFF_ARGS="-p ab --no-timestamps --no-index --color=auto"
    QUILT_REFRESH_ARGS="-p ab --no-timestamps --no-index"
    QUILT_COLORS="diff_hdr=1;32:diff_add=1;34:diff_rem=1;31:diff_hunk=1;33:diff_ctx=35: ↩
        diff_cctx=33"
    if ! [ -d $d/debian/patches ]; then mkdir $d/debian/patches; fi
fi
```

Per l'utilizzo di **quilt** si veda quilt(1) e `/usr/share/doc/quilt/quilt.pdf.gz`.

3.2 Correggere i bug nel sorgente originale

Si supponga di trovare un errore, nel file `Makefile` distribuito con il programma originale, ad esempio la stringa `install: gentoo` avrebbe dovuto essere `install: gentoo-target`.

```
install: gentoo
        install ./gentoo $(BIN)
        install icons/* $(ICONS)
        install gentoorc-example $(HOME)/.gentoorc
```

Si può correggere questo errore e registrarlo, con il comando **dquilt** utilizzando come file `fix-gentoo-target.patch`: [1]

```
$ mkdir debian/patches
$ dquilt new fix-gentoo-target.patch
$ dquilt add Makefile
```

Si modifichi il file `Makefile` come segue:

```
install: gentoo-target
        install ./gentoo $(BIN)
        install icons/* $(ICONS)
        install gentoorc-example $(HOME)/.gentoorc
```

Adesso bisogna chiedere a **dquilt** di generare la patch `debian/patches/fix-gentoo-target.patch` e di aggiungere una descrizione conforme a DEP-3: Patch Tagging Guidelines (http://dep.debian.net/deps/dep3/) :

```
$ dquilt refresh
$ dquilt header -e
... describe patch
```

3.3 Installazione dei file nei loro percorsi

La maggiorparte dei programmi di terze parti si auto-installano in sottodirectory di `/usr/local`. I pacchetti Debian invece non devono usare quella directory, dal momento che è riservata agli amministratori (o utenti) del sistema per uso privato, ma dovrebbero usare le directory di sistema come la sottodirectory `/usr/bin`, conformi allo Standard di Gerarchia dei Filesystem (FHS (http://www.debian.org/doc/packaging-manuals/fhs/fhs-3.0.html) , `/usr/share/doc/debian-policy/fhs/fhs-2.3.html`).

Normalmente, make(1) è usato per costruire automaticamente il programma, invece l'esecuzione di `make install` installa il programma direttamente nella destinazione prestabilita (configurata nella sezione `install` nel file `Makefile`). Per far in modo che Debian fornisca dei pacchetti installabili pre-compilati, viene modificato il sistema di compilazione in modo da installare i programmi in un'immagine dell'albero dei file, creata dentro una directory temporanea, anziché nella destinazione prestabilita.

Le 2 differenze tra la normale installazione di un programma e l'installazione tramite il sistema di pacchettizzazione Debian può essere affrontato in modo trasparente dal pacchetto `debhelper`, attraverso i comandi **dh_auto_configure** e **dh_auto_install**, se le seguenti condizioni sono soddisfatte:

- Il file `Makefile` deve seguire le convenzioni GNU e supportare la variabile `$(DESTDIR)`. [2]
- Il sorgente deve essere conforme allo Standard di Gerarchia dei Filesystem (FHS).

I programmi che usano GNU **autoconf** sono automaticamente conformi alle convenzioni GNU e la loro pacchettizzazione può essere molto semplice. Con questi ed altri accorgimenti, si stima che il pacchetto `debhelper` funzioni su quasi il 90% dei pacchetti senza apportare pesanti modifiche al loro sistema di compilazione. La pacchettizzazione non è così complicata come potrebbe sembrare.

Se è necessario apportare delle modifiche al `Makefile`, si dovrebbe fare attenzione a supportare la variabile `$(DESTDIR)`. Anche se non è impostata di default, la variabile `$(DESTDIR)` viene anteposta ad ogni percorso usato per l'installazione del programma. Lo script di pacchettizzazione imposta la variabile `$(DESTDIR)` nella directory temporanea.

Per il pacchetto sorgente che genera un singolo pacchetto binario, la directory temporanea, utilizzata dal comando **dh_auto_install**, è `debian/`*`pacchetto`*. [3] Tutto quello contenuto nella directory temporanea verrà installato sul sistema dell'utente, appena si installa il pacchetto, l'unica differenza è che **dpkg** installerà i file nella radice del file system anzichè nella directory di lavoro.

[1] La directory `debian/patches` dovrebbe essere stata creata se si è eseguito **dh_make**, come descritto prima. Questo esempio crea la directory nel caso in cui si stia aggiornando un pacchetto esistente.

[2] Vedere GNU Coding Standards: 7.2.4 DESTDIR: Support for Staged Installs (http://www.gnu.org/prep/standards/html_node/DESTDIR.html#DESTDIR) .

[3] Per il pacchetto sorgente che genera pacchetti binari multipli, il comando **dh_auto_install** utilizza la directory temporanea `debian/tmp`, mentre il comando **dh_install**, con l'aiuto dei file `debian/`*`pacchetto-1`*`.install` e `debian/`*`pacchetto-2`*`.install` suddivide il contenuto di `debian/tmp` nelle directory temporanee `debian/`*`pacchetto-1`* e `debian/`*`pacchetto-2`* per creare pacchetti binari *`pacchetto-1_*.deb`* e *`pacchetto-2_*.deb`*.

Bisogna tenere in considerazione che, anche se il programma viene installato in `debian/`*`pacchetto`*, deve comunque potersi installare nella directory radice dal pacchetto `.deb`. Per questo motivo non bisogna consentire al sistema di compilazione di utilizzare stringhe impostate manualmente come costanti, ad esempio `/home/me/deb/`*`pacchetto-versione`*`/usr/share/`*`pacchetto`* nei file del pacchetto.

Questa è la parte più importante del `Makefile` del pacchetto `gentoo` [4]:

```
# Where to put executable commands on 'make install'?
BIN     = /usr/local/bin
# Where to put icons on 'make install'?
ICONS   = /usr/local/share/gentoo
```

Da notare che i file verranno installati in `/usr/local`. Come descritto sopra, questa directory, in Debian, è riservata per l'utilizzo locale, si modifichino questi percorsi con:

```
# Where to put executable commands on 'make install'?
BIN     = $(DESTDIR)/usr/bin
# Where to put icons on 'make install'?
ICONS   = $(DESTDIR)/usr/share/gentoo
```

Le posizioni che dovrebbero essere usate per i binari, le icone, la documentazione, ecc. sono specificate nella Gerarchia dei Filesystem (FHS). Si consiglia di sfogliarlo e leggere le sezioni riguardanti il pacchetto interessato.

Si dovrà installare, quindi, il file eseguibile in `/usr/bin` anziché in `/usr/local/bin`, la pagina di manuale in `/usr/share/man/man1` anziché `/usr/local/man/man1`, ecc. Da notare che nel `Makefile` del pacchetto `gentoo` non è presente una pagina di manuale, ma dal momento che la policy di Debian prevede che ogni programma ne abbia una, ne verrà creata una e sarà installata in `/usr/share/man/man1`.

Alcuni programmi non usano le variabili nel `Makefile` per definire dei percorsi come questi. Questo indica che potrebbe essere necessario modificare qualche sorgente in C, per fargli usare il percorso giusto. Ma dove cercarlo, e per farci cosa? Lo si può scoprire eseguendo questo:

```
$ grep -nr --include='*.[c|h]' -e 'usr/local/lib' .
```

grep cercherà tutte le corrispondenze in maniera ricorsiva attraverso tutto l'albero dei sorgenti, indicando il nome del file e il numero della riga.

Si modifichino quei file in quelle righe, sostituendo `usr/local/lib` con `/usr/lib`. Questo può essere fatto automaticamente, come mostrato di seguito:

```
$ sed -i -e 's#usr/local/lib#usr/lib#g' \
       $(find . -type f -name '*.[c|h]')
```

Se invece si vuole dare la conferma per ogni sostituzione, si può utilizzare il seguente comando:

```
$ vim '+argdo %s#usr/local/lib#usr/lib#gce|update' +q \
       $(find . -type f -name '*.[c|h]')
```

A questo punto si dovrebbe trovare il target `install` (si cerchi la riga che inizia con `install:`, di solito è quella corretta) e modificare tutti i riferimenti alle directory diverse da quelle definite nel `Makefile`.

Originariamente, il target del pacchetto `gentoo` riporterà:

```
install: gentoo-target
       install ./gentoo $(BIN)
       install icons/* $(ICONS)
       install gentoorc-example $(HOME)/.gentoorc
```

Si può correggere e registrare la modifica con il comando **dquilt** salvandola come `debian/patches/install.patch`.

[4] Questo è solo un esempio che mostra come un `Makefile` dovrebbe apparire. Se il `Makefile` è creato dal comando **./configure**, il modo giusto per correggere il `Makefile` è eseguire il comando **./configure** dal comando **dh_auto_configure** includendo come opzione predefinita `--prefix=/usr`.

```
$ dquilt new install.patch
$ dquilt add Makefile
```

Per il pacchetto debian si modifichi come segue, utilizzando un editor:

```
install: gentoo-target
        install -d $(BIN) $(ICONS) $(DESTDIR)/etc
        install ./gentoo $(BIN)
        install -m644 icons/* $(ICONS)
        install -m644 gentoorc-example $(DESTDIR)/etc/gentoorc
```

Si sarà fatto caso che adesso c'è un comando `install -d` prima degli altri, nella regola. Il `Makefile` originale non ce l'ha perché generalmente usa `/usr/local/bin` e altre directory che già esistono nel sistema su cui si esegue `make install`. Tuttavia, dal momento che verrà installato nella nostra directory vuota (o anche inesistente), si dovrà creare ogni singola directory.

È possibile anche aggiungere altre cose alla fine della regola, come l'installazione di documentazione aggiuntiva che gli autori originali talvolta omettono:

```
        install -d $(DESTDIR)/usr/share/doc/gentoo/html
        cp -a docs/* $(DESTDIR)/usr/share/doc/gentoo/html
```

Dopo un'attenta analisi, se tutto è andato bene, si può generare con **dquilt** la patch che crea il file `debian/patches/install.patch` ed aggiungendogli una descrizione:

```
$ dquilt refresh
$ dquilt header -e
... describe patch
```

Adesso si avranno una serie di patch.

1. Correzione del bug riguardante il sorgente originale: `debian/patches/fix-gentoo-target.patch`
2. Modifica specifica per il sistema di pacchettizzazione Debian: `debian/patches/install.patch`

Ogni volta che si apportano delle modifiche che non sono specificatamente legate alla pacchettizzazione Debian, come `debian/patches/fix-gentoo-target.patch`, bisogna assicurarsi di inviare le modifiche al manutentore originale, in modo che possano essere incluse nella prossima versione del programma e possano beneficiarne altri utenti. Ci si ricordi, di creare delle correzioni portabili, ovvero di non renderle specifiche per Debian o Linux —o altri Unix!. Questo renderà più semplice applicare le correzioni.

Da notare che non è necessario inviare i file `debian/*` all'autore originale.

3.4 Distinguere le librerie

C'è un altro problema comune: le librerie sono spesso diverse da piattaforma a piattaforma. Per esempio, il `Makefile` può contenere un riferimento a una libreria che non esiste nei sistemi Debian. In tal caso occorre cambiare il riferimento ad una libreria che serve allo stesso scopo e che esista in Debian.

Supponiamo che una riga nel `Makefile` del programma (o nel `Makefile.in`) che riporta qualcosa come segue.

```
LIBS = -lfoo -lbar
```

Se il programma non viene compilato perché la libreria `foo` non esiste e il suo equivalente, su un sistema Debian, è fornito dalla libreria `foo2`, è possibile risolvere questo problema di compilazione, modificando il file `debian/patches/foo2.patch` cambiando `foo` con `foo2`: [5]

[5] Se ci sono modifiche delle API dalla libreria `foo` alla libreria `foo2`, le modifiche necessarie per il codice sorgente devono essere fatte basandosi sulle nuove API.

```
$ dquilt new foo2.patch
$ dquilt add Makefile
$ sed -i -e ’s/-lfoo/-lfoo2/g’ Makefile
$ dquilt refresh
$ dquilt header -e
... describe patch
```

Capitolo 4

File richiesti nella directory `debian`

C'è una nuova sottodirectory all'interno della cartella contenente i sorgenti del programma ed è chiamata `debian`. All'interno di questa vi sono una serie di file che dovranno essere modificati per personalizzare il comportamento del pacchetto. I più importanti fra tutti questi sono i file `control`, `changelog`, `copyright` e `rules`, che vengono richiesti per tutti i pacchetti. [1]

4.1 `control`

Questo file contiene diversi valori che **dpkg**, **dselect**, **apt-get**, **apt-cache**, **aptitude**, ed altri strumenti utilizzeranno per gestire il pacchetto. Il tutto è definito nel Manuale delle policy di Debian, 5 "File di controllo ed i loro campi" (http://www.debian.org/doc/debian-policy/ch-controlfields.html) .

Questo è il file di `control` che **dh_make** crea:

```
 1 Source: gentoo
 2 Section: unknown
 3 Priority: optional
 4 Maintainer: Josip Rodin <joy-mg@debian.org>
 5 Build-Depends: debhelper (>=10)
 6 Standards-Version: 4.0.0
 7 Homepage: <insert the upstream URL, if relevant>
 8
 9 Package: gentoo
10 Architecture: any
11 Depends: ${shlibs:Depends}, ${misc:Depends}
12 Description: <insert up to 60 chars description>
13  <insert long description, indented with spaces>
```

(Sono stati aggiunti i numeri di riga.)

Le righe 1–7 contengono le informazioni di controllo per il pacchetto sorgente. Le righe 9-13 contengono le informazioni di controllo per il pacchetto binario.

La riga 1 contiene il nome del pacchetto sorgente.

La riga 2 indica la sezione della distribuzione in cui il pacchetto sorgente dovrà andare.

Come si sarà notato, l'archivio Debian è diviso in diverse aree: `main` (software libero), `non-free` (software non propriamente libero) e `contrib` (software libero che dipende da software non libero). All'interno di queste esistono delle sezioni che descrivono brevemente quali pacchetti vi si possono trovare. Quindi si hanno le sezioni `admin` per i programmi legati all'amministrazione di sistema, `base` per gli strumenti di base, `devel` per gli strumenti di sviluppo, `doc` per la documentazione, `libs`

[1] In questo capitolo, per semplicità, i file nella directory `debian` sono indicati senza la directory radice `debian/`, ogni volta che il loro significato è scontato.

per le librerie, `mail` per client di posta e server associati, `net` per applicazioni e servizi di rete, `x11` per programmi X11 che non appartengono alle altre categorie, e tanti altri. [2]

Si può cambiare il valore in x11. (Il prefisso `main/` è implicito e può essere omesso.)

La riga numero 3 indica quanto sia importante per l'utente installare questo pacchetto. [3]

- La priorità `optional` solitamente viene usata per i nuovi pacchetti che non vanno in conflitto con altri pacchetti con priorità `required`, `important` o `standard`.

Le sezioni e le priorità vengono solitamente utilizzate da interfacce come **aptitude** in cui i pacchetti vengono suddivisi e vengono selezionati quelli predefiniti. Una volta caricato il pacchetto in Debian, il valore di ciascuno di questi due campi può essere sovrascritto dai manutentori dell'archivio, in tal caso si verrà avvertiti via mail.

Dal momento che il pacchetto trattato ha una priorità normale e non va in conflitto con altri, si cambierà la priorità a `optional`.

La riga 4 indica il nome e l'indirizzo email del manutentore. Ci si assicuri che questo campo includa una testata `To` valida per un indirizzo mail, perché una volta caricato il pacchetto, il sistema di rilevazione bug la userà per inviare le mail contenenti i bug. Si eviti di utilizzare virgole, 'e' commerciali o parentesi.

La riga 5 include la lista dei pacchetti richiesti per costruire il pacchetto, ad es. il campo `Build-Depends`. Si può, inoltre, avere una riga contenente il campo `Build-Depends-Indep`. [4] . Alcuni pacchetti come `gcc` e `make` sono richiesti implicitamente, dal pacchetto `build-essential`. Se si ha la necessità di avere altri strumenti per costruire il pacchetto, questi devono essere aggiunti negli appositi campi. I campi multipli sono separati con le virgole; si legga una spiegazione sulle dipendenze binarie per scoprirne di più sulla sintassi di queste righe.

- Per tutti i pacchetti creati utilizzando il comando **dh** nel file `debian/rules`, è necessario avere `debhelper (>=9)` nel campo `Build-Depends`, per aderire alle policy di Debian che richiedono per l'obiettivo `clean`.
- I sorgenti dei pacchetti che hanno i pacchetti binari con il campo `Architecture: any`, vengono ricompilati con autobuilder. Poichè questa procedura installa i soli pacchetti elencati nel campo `Build-Depends`, prima di eseguire `debian/rules build` (vedere Sezione 6.2), il campo `Build-Depends` ha bisogno di tutti i pacchetti necessari, ed il campo `Build-Depends-In` è raramente utilizzato.
- Per i sorgenti dei pacchetti che hanno i pacchetti binari con campo `Architecture: all`, il campo `Build-Depends-Indep` elenca tutti i pacchetti necessari, se non sono già elencati nel campo `Build-Depends`, per essere conforme alle linee guida di Debian riguardanti il target `clean`.

Se non si è sicuri sul campo da utilizzare, si può scegliere il campo `Build-Depends`.[5]

Per scoprire di quali pacchetti si ha bisogno per la compilazione si può eseguire il comando:

```
$ dpkg-depcheck -d ./configure
```

Per scoprire manualmente le esatte dipendenze per ***`/usr/bin/foo`***, basta eseguire

```
$ objdump -p /usr/bin/foo | grep NEEDED
```

e per ogni libreria elencata (ad esempio, **libfoo.so.6**), si esegue

```
$ dpkg -S libfoo.so.6
```

[2] Vedere Manuale delle policy di Debian, 2.4 "Sezioni" (http://www.debian.org/doc/debian-policy/ch-archive.html#s-subsections) e Elenco delle sezioni in `sid` (http://packages.debian.org/unstable/) .

[3] Vedere Manuale delle policy di Debian, 2.5 "Priorità" (http://www.debian.org/doc/debian-policy/ch-archive.html#s-priorities) .

[4] Vedere Manuale delle policy di Debian, 7.7 "Relazioni tra pacchetti sorgenti e i pacchetti binari - Build-Depends, Build-Depends-Indep, Build-Conflicts, Build-Conflicts-Indep" (http://www.debian.org/doc/debian-policy/ch-relationships.html#s-sourcebinarydeps) .

[5] Questa strana situazione è ben documentata in Debian Policy Manual, Footnotes 55 (http://www.debian.org/doc/debian-policy/footnotes.html#f55) . Questo non è dovuto all'uso del comando **dh** nel file `debian/rules`, ma bensì al funzionamento di **dpkg-buildpackage**.La stessa situazione si presenta per il sistema automatico di compilazione di Ubuntu (https://bugs.launchpad.net/launchpad-buildd/+bug/238141) .

A questo punto si indica la versione `-dev` di ogni pacchetto come voce `Build-Depends`. Se si usa **ldd** per questo scopo, verranno considerate anche le dipendenze indirette, il che potrà portare ad avere un numero eccessivo di dipendenze.

Il pacchetto `gentoo` richiede anche `xlibs-dev`, `libgtk1.2-dev` e `libglib1.2-dev` per poter essere costruito, quindi tali dipendenze si aggiungeranno subito dopo `debhelper`.

La riga 6 indica la versione delle delle linee guida Debian (http://www.debian.org/doc/devel-manuals#policy) che il pacchetto deve rispettare, che corrisponde a quello che si legge quando lo si crea.

Nella riga 7 si può inserire l'URL della pagina del programma originale.

La riga 9 indica il nome del pacchetto binario. Questo è normalmente lo stesso nome del pacchetto sorgente, ma non deve essere necessariamente così.

La riga 10 specifica le architetture per cui è possibile compilare il pacchetto. Questo valore è di solito uno dei seguenti, a seconda del tipo di pacchetto binario: [6]

- `Architecture: any`
  - Il pacchetto binario generato è compatibile con una sola architettura, solitamente è utilizzato in programmi creati con un linguaggio compilato.
- `Architecture: all`
  - Il pacchetto binario generato è indipendente dall'architettura, di solito si tratta di testo, immagini o script in un linguaggio interpretato.

Si lasci la riga 10 così com'è visto che il programma è scritto in C.dpkg-gencontrol(1) riempirà il campo dell'architettura con un valore adeguato per ciascuna macchina in cui il pacchetto viene compilato.

Se il pacchetto è indipendente dall'architettura (per esempio, uno script shell o Perl, o un documento), si cambi questo valore in `all`, e si legga in seguito in Sezione 4.4 riguardo l'utilizzo della regola `binary-indep` al posto di `binary-arch` per costruire il pacchetto.

La riga 11 mostra una delle caratteristiche più potenti del sistema di pacchettizzazione Debian. I pacchetti possono relazionarsi tra di loro in vari modi. A parte `Depends`, altri campi di relazione sono `Recommends`, `Suggests`, `Pre-Depends`, `Breaks`, `Conflicts`, `Provides` e `Replaces`.

Gli strumenti di gestione dei pacchetti solitamente si comportano allo stesso modo quando si occupano di tali relazioni; in caso contrario, il comportamento verrà spiegato. (si legga dpkg(8), dselect(8), apt(8), aptitude(1) ecc.)

Ecco una descrizione semplificata delle relazioni tra i pacchetti: [7]

- `Depends`

  Il pacchetto non verrà installato a meno che tutti i pacchetti da cui dipende vengono installati. Si usi questa relazione se il programma non funzionerà assolutamente (o sarà praticamente inutilizzabile) a meno della presenza di particolari pacchetti.

- `Recommends`

  Si usi questa relazione per i pacchetti che non sono strettamente necessari ma sono solitamente utilizzati dal programma. Quando un utente installa il programma, tutte le interfacce di APT probabilmente chiederanno l'installazione dei pacchetti raccomandati. **aptitude** e **apt-get** installano, in modo predefinito, i pacchetti raccomandati insieme al pacchetto principale (ma l'utente può disabilitare questo comportamento predefinito). **dpkg** ignorerà questo campo.

- `Suggests`

  Si usi questa relazione per pacchetti che funzionano bene con il programma ma non sono per niente necessari. Quando un utente installa il programma, probabilmente non verrà chiesta l'installazione dei pacchetti consigliati. **aptitude** può essere configurato per installare i pacchetti consigliati insieme al pacchetto principale ma questo non è il comportamento predefinito. **dpkg** ed **apt-get** ignoreranno questo campo.

[6] Vedere Manuale delle policy di Debian, 5.6.8 "Architettura" (http://www.debian.org/doc/debian-policy/ch-controlfields.html#s-f-Architecture) per maggiori informazioni.

[7] Vedere Manuale delle policy di Debian, 7 "Dichiarare le relazioni tra i pacchetti" (http://www.debian.org/doc/debian-policy/ch-relationships.html) .

- `Pre-Depends`

  Questa relazione è più forte di `Depends`. Il pacchetto non verrà installato a meno che i pacchetti da cui pre-dipende sono stati installati e *correttamente configurati*. Si usi questa relazione con *molta* parsimonia e solo dopo averne discusso sulla mailing list debian-devel@lists.debian.org (http://lists.debian.org/debian-devel/) . Leggasi: non utilizzarla affatto. :-)

- `Conflicts`

  Il pacchetto non verrà installato a meno che tutti i pacchetti con i quali va in conflitto siano rimossi. Si usi questa relazione se il programma non funzionerà o causerà gravi problemi se un certo pacchetto è presente.

- `Breaks`

  Una volta installato il pacchetto verranno marcati come ”guasti” tutti i pacchetti elencati. Normalmente la voce `Breaks` specifica che si applica a version precedenti di un certo valore. Per risolvere il problema, generalmente, basta aggiornare la lista dei pacchetti utilizzando uno strumento di gestione di alto livello, del sistema di pacchettizzazione.

- `Provides`

  Per alcuni tipi di pacchetto di cui vi sono molteplici alternative, sono stati definiti dei nomi virtuali. Si può trovare la lista completa nel file virtual-package-names-list.txt.gz (http://www.debian.org/doc/packaging-manuals/virtual-package-names-list.txt) . Si consulti questo file se il programma da pacchettizzare fornisce la funzione di un pacchetto virtuale esistente.

- `Replaces`

  Si usi questa relazione quando il programma rimpiazza i file di un altro pacchetto, o lo rimpiazza completamente (utilizzato in congiunzione con `Conflicts`). I file dei pacchetti indicati saranno sovrascritti con i file del nuovo pacchetto.

Tutti i campi qui descritti hanno una sintassi uniforme. Sono costituiti da una lista contenente i nomi dei pacchetti separati da virgole. Questi possono essere anche costituiti da liste di nomi di pacchetto alternativi, separati da barre verticali `|` (simboli pipe).

I campi possono limitare la loro applicabilità a particolari versioni di ogni pacchetto indicato. Le restrizioni di ogni singolo sono elencate tra parentesi dopo il nome de pacchetto, e dovrebbero contenere una relazione presa dalla lista qui sotto, seguita dal numero di versione. Le relazioni permesse sono: <<, <=, =, >= e >> per strettamente inferiore, inferiore o uguale, esattamente uguale, superiore o uguale e strettamente superiore, rispettivamente. Per esempio,

```
Depends: foo (>= 1.2), libbar1 (= 1.3.4)
Conflicts: baz
Recommends: libbaz4 (>> 4.0.7)
Suggests: quux
Replaces: quux (<< 5), quux-foo (<= 7.6)
```

L’ultima caratteristica utile da conoscere è `${shlibs:Depends}`, `${perl:Depends}`, `${misc:Depends}`, ecc.

dh_shlibdeps(1) calcola le dipendenze delle librerie condivise per i pacchetti binari. Questo genera un’elenco di eseguibili ELF e di librerie condivise che sono stati trovati in ogni pacchetto binario. Questa lista è utilizzata per `${shlibs:Depends}`.

dh_perl(1) calcola le dipendenze perl. Questo genera un elenco di dipendenze `perl` o `perlapi` per ogni pacchetto binario. Questa lista è utilizzata per `${perl:Depends}`.

Alcuni comandi `debhelper` possono far si che il pacchetto generato abbia bisogno di dipendere da altri pacchetti. Questa lista di pacchetti richiesti è utilizzata per `${misc:Depends}`.

dh_gencontrol(1) genera il file `DEBIAN/control` per ogni pacchetto binario che utilizza `${shlibs:Depends}`, `${perl:Depend` `${misc:Depends}`, ecc.

Detto ciò, si può lasciare la riga `Depends` esattamente come è ora, e si può inserire un’altra riga dopo questa che dica `Suggests: file`, perché `gentoo` può utilizzare alcune caratteristiche fornite dal pacchetto `file`.

La riga 9 è l’URL dell’homepage. Supponiamo che questa sia http://www.obsession.se/gentoo/ .

La riga 12 contiene una breve descrizione del pacchetto. Usualmente la larghezza dei terminali è di 80 colonne quindi il contenuto non dovrebbe superare i 60 caratteri. Si cambia questo valore in `fully GUI-configurable, two-pane X file manager`.

Nella riga 13 va messa la descrizione lunga. Questa dovrebbe consistere in un paragrafo che fornisce più dettagli sul pacchetto. La prima colonna di ogni riga dovrebbe essere vuota. Non ci dovrebbero essere linee vuote, ma si può mettere un singolo `.` (punto) in una colonna per simularle. Inoltre non ci dovrebbe essere più di una linea vuota dopo questa descrizione. [8]

Si possono inseriscono i campi `Vcs-*` per documentare il Version Control System (VCS) tra la linea 6 e 7. [9] Si supponga che il pacchetto `gentoo` abbia il suo VCS nel servizio Git Debian Alioth su `git://git.debian.org/git/collab-maint/gentoo`

E per concludere, questo è il file `control` aggiornato:

```
 1 Source: gentoo
 2 Section: x11
 3 Priority: optional
 4 Maintainer: Josip Rodin <joy-mg@debian.org>
 5 Build-Depends: debhelper (>=10), xlibs-dev, libgtk1.2-dev, libglib1.2-dev
 6 Standards-Version: 4.0.0
 7 Vcs-Git: https://anonscm.debian.org/git/collab-maint/gentoo.git
 8 Vcs-browser: https://anonscm.debian.org/git/collab-maint/gentoo.git
 9 Homepage: http://www.obsession.se/gentoo/
10
11 Package: gentoo
12 Architecture: any
13 Depends: ${shlibs:Depends}, ${misc:Depends}
14 Suggests: file
15 Description: fully GUI-configurable, two-pane X file manager
16  gentoo is a two-pane file manager for the X Window System. gentoo lets the
17  user do (almost) all of the configuration and customizing from within the
18  program itself. If you still prefer to hand-edit configuration files,
19  they're fairly easy to work with since they are written in an XML format.
20  .
21  gentoo features a fairly complex and powerful file identification system,
22  coupled to an object-oriented style system, which together give you a lot
23  of control over how files of different types are displayed and acted upon.
24  Additionally, over a hundred pixmap images are available for use in file
25  type descriptions.
26  .
29  gentoo was written from scratch in ANSI C, and it utilizes the GTK+ toolkit
30  for its interface.
```

(Sono stati aggiunti i numeri di riga.)

4.2 `copyright`

Questo file contiene le informazioni sul copyright e la licenza del programma originale. Il suo contenuto è descritto nel Manuale delle policy di Debian, 12.5 'Informazioni di copyright' (http://www.debian.org/doc/debian-policy/ch-docs.html#s-copyrightfile) , e il documento DEP-5: Machine-parseable debian/copyright (http://dep.debian.net/deps/dep5/) fornisce le linee guida del suo formato.

dh_make può fornire un modello di file del `copyright`, basta utilizzare l'opzione `--copyright` per selezionare quello giusto, se si desidera rilasciare il pacchetto `gentoo` sotto licenza GPL-2.

You must fill in missing information to complete this file, such as the place you got the package from, the actual copyright notice, and the license. For certain common free software licenses (GNU GPL-1, GNU GPL-2, GNU GPL-3, LGPL-2, LGPL-2.1, LGPL-3, GNU FDL-1.2, GNU FDL-1.3, Apache-2.0, 3-Clause BSD, CC0-1.0, MPL-1.1, MPL-2.0 or the Artistic license), you can just refer to the appropriate file in the `/usr/share/common-licenses/` directory that exists on every Debian system. Otherwise, you must include the complete license.

Brevemente, ecco come il file di `copyright` del pacchetto `gentoo` dovrebbe apparire:

[8] Queste descrizioni sono in inglese. Le traduzioni di queste descrizioni sono fornite da The Debian Description Translation Project - DDTP (http://www.debian.org/intl/l10n/ddtp) .

[9] Vedere Guida di riferimento per lo sviluppatore Debian, 6.2.5. "Version Control System location" (http://www.debian.org/doc/manuals/developers-reference/best-pkging-practices.html#bpp-vcs) .

```
 1 Format: https://www.debian.org/doc/packaging-manuals/copyright-format/1.0/
 2 Upstream-Name: gentoo
 3 Upstream-Contact: Emil Brink <emil@obsession.se>
 4 Source: http://sourceforge.net/projects/gentoo/files/
 5
 6 Files: *
 7 Copyright: 1998-2010 Emil Brink <emil@obsession.se>
 8 License: GPL-2+
 9
10 Files: icons/*
11 Copyright: 1998 Johan Hanson <johan@tiq.com>
12 License: GPL-2+
13
14 Files: debian/*
15 Copyright: 1998-2010 Josip Rodin <joy-mg@debian.org>
16 License: GPL-2+
17
18 License: GPL-2+
19  This program is free software; you can redistribute it and/or modify
20  it under the terms of the GNU General Public License as published by
21  the Free Software Foundation; either version 2 of the License, or
22  (at your option) any later version.
23  .
24  This program is distributed in the hope that it will be useful,
25  but WITHOUT ANY WARRANTY; without even the implied warranty of
26  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
27  GNU General Public License for more details.
28  .
29  You should have received a copy of the GNU General Public License along
30  with this program; if not, write to the Free Software Foundation, Inc.,
31  51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.
32  .
33  On Debian systems, the full text of the GNU General Public
34  License version 2 can be found in the file
35  '/usr/share/common-licenses/GPL-2'.
```

(Sono stati aggiunti i numeri di riga.)

Si prega di seguire l'HOWTO fornito da ftpmasters ed inviato a debian-devel-announce: http://lists.debian.org/debian-devel-announce/2006/03/msg00023.html.

4.3 changelog

Questo è un file obbligatorio, che ha un formato speciale descritto nel Manuale delle policy di Debian, 4.4 "debian/changelog" (http://www.debian.org/doc/debian-policy/ch-source.html#s-dpkgchangelog) . Questo formato è utilizzato da **dpkg** ed altri programmi per ottenere il numero di versione, revisione, distribuzione ed urgenza del pacchetto.

Tale file è anche utile allo scopo di aver documentato tutti i cambiamenti che sono stati fatti. Sarà inoltre d'aiuto agli utenti che scaricano il pacchetto per vedere se ci sono problemi di cui dovrebbero essere al corrente. Il file verrà salvato come `/usr/share/doc/gentoo/changelog.Debian.gz` nel pacchetto binario.

dh_make ne crea uno predefinito, ecco come appare:

```
1  gentoo (0.9.12-1) unstable; urgency=medium
2
3   * Initial release. (Closes: #nnnn)  <nnnn is the bug number of your ITP>
4
5  -- Josip Rodin <joy-mg@debian.org>  Mon, 22 Mar 2010 00:37:31 +0100
6
```

(Sono stati aggiunti i numeri di riga.)

La riga 1 indica il nome del pacchetto, la versione, la distribuzione e l'urgenza. Il nome deve corrispondere al nome del pacchetto sorgente, mentre la distribuzione dovrebbe essere `unstable`, e l'urgenza dovrebbe essere impostata come media, a meno chè non ci sia un motivo valido per impostarlo diversamente.

Le righe 3-5 sono una voce del registro, in cui vengono documentati i cambiamenti fatti nella revisione del pacchetto (non dei cambiamenti del pacchetto originario —c'è un file apposta per questo scopo, creato dagli autori originali, che verrà installato successivamente `/usr/share/doc/gentoo/changelog.gz`). Supponiamo che il numero di servizio del ticket ITP fosse `12345`. Nuove righe devono essere aggiunte appena prima della riga più in alto che comincia con * (asterisco). Ciò si può fare con dch(1), o manualmente con un editor testuale.

Per evitare che un pacchetto venga accidentalmente caricato prima che il quest'ultimo sia pronto, è buona prassi cambiare il valore del campo distribuzione, con il valore fittizio `UNRELEASED`.

Alla fine si avrà qualcosa del genere:

```
1  gentoo (0.9.12-1) UNRELEASED; urgency=low
2
3   * Initial Release. Closes: #12345
4   * This is my first Debian package.
5   * Adjusted the Makefile to fix $(DESTDIR) problems.
6
7  -- Josip Rodin <joy-mg@debian.org>  Mon, 22 Mar 2010 00:37:31 +0100
8
```

(Sono stati aggiunti i numeri di riga.)

Quando si è soddisfatti di tutte le modifiche e quest'ultime sono state documentate nel file `changelog`, si può modificare il valore del campo distribuzione da `UNRELEASED` a `unstable` (o anche `experimental`). [10]

Si possono avere più informazioni su come aggiornare il file `changelog` nel capitolo Capitolo 8.

4.4 Il file `rules`

Ora si darà uno sguardo alle regole esatte che dpkg-buildpackage(1) userà per creare il pacchetto. In realtà questo file non è che un altro `Makefile`, ma diverso da quelli della sorgente originale. Differentemente dagli altri file sotto `debian`, questo è marcato come eseguibile.

4.4.1 Target del file `rules`

Ogni file `rules`, come ogni altro `Makefile`, si compone di diverse regole, ciascuno dei quali definisce un target e descrive come eseguirlo. [11] Una nuova regola inizia con la dichiarazione del target, nella prima colonna. Le righe seguenti che iniziano con il TAB (codice ASCII 9) specificano il come effettuare il target. Le righe vuote e quelle che iniziano con # (cancelletto) vengono trattate come commenti e ignorate. [12]

Quando si vuole eseguire una regola basta eseguire il comando passandogli come argomento il nome del target. Ad esempio, `debian/rules` *`build`* e `fakeroot make -f debian/rules` *`binary`* esegue le regole per i target *`build`* e *`binary`*.

Ecco una spiegazione semplificata dei target:

- target `clean`: ripulisce tutti i file compilati, generati ed inutili nella struttura delle directory del pacchetto. (richiesto)

[10] Se si utilizza il comando `dch -r` per effettuare quest'ultima modifica, ci si assicuri che l'editor salvi il file con il nome `changelog`.

[11] Si può imparare come scrivere il `Makefile` dalla Guida di riferimento Debian, 12.2. "Make" (http://www.debian.org/doc/manuals/debian-reference/-ch12#_make) . La documentazione completa è disponibile su http://www.gnu.org/software/make/manual/html_node/index.html o come pacchetto `make-doc` nella sezione `non-free` del repository.

[12] Il manuale delle policy Debian, 4.9 "Main building script: debian/rules" (http://www.debian.org/doc/debian-policy/ch-source.html#s-debianrules) descrive i dettagli.

- target `build`: costruisce tutti i sorgenti per ottenere programmi compilati e documenti formattati nella struttura delle directory del pacchetto. (richiesto)
- target `build-arch`: costruisce i sorgenti per ottenere programmi compilati, dipendenti dall'architettura, nella radice dei sorgenti. (richiesto)
- target `build-indep`: costruisce i sorgenti per ottenere documenti formattati indipendenti dall'architettura nella radice dei sorgenti. (richiesto)
- target `install`: installa i file in una struttura ad albero per ogni pacchetto binario nella directory `debian`. Se definito, tutti i target `binary*` dipenderanno effettivamente da quest'ultimo. (opzionale)
- target `binary`: crea tutta una serie di pacchetti binari (combinando i target `binary-arch` e `binary-indep`). (richiesto)[13]
- target `binary-arch`: crea una serie di pacchetti binari dipendenti dall'architettura (`Architecture: any`) nella directory padre. (richiesto)[14]
- target `binary-indep`: crea una serie di pacchetti binari indipendenti dall'architettura (`Architecture: all`) nella directory padre. (richiesto)[15]
- target `get-orig-source`: ottiene la versione più recente del pacchetto sorgente originale dal relativo archivio originale. (optional)

È probabile che adesso ci sia un po' di confusione, ma sarà tutto più chiaro una volta esaminato il file `rules` che **dh_make** fornisce in modo predefinito.

4.4.2 Il file `rules` predefinito

Le nuove versioni di **dh_make** generano un file `rules` molto semplice ma potente utilizzando il comando **dh**:

```
 1 #!/usr/bin/make -f
 2 # See debhelper(7) (uncomment to enable)
 3 # output every command that modifies files on the build system.
 4 #DH_VERBOSE = 1
 5
 6 # see FEATURE AREAS in dpkg-buildflags(1)
 7 #export DEB_BUILD_MAINT_OPTIONS = hardening=+all
 8
 9 # see ENVIRONMENT in dpkg-buildflags(1)
10 # package maintainers to append CFLAGS
11 #export DEB_CFLAGS_MAINT_APPEND  = -Wall -pedantic
12 # package maintainers to append LDFLAGS
13 #export DEB_LDFLAGS_MAINT_APPEND = -Wl,--as-needed
14
15
16 %:
17         dh $@
```

(Sono stati aggiunti i numeri di riga ed eliminati alcuni commenti. Nel vero file `rules`, gli spazi vengono sostituiti da TAB.)

Probabilmente si sarà già familiari con le righe tipo la prima che ricordano gli script shell e Perl. In pratica indica al sistema operativo che il file andrà elaborato con `/usr/bin/make`.

La riga 4 può essere decommentata per impostare la variabile `DH_VERBOSE` a 1, in modo da mostrare gli output del comando **dh** che generano i programmi **dh_*** in esecuzione. È anche possibile aggiungere la riga `export DH_OPTIONS=-v`, in modo che ogni comando **dh_*** stampi quale comando sta eseguendo. Questo aiuta a comprendere esattamente cosa c'è dietro al singolo file `rules` file ed a diagnosticare i problemi. Il nuovo **dh** è progettato per formare una parte fondamentale degli strumenti `debhelper` e per essere trasparente, ovvero non nascondere nulla all'utente.

[13] Questo obiettivo è utilizzato da `dpkg-buildpackage` come in Sezione 6.1.
[14] Questo target è usato da `dpkg-buildpackage -B` come descritto in Sezione 6.2.
[15] Questo target è utilizzato da `dpkg-buildpackage -A`.

Tutto il lavoro è svolto nelle righe 20 e 21, grazie ad una regola implicita dichiarata nel modello. Il simbolo della percentuale significa che ”ogni target” esegue una chiamata ad un singolo programma, **dh** con il nome del target stesso. [16] Il comando **dh** è uno script wrapper, che esegue una sequenza appropriata di programmi **dh_*** in base all'argomento passato. [17]

- `debian/rules clean` esegue `dh clean`; che a sua volta esegue i seguenti:

```
dh_testdir
dh_auto_clean
dh_clean
```

- `debian/rules build` esegue `dh build`, che a sua volta esegue i seguenti:

```
dh_testdir
dh_auto_configure
dh_auto_build
dh_auto_test
```

- `fakeroot debian/rules binary` esegue `fakeroot dh binary`, che a sua volta esegue i seguenti[18]:

```
dh_testroot
dh_prep
dh_installdirs
dh_auto_install
dh_install
dh_installdocs
dh_installchangelogs
dh_installexamples
dh_installman
dh_installcatalogs
dh_installcron
dh_installdebconf
dh_installemacsen
dh_installifupdown
dh_installinfo
dh_installinit
dh_installmenu
dh_installmime
dh_installmodules
dh_installlogcheck
dh_installlogrotate
dh_installpam
dh_installppp
dh_installudev
dh_installwm
dh_installxfonts
dh_bugfiles
dh_lintian
dh_gconf
dh_icons
dh_perl
dh_usrlocal
dh_link
```

[16] Questo è una nuova caratteristica di `debhelper` v7+. La sua architettura è documentata su Not Your Grandpa's Debhelper (http://joey.kitenet.net/talks/debhelper/debhelper-slides.pdf) ed è stata presentata alla Debconf9 dall'autore di `debhelper`. Su sistemi Debian `lenny`, **dh_make** crea file `rules` molto più complicati, con molti script **dh_*** elencati per ogni target, la maggior parte dei quali sono ormai inutili (e rispecchiano l'anzianità del pacchetto). La nuova versione del programma **dh** è molto più semplice e ci libera da questo vincolo. Si continuerà ad avere il pieno potere di personalizzazione utilizzando i target `override_dh_*`. Si veda Sezione 4.4.3. Esso si basa solo sul pacchetto `debhelper` e non offusca il processo di compilazione come il pacchetto `cdbs`.

[17] You can verify the actual sequences of **dh_*** programs invoked for a given *`target`* without really running them by invoking `dh` *`target`* `--no-act` or `debian/rules -- 'target --no-act'`.

[18] Il seguente esempio assume che il file `debian/compat` abbia un valore uguale o superiore a 9, per evitare l'esecuzione automatica di qualsiasi comando python di supporto.

```
dh_compress
dh_fixperms
dh_strip
dh_makeshlibs
dh_shlibdeps
dh_installdeb
dh_gencontrol
dh_md5sums
dh_builddeb
```

- `fakeroot debian/rules binary-arch` esegue `fakeroot dh binary-arch`, che a sua volta esegue la stessa sequenza di `fakeroot dh binary` ma con l'opzione `-a` aggiunta ad ogni comando.
- `fakeroot debian/rules binary-indep` esegue `fakeroot dh binary-indep`, che a sua volta esegue la stessa sequenza di `fakeroot dh binary` ma escludendo **dh_strip**, **dh_makeshlibs**, e **dh_shlibdeps** con l'opzione `-i` aggiunta ad ogni comando rimanente.

Le funzioni dei comandi **dh_*** sono, in gran parte, auto-esplicativi. [19] Ce ne sono alcune però, per cui vale la pena dare più spiegazioni, tenendo conto che si stia lavorando su un ambiente di sviluppo basato sul `Makefile`: [20]

- **dh_auto_clean** normalmente esegue i seguenti comandi se nel `Makefile` è presente il target `distclean`.[21]

```
make distclean
```

- **dh_auto_configure** normalmente esegue i seguenti comandi se esiste il file `./configure` (argomenti abbreviati for una maggiore leggibilità).

```
./configure --prefix=/usr --sysconfdir=/etc --localstatedir=/var ...
```

- **dh_auto_build** normalmente lancia il seguente comando per eseguire, se esiste, il primo target del `Makefile`.

```
make
```

- **dh_auto_clean** normalmente esegue i seguenti comandi, se nel `Makefile` esiste il target `test`.[22]

```
make test
```

- **dh_auto_install** normalmente esegue il seguente comando se nel `Makefile` esiste il target `install` (riga spezzata per aumentare la leggibilità).

```
make install \
  DESTDIR=/path/to/package_version-revision/debian/package
```

Tutti i target che richiedono il comando **fakeroot** dovrebbero contenere **dh_testroot**, che restituisce un errore se non si utilizza questo comando per fingere di essere root.

La cosa importante da sapere riguardo al file `rules` creato da **dh_make**, è che il suo contenuto contiene dei semplici consigli. Funzionerà per la maggior parte dei pacchetti ma per i più complicati non si esiti a personalizzarlo secondo le proprie esigenze.

Anche se il target `install` non è richiesto, è comunque supportato. `fakeroot dh install` si comporta come `fakeroot dh binary` ma si ferma dopo **dh_fixperms**.

[19] Per informazioni dettagliate sul funzionamento di tutti questi script **dh_ ***, e sulle loro opzioni, leggere le rispettive pagine del manuale e `debhelper` documentazione.

[20] Questi comandi supportano altri ambienti di sviluppo, come `setup.py` che può essere elencato eseguendo `dh_auto_build --list` nella directory del sorgente del pacchetto.

[21] Attualmente il programma esegue il primo target presente nel `Makefile`, tra `distclean`, `realclean` o `clean`.

[22] Attualmente il programma esegue il primo target `test` o `check` disponibile nel `Makefile`.

4.4.3 Personalizzazione del file `rules`

Verrà qui spiegata la personalizzazione del file `rules`, creato con il nuovo comando **dh**.

Il comando `dh $@` può essere personalizzato come segue: [23]

- Aggiunge il supporto per il comando **dh_python2**. (La scelta migliore per Python) [24]
  - Include il pacchetto `python` in `Build-Depends`.
  - Utilizza `dh $@ --with python2`.
  - Gestisce i moduli Python utilizzando il framework `python`.
- Aggiunge il supporto per il comando **dh_pysupport**. (deprecato)
  - Include il pacchetto `python-support` in `Build-Depends`.
  - Utilizza `dh $@ --with pysupport`.
  - Gestisce i moduli Python utilizzando l'infrastruttura `python-support`.
- Aggiunge il supporto al comando **dh_pycentral**. (deprecato)
  - Include il pacchetto `python-central` in `Build-Depends`.
  - Utilizza in alternativa `dh $@ --with python-central`.
  - Disattiva anche il comando **dh_pysupport**.
  - Gestisce i moduli Python utilizzando l'infrastruttura `python-central`.
- Aggiunge il supporto per il comando **dh_installtex**.
  - Include il pacchetto `tex-common` in `Build-Depends`.
  - Utilizza in alternativa `dh $@ --with tex`.
  - Registra i caratteri Type 1, le regole di sillabazione, ed i formati con TeX.
- Aggiunge il supporto per i comandi **dh_quilt_patch** e **dh_quilt_unpatch**.
  - Include il pacchetto `quilt` in `Build-Depends`.
  - Utilizza in alternativa `dh $@ --with quilt`.
  - Applica e rimuove le patch al sorgente originale dai file nella directory `debian/patches` per i sorgenti dei pacchetti con formato `1.0`.
  - Non è necessario se si utilizza il nuovo formato del sorgente del pacchetto `3.0 (quilt)`.
- Aggiunge il supporto per il comando **dh_dkms**.
  - Include il pacchetto `dkms` in `Build-Depends`.
  - Utilizza in alternativa `dh $@ --with dkms`.
  - Gestisce in maniera corretta DKMS, usato dal pacchetto del modulo del kernel.
- Aggiunge il supporto per i comandi **dh_autotools-dev_updateconfig** e **dh_autotools-dev_restoreconfig**.
  - Include il pacchetto `autotools-dev` in `Build-Depends`.
  - Include in alternativa `dh $@ --with autotools-dev`.
  - Aggiorna e ripristina i file `config.sub` and `config.guess`.
- Aggiunge il supporto per i comandi **dh_autoreconf** e **dh_autoreconf_clean**.

[23] Se il pacchetto installa il file `/usr/share/perl5/Debian/Debhelper/Sequence/`*`nome_personalizzato`*`.pm`, si deve attivare la funzione di personalizzazione tramite il comando `dh $@ --with` *`nome-personalizzato`*.

[24] L'uso del comando **dh_python2** è preferito rispetto all'uso di dei comandi **dh_pysupport** o **dh_pycentral**. Non si deve utilizzare il comando **dh_python**.

  - Include il pacchetto `dh-autoreconf` in `Build-Depends`.
  - Utilizza in alternativa `dh $@ --with autoreconf`.
  - Aggiorna i file del sistema di compilazione GNU e ripristina i file dopo la sua compilazione.

- Aggiunge il supporto per il comando **dh_girepository**.
  - Include il pacchetto `gobject-introspection` in `Build-Depends`.
  - Utilizza in alternativa `dh $@ --with gir`.
  - Questa operazione calcola le dipendenze per i pacchetti che spediscono dei dati d'auto-analisi di GObject e genera la variabile di sostituzione `${gir:Depends}` per la dipendenza del pacchetto.

- Aggiunge il supporto alla funzionalità di completamento di **bash**.
  - Include il pacchetto `bash-completion` in `Build-Depends`.
  - Utilizza in alternativa `dh $@ --with bash-completion`.
  - Installa il completamento per **bash** utilizzando il file di configurazione `debian/`*`package`*`.bash-completion`.

Molti comandi del tipo **dh_***, invocati da **dh**, possono essere personalizzati modificando i rispettivi file di configurazione nella directory `debian`. Si veda Capitolo 5 per la personalizzazione di tali caratteristiche.

Alcuni comandi del tipo **dh_***, invocati da **dh**, possono richiedere la propria esecuzione con alcuni parametri o in aggiunta ad altri comandi da eseguire contestualmente o al posto dei comandi originali. In tali casi viene creato nel file `rules` il target `override_dh_`*`foo`* aggiungendo una regola solo per il comando **dh_*foo***** che si intende modificare. Fondamentalmente tale regola dice *esegui me al posto di*. [25]

Si noti che il comando **dh_auto_*** tende a fare più di ciò che è stato discusso in questa spiegazione (ultra)semplificata. È una cattiva idea utilizzare i target `override_dh_*` per sostituire i comandi equivalenti (ad eccezione del target `override_dh_auto_clean`) in quanto può bypassare delle caratteristiche "intelligenti" di `debhelper`.

Se si vogliono registrare i dati di configurazione di sistema nella directory `/etc/gentoo` invece che nella solita directory `/etc`, per il pacchetto `gentoo`, che utilizza gli Autotools, si può sovrascrivere il parametro predefinito `--sysconfig=/etc` dato dal comando **dh_auto_configure** al comando **./configure** nel modo seguente:

```
override_dh_auto_configure:
        dh_auto_configure -- --sysconfig=/etc/gentoo
```

I parametri immessi dopo `--` vengono aggiunti dopo i parametri predefiniti dei programmi eseguiti per sovrascriverli. Utilizzare il comando **dh_auto_configure** è preferibile rispetto al comando **./configure** dal momento che sovrascriverà esclusivamente il parametro `--sysconfig` e manterrà gli altri parametri del comando **./configure**.

Se il `Makefile` del sorgente per il pacchetto `gentoo` necessita che venga specificato il target per costruirlo [26], basterà creare il target `override_dh_auto_build` per abilitarlo.

```
override_dh_auto_build:
        dh_auto_build -- build
```

Questo assicura che `$(MAKE)` verrà eseguito con tutti i parametri predefiniti del comando **dh_auto_build** ed il parametro `build`.

Se il `Makefile` del sorgente per il pacchetto `gentoo` necessita che venga specificato il target `packageclean` per pulirlo per il pacchetto Debian, al posto dell'utilizzo dei target `distclean` o `clean`, si può creare il target `override_dh_auto_build` pe abilitarlo.

```
override_dh_auto_clean:
        $(MAKE) packageclean
```

Se il `Makefile` di un sorgente per il pacchetto `gentoo` contiene il target `test` che non vuole essere eseguito nel processo di costruzione del pacchetto Debian, si può utilizzare l'obiettivo `override_dh_auto_test` per saltarlo.

[25] Sotto `lenny`, se si vuole cambiare il comportamento di uno script **dh_*** basta cercare la riga relativa nel file `rules` e modificarla.
[26] **dh_auto_build** senza alcun argomento eseguirà il primo target del file `Makefile`.

```
override_dh_auto_test:
```

Se il programma originale `gentoo` contiene un inusuale file di changelog chiamato `FIXES`, **dh_installchangelogs** non installerà questo file in modo predefinito. Il comando **dh_installchangelogs** richiede che venga fornito il parametro `FIXES` per installarlo.[27]

```
override_dh_installchangelogs:
        dh_installchangelogs FIXES
```

Quando si usa il nuovo comando **dh**, l'utilizzo di target espliciti come quelli elencanti in Sezione 4.4.1, possono rendere difficile capire i loro effetti. Si prega di limitare i target espliciti in favore dei target `override_dh_*` e, se possibile, quelli completamente indipendenti.

[27] I file `debian/changelog` e `debian/NEWS` vengono sempre installati automaticamente. Il changelog originale viene trovato convertendo i nomi dei file in minuscolo e cercando la corrispondenza con `changelog`, `changes`, `changelog.txt`, e `changes.txt`.

Capitolo 5

Altri file nella directory `debian`

Per controllare la maggior parte delle operazioni che `debhelper` effettua durante la creazione del pacchetto, si possono inserire dei file di configurazione all'interno della directory `debian`. Questo capitolo fornirà una panoramica sull'utilizzo di ciascuno di essi ed il loro formato. Si legga il Manuale delle policy di Debian (http://www.debian.org/doc/devel-manuals#policy) e la Guida di riferimento per lo sviluppatore Debian (http://www.debian.org/doc/devel-manuals#devref) per le linee guida sulla creazione dei pacchetti.

Il comando **dh_make** creerà il modello dei file di configurazione nella directory `debian`. Molti di questi file terminano con l'estensione `.ex`. In altri inoltre il nome del file viene preceduto dal nome del pacchetto in binario, come *`pacchetto`*. Si dia uno sguardo a tutti i differenti file di configurazione. [1]

Alcuni modelli di file di configurazione per `debhelper` non possono essere creati dal comando **dh_make**. In questo caso, è necessario crearlo con un editor.

Se si vuole o si ha bisogno di attivare uno di questi file, si effettuino le seguenti operazioni:

- si rinomini il file modello rimuovendo l'estensione `.ex` o `.EX` se presente;
- si rinominino i file di configurazione con il nome usato dall'attuale pacchetto binario al posto di *`pacchetto`*;
- si modifichi il contenuto del file in base alle proprie esigenze;
- si elimini il file modello di cui non si ha più bisogno;
- si modifichi il file `control` (si veda Sezione 4.1), se necessario;
- si modifichi il file delle `regole` (si veda Sezione 4.4), se necessario.

Qualsiasi file di configurazione di `debhelper` senza prefisso del *`pacchetto`*, come nel file `install` vengono applicati al primo pacchetto binario. Quando ci sono molti pacchetti binari le loro configurazioni possono essere specificate aggiungendo il loro prefisso al nome del file di configurazione come *`pacchetto-1`*`.install`, *`pacchetto-2`*`.install`, ecc.

5.1 `README.Debian`

Ogni ulteriore dettaglio o discrepanza tra il pacchetto originale e la versione Debian dovrebbe essere documentato qui.

dh_make ne crea uno predefinito; ecco come appare:

```
gentoo for Debian
-----------------
<possible notes regarding this package - if none, delete this file>
 -- Josip Rodin <joy-mg@debian.org>, Wed, 11 Nov 1998 21:02:14 +0100
```

Qui dovrebbero inserire delle brevi informazioni specifiche di Debian. Si veda dh_installdocs(1).

[1] In questo capitolo, per semplicità, i file nella directory `debian` sono indicati senza la directory radice `debian/`, ogni volta che il loro significato è scontato.

5.2 `compat`

Il file `compat` definisce il livello di compatibilità di `debhelper`. Al momento dovrebbe essere impostato a `debhelper` v10 nel modo seguente:

```
$ echo 10 > debian/compat
```

È possibile utilizzare il livello di compatibilità v9 in determinate circostanze per la compatibilità con i sistemi meno recenti. Tuttavia, l'utilizzo di qualsiasi livello inferiore al v9 non è raccomandato e dovrebbe essere evitato per i nuovi pacchetti.

5.3 `conffiles`

Una delle cose più fastidiose riguardanti il software accade quando si spende una grande quantità di tempo e di sforzi per personalizzare un programma solo per vedere un aggiornamento spazzare via tutte le modifiche fatte. Debian risolve questo problema marcando i file di configurazione come conffiles. [2] Quando si aggiorna un pacchetto, verrà richiesto se si vogliono mantenere o meno i vecchi file di configurazione.

dh_installdeb(1) marcherà *automaticamente* ogni file che si trova nella directory `/etc` come conffiles, così che se il programma ha soli file di configurazione in quella directory, non ci sarà bisogno di specificarli in questo file. Per la maggior parte dei tipi di pacchetto, l'unico posto in cui si dovrebbero trovare i conffile è all'interno di `/etc` e quindi non c'è bisogno che questo file esista.

Se il programma creato utilizza file di configurazione e li sovrascrive in automatico, è meglio non rendere questi ultimi dei file di configurazione perché **dpkg** chiederà ogni volta agli utenti di verificare i cambiamenti.

Se il programma di cui si sta creando il pacchetto richiede ad ogni utente di modificare i file di configurazione nella directory `/etc`, ci sono principalmente due modi per non renderli conffiles e quindi di mantenere **dpkg** tranquillo:

- Si può creare un link simbolico nella directory `/etc` che punti ad un file nella directory `/var` generato dagli script del manutentore.
- Si crei un file generato dagli script del manutentore nella directory `/etc`.

Per maggiori informazioni sugli script del manutentore, si veda Sezione 5.18.

5.4 *`pacchetto`*`.cron.*`

Se il pacchetto creato richiede che vengano programmate delle operazioni per funzionare correttamente, si può utilizzare questo file per lo scopo. Si possono programmare delle operazioni in modo tale che vengano eseguite su base oraria, giornaliera, settimanale, mensile o che vengano eseguite alternativamente in qualsiasi momento si voglia. I file saranno:

- *`pacchetto`*`cron.hourly` - Installato come `/etc/cron.hourly/`*`pacchetto`*: viene eseguito una volta ogni ora.
- *`pacchetto`*`cron.daily` - Installato as `/etc/cron.daily/`*`pacchetto`*: viene eseguito una volta al giorno.
- *`pacchetto`*`cron.weekly` - Installato come `/etc/cron.weekly/`*`pacchetto`*: viene eseguito una volta a settimana.
- *`pacchetto`*`cron.monthly` - Installato come `/etc/cron.monthly/`*`pacchetto`*: viene eseguito una volta al mese.
- *`pacchetto`*`cron.d` - Installato come `/etc/cron.d/`*`pacchetto`*: per qualsiasi altro periodo.

Molti di questi file sono script di shell, fatta eccezione di *`pacchetto`*`.cron.d` che segue il formato di crontab(5).

Nessun esplicito file `cron.*` è necessario per impostare la rotazione dei log; a questo proposito si veda dh_installlogrotate(1) e logrotate(8).

[2] Vedere dpkg(1) e il manuale delle policy Debian, "D.2.5 Conffiles" (http://www.debian.org/doc/debian-policy/ap-pkg-controlfields.html#s-pkg-f-Conffiles) .

5.5 dirs

Questo file specifica le directory di cui si ha bisogno ma che non vengono create nella normale procedura di installazione (`make install DESTDIR=...` chiamata da `dh_auto_install`). Questo generalmente indica la presenza di un problema nel `Makefile`.

I file elencati nel file `install` non hanno bisogno che le directory vengano create prima. Si veda Sezione 5.11.

Si raccomanda di provare prima ad eseguire l'installazione ed utilizzare questo file solo se si incorre in problemi. Si noti che il carattere slash non precede il nome delle directory elencate nel file `dirs`.

5.6 *pacchetto*.doc-base

Se il pacchetto generato ha altra documentazione oltre alle pagine di manuale e di info, dovrebbe essere utilizzato il file `doc-base` per segnalarla in modo che l'utente possa trovarla con, ad esempio dhelp(1), dwww(1) o doccentral(1).

Questa documentazione è solitamente costituita da documenti HTML, file PS e PDF, disponibili in `/usr/share/doc/`*`nomepacchett`*

Questo è il modo in cui il file doc-base di `gentoo`, `gentoo.doc-base`, appare:

```
Document: gentoo
Title: Gentoo Manual
Author: Emil Brink
Abstract: This manual describes what Gentoo is, and how it can be used.
Section: File Management
Format: HTML
Index: /usr/share/doc/gentoo/html/index.html
Files: /usr/share/doc/gentoo/html/*.html
```

Per informazioni sul formato del file si veda install-docs(8) ed la copia locale del manuale Debian `doc-base`, reperibile con il pacchetto `doc-base`.

Per ulteriori dettagli su come installare documentazione aggiuntiva, si veda in Sezione 3.3.

5.7 docs

Questo file specifica il nome dei file di documentazione che si possono avere dh_installdocs(1) li installa nella directory temporanea automaticamente.

Normalmente, questo file includerà tutti i file esistenti nella directory dei sorgenti di più alto livello che sono chiamati `BUGS`, `README*`, `TODO` ecc.

Per il pacchetto `gentoo`, sono stati inclusi anche altri files:

```
BUGS
CONFIG-CHANGES
CREDITS
NEWS
README
README.gtkrc
TODO
```

5.8 emacsen-*

Se il pacchetto generato contiene file Emacs che possono essere compilati al momento dell'installazione, possono essere usati per impostarli.

Tali file sono installati nella directory temporanea con dh_installemacsen(1).

Se non se ne ha bisogno, possono essere eliminati.

5.9 *pacchetto*.examples

Il comando dh_installexamples(1) installa i file e le directory elencati al suo interno come file di esempio.

5.10 *pacchetto*.init e *pacchetto*.default

Se il pacchetto generato è un demone che deve partire all'avvio del sistema, hai chiaramente ignorato le mie raccomandazioni iniziali, vero? :-)

Il file *pacchetto*.init viene installato in /etc/init.d/*pacchetto* ed è lo script che avvia e stoppa il demone. La sua struttura abbastanza generica è fornita dal comando **dh_make** come init.d.ex. Probabilmente si dovrà rinominare e modificare un bel po', cercando di assicurarsi di fornire degli header che rispettino Linux Standard Base (http://www.linuxfoundation.org/collaborate/workgroups/lsb) (LSB). Il file viene installato nella directory temporanea con dh_installinit(1).

The *package*.default file will be installed as /etc/default/*package*. This file sets defaults that are sourced by the init script. This *package*.default file is most often used to set some default flags or timeouts. If your init script has certain configurable features, you can set them in the *package*.default file, instead of in the init script itself.

Se il programma originale ha un file di init si può decidere di utilizzarlo o meno. Se non viene utilizzato quello script init.d allora ne dovrà essere creato uno nuovo in debian/*pacchetto*.init. Comunque se lo script di init originale sembra funzionare bene ed installa tutto nella corretta destinazione, si devono comunque impostare i link simbolici rc*. Per fare questo si deve sovrascrivere **dh_installinit** nel file rules con le seguenti righe:

```
override_dh_installinit:
        dh_installinit --onlyscripts
```

Se non si ha bisogno di tutto ciò, si possono rimuovere i file.

5.11 install

Se ci sono dei file che devono essere installati nel pacchetto ma con il normale comando make install non vengono elaborati, i nomi di quest'ultimi e le cartelle di destinazione vanno messe in questo file install. Tali file vengono installati con il comando dh_install(1).[3] Andrebbe innanzitutto controllato che non ci sia uno strumento più specifico da poter utilizzare. Per esempio, i documenti dovrebbero essere elencati nel file docs e non in questo file.

Nel file install compare una riga per ogni file installato, contenente il nome del file (relativo alla directory in cui avviene la costruzione del pacchetto), uno spazio e la directory di installazione (relativa alla posizione del file install). Un esempio dell'utilizzo è quando un file binario src/*bar* non è stato installato, in questo caso il file install dovrebbe essere:

```
src/bar usr/bin
```

Questo significa che quando il pacchetto verrà installato, ci sarà un file eseguibile /usr/bin/*bar*.

Alternativamente, questo file install può presentare il nome del file senza la directory di installazione solo quando il percorso relativo della directory non cambia. Questo formato è solitamente utilizzato per grandi pacchetti che organizzano i risultati della costruzione in pacchetti multipli utilizzando *pacchetto-1*.install, *pacchetto-2*.install, ecc.

Il comando **dh_install** andrà a controllare nella cartella debian/tmp alla ricerca di file, se non ne trova nella directory corrente (or in qualsiasi altro posto si sia indicato di guardare utilizzando --sourcedir).

5.12 *pacchetto*.info

Se il pacchetto generato ha delle pagine info, queste andrebbero installate utilizzando il comando dh_installinfo(1) ed elencandole nei file *pacchetto*.info.

[3] Questo comando rimpiazza il comando dh_movefiles(1), ormai deprecato, che veniva configurato dal file files.

5.13 *pacchetto*.links

Se è necessario creare dei collegamenti simbolici aggiuntivi nella directory utilizzata per la creazione del pacchetto, si dovrebbero installare, come maintainer del pacchetto, utilizzando dh_link(1) con il path per esteso della sorgente e della destinazione dei file nel file *package*.links.

5.14 {*pacchetto*.,source/}lintian-overrides

Se il pacchetto lintian segnala degli errori di diagnostica in un caso in cui esista una policy Debian che ammette delle eccezioni alle regole, si possono utilizzare i file *pacchetto*.lintian-overrides o source/lintian-overrides per inibire le segnalazioni. Si legga il manuale utente di Lintian (https://lintian.debian.org/manual/index.html) e si eviti di farne un uso eccessivo.

Il file *pacchetto*.lintian-overrides è utilizzato per il pacchetto binario chiamato *pacchetto* ed è installato in usr/share/lintian/overrides/*pacchetto* dal comando **dh_lintian**.

Il file source/lintian-overrides è utilizzato per il pacchetto sorgente. Questo non viene installato.

5.15 manpage.*

I programmi creati dovrebbero avere una pagina di manuale. In caso contrario bisogna crearla. Il comando **dh_make** crea diversi file modello per la pagina di manuale. Questi devono essere copiati e modificati per ogni comando senza pagina di manuale. Si faccia attenzione a rimuovere i file modello non utilizzati.

5.15.1 manpage.1.ex

Le pagine del manuale sono solitamente scritte per nroff(1). Anche il file modello manpage.1.ex è scritto per **nroff**. Si veda la pagina di manuale man(7) per una breve descrizione su come modificare un file del genere.

L'ultimo file delle pagine del manuale dovrebbe includere il nome del programma che si sta documentando, quindi verrà rinominato da manpage a gentoo. Il nome del file include anche .1 come primo suffisso, il che sta ad indicare che la sezione della pagina del manuale è relativa ad un comando dell'utente. Si verifichi che questa sezione sia quella corretta. Qui di seguito viene presentata una breve lista delle sezioni delle pagine del manuale:

Sezione	Descrizione	Note
1	Comandi utente	Comandi eseguibili o script
2	Chiamate di sistema	Funzioni fornite dal kernel
3	Chiamate alla libreria	Funzioni delle librerie di sistema
4	File speciali	Posizionati normalmente in /dev
5	Formati di file	Ad esempio il formato del file /etc/passwd
6	Giochi	Giochi o altri programmi frivoli
7	Pacchetti di macro	Come le macro di **man**
8	Amministrazione del sistema	Programmi di norma eseguibili solo da root
9	Kernel routine	Chiamate non standard e interne

Così la pagina man del pacchetto gentoo dovrebbe chiamarsi gentoo.1. Se non ci fosse alcuna pagina man gentoo.1 nei sorgenti originali, andrebbe creata rinominando il modello manpage.1.ex in gentoo.1 e modificandolo utilizzando le informazioni contenute negli esempi e nei documenti originali.

Si può utilizzare il comando **help2man** per generare una pagina di manuale, priva del risultato di --help and --version, per ogni programma. [4]

[4] Si noti che i segnaposto di **help2man** contengono la documentazione più dettagliata disponibile nel sistema info. Se al comando manca la pagina **info** si dovrebbe modificare manualmente la pagina man generata dal comando **help2man**.

5.15.2 `manpage.sgml.ex`

Se d'altra parte si preferisce scrivere in SGML piuttosto che utilizzare **nroff**, si può utilizzare il modello `manpage.sgml.ex`. Se si procede in questo modo andrà:

- rinominato il file in qualcosa del tipo `gentoo.sgml`.
- installato il pacchetto `docbook-to-man`
- aggiunto `docbook-to-man` alla linea `Build-Depends` nel file `control`
- aggiungere un obiettivo `override_dh_auto_build` al file `rules` :

```
override_dh_auto_build:
        docbook-to-man debian/gentoo.sgml > debian/gentoo.1
        dh_auto_build
```

5.15.3 `manpage.xml.ex`

Se si preferisce l'XML all'SGML, si può utilizzare il modello `manpage.xml.ex`. Se si decide questa modalità si avranno due scelte:

- rinominare il file in qualcosa del tipo `gentoo.1.xml`
- installare il pacchetto `docbook-xsl` e l'elaboratore XSLT come `xsltproc` (recommended)
- aggiungere i pacchetti `docbook-xsl`, `docbook-xml` e `xsltproc` alla linea `Build-Depends` nel file `control`
- aggiungere un obiettivo `override_dh_auto_build` al file `rules` :

```
override_dh_auto_build:
        xsltproc --nonet \
         --param make.year.ranges 1 \
         --param make.single.year.ranges 1 \
         --param man.charmap.use.subset 0 \
         -o debian/ \
http://docbook.sourceforge.net/release/xsl/current/manpages/docbook.xsl\
        debian/gentoo.1.xml
        dh_auto_build
```

5.16 *`pacchetto`*`.manpages`

Se il pacchetto creato presenta delle pagine del manuale, queste andrebbero installate utilizzando il comando dh_installman(1) ed elencandole nei file *`pacchetto`*`.manpages`.

Per installare il file `doc/gentoo.1` come pagine di manuale per il pacchetto `gentoo`, si deve creare il file `gentoo.manpages` contenente:

```
docs/gentoo.1
```

5.17 `NEWS`

Il comando dh_installchangelogs(1) installa questo file.

5.18 {pre,post}{inst,rm}

I files `postinst`, `preinst`, `postrm` e `prerm` [5] vengono chiamati *script del manutentore*. Questi sono script che vengono messi nell'area di controllo del pacchetto e vengono lanciati dal comando **dpkg** quando il pacchetto viene installato, aggiornato o rimosso.

Un nuovo manutentore dovrebbe, se possibile, evitare ogni modifica manuale degli script del manutentore perché potrebbero creare dei problemi. Per maggiori informazioni si guardi nel Manuale delle policy di Debian, 6 'Script di manutenzione del pacchetto e procedure di installazione' (http://www.debian.org/doc/debian-policy/ch-maintainerscripts.html) , e si dia un'occhiata ai file di esempio forniti da **dh_make**.

Se si sono, comunque, creati dei script del manutentore personalizzati per un pacchetto, bisogna essere sicuri di averli provati non solo per **install** e **upgrade**, ma anche per **remove** e **purge**.

Gli aggiornamenti alle nuove versioni dovrebbero essere indolore e non invadenti (gli utenti esistenti non dovrebbero notare gli aggiornamenti a meno di scoprire che vecchi bug sono stati corretti e che vi sono magari delle nuove funzionalità).

Quando l'aggiornamento deve per forza di cose essere invadente (per esempio, file di configurazione sparsi all'interno di diverse cartelle home con strutture totalmente differenti), si può impostare il pacchetto ai valori sicuri predefiniti (esempio, servizi disabilitati) e fornire una valida documentazione come richiesto dalla policy (`README.Debian` e `NEWS.Debian`) come ultima spiaggia. Sarebbe meglio non disturbare l'utente con la nota **debconf** richiamata da questi script del manutentore per gli aggiornamenti.

Il pacchetto `ucf` fornisce una infrastruttura di gestione *simil-conffile* per preservare cambiamenti effettuati dall'utente in file che non possono essere etichettati come *conffile*, come ad esempio quelli gestiti dagli script del manutentore. Questo dovrebbe ridurre al minimo i problemi ad esso associati.

Questi script del manutentore sono delle migliorie di Debian che fanno capire **come mai le persone scelgano Debian**. Bisogna stare molto attenti a non infastidirle a causa loro.

5.19 *package*.symbols

Creare il pacchetto di una libreria non è semplice per un maintainer con poca esperienza e dovrebbe essere evitato. Detto questo, se il pacchetto in questione ha delle librerie, si dovrebbero avere dei file `debian/`*`package`*`.symbols` files. Si veda Sezione A.2.

5.20 TODO

Il comando dh_installdocs(1) installa questo file.

5.21 watch

Il formato del file `watch` è documentato nella pagina di manuale uscan(1). Il file `watch` configura il programma **uscan** (nel pacchetto `devscripts`) per controllare il sito da cui è stato scaricato il sorgente originale. Questo file viene utilizzato pure dal servizio Debian Package Tracker (https://tracker.debian.org/) .

Ecco il suo contenuto:

```
# watch control file for uscan
version=3
http://sf.net/gentoo/gentoo-(.+)\.tar\.gz debian uupdate
```

[5] Nonostante l'utilizzo dell'espressione **bash** `{pre,post}{inst,rm}` per indicare questi file, è necessario utilizzare la sintassi POSIX per questi script di manutenzione per mantenere la compatibilità con la shell di sistema **dash**.

Normalmente con il file `watch`, l'URL `http://sf.net/gentoo` viene scaricato per cercare dei collegamenti del tipo `<a href=...>`. Il nome base (la parte appena dopo l'ultimo `/`) di questi URL sono confrontati con l'espressione regolare Perl (si veda perlre(1)) `gentoo-(.+)\.tar\.gz`. Tra i file che combaciano viene scaricato quello avente il numero di versione più grande e viene avviato il programma **uupdate** per creare un albero dei sorgenti aggiornato.

Sebbene questo sia vero per la maggior parte dei siti, il servizio di scaricamento di SourceForge su http://sf.net è un'eccezione. Quando il file `watch` ha un URL che corrisponde all'espressione regolare perl `^http://sf\.net/`, il programma **uscan** lo sostituisce con `http://qa.debian.org/watch/sf.php/`. Il servizio di reindirizzamento degli URL http://qa.debian.org/ è progettato per mettere a disposizione un sistema stabile per reindirizzare gli URL tipo `http://sf.net/`*`project/tar-name`*`-(.+)\` presenti i nel file `watch`. Questo risolve il problema relativo al cambiamento periodico degli URL di SourceForge.

Se il pacchetto originale dispone di una firma crittografica dell'archivio, si consiglia di verificarne l'autenticità utilizzando l'opzione `pgpsigurlmangle` come descritto in uscan(1).

5.22 `source/format`

Nel file `debian/source/format`, ci dovrebbe essere una unica riga che indichi il formato desiderato per il pacchetto sorgente (controllare dpkg-source(1) per una lista completa). Dopo `squeeze`, dovrebbe scrivere:

- `3.0 (native)` per i pacchetti nativi Debian o
- `3.0 (quilt)`per tutti gli altri.

Il nuovo formato sorgente `3.0 (quilt)` registra le modifiche in una serie di patch **quilt** all'interno di `debian/patches`. Questi cambiamenti vengono poi automaticamente applicati durante l'estrazione del pacchetto sorgente. [6] Le modifiche di Debian sono semplicemente mantenute in un archivio `debian.tar.gz` contenente tutti i file sotto la directory `debian`. Questo nuovo formato supporta l'inclusione di file binari come per esempio le icone PNG del manutentore del pacchetto senza richiedere trucchi.[7]

Quando **dpkg-source** estrae un pacchetto sorgente nel formato `3.0 (quilt)`, applica automaticamente tutte le patch elencate nel file `debian/patches/series`. Si può evitare di applicare le patch alla fine dell'estrazione con l'opzione `--skip-patches`.

5.23 `source/local-options`

Quando si vogliono gestire le attività di pacchettizzazione utilizzando un VCS, di norma si crea un ramo (es. `upstream`) in cui si tiene traccia dei cambiamenti apportanti al sorgente originale, e un altro ramo (es. di solito `master` per Git) in cui si tiene traccia delle modifiche apportate al pacchetto. In fine, è consigliato avere i sorgenti originali senza pach con i file in `debian/*` per il pacchetto Debian, per fare facilitare le operazioni di fusione con il nuovo sorgente originale.

Dopo aver costruito il pacchetto, il sorgente è di norma pachato. È necessario togliere la patch manualmente, eseguendo `dquilt pop -a`, prima di caricarlo nel ramo `master`. Si può automatizzare l'operazione, aggiungendo il file `debian/source/local-options` con contenuto `unapply-patches`. Questo file non è incluso nel sorgente del pacchetto generato, e cambia solamente la modalità di costruzione in locale. Questo file può contenere anche `abort-on-upstream-changes`, (si veda dpkg-source(1)).

```
unapply-patches
abort-on-upstream-changes
```

[6] Si veda DebSrc3.0 (http://wiki.debian.org/Projects/DebSrc3.0) per una serie di informazioni generali riguardanti il passaggio al nuovo formato `3.0 (quilt)` ed ai formati sorgente `3.0 (native)`.

[7] Al momento questo nuovo formato supporta anche molteplici archivi e più metodi di compressione. Questi però esulano dall'obiettivo di questo documento.

5.24 source/options

I file generati automaticamente nell'albero dei sorgenti possono essere molto fastidiosi per la pacchettizzazione, in quanto generano file di patch senza senso e di grandi dimensioni. Ci sono moduli personalizzati come **dh_autoreconf** per facilitare questo problema, come descritto nella Sezione 4.4.3 .

È possibile fornire un'espressione regolare Perl al parametro `--extend-diff-ignore` di dpkg-source(1) per ignorare le modifiche apportate ai file generati automaticamente durante la creazione del pacchetto sorgente.

Come soluzione generale per affrontare il problema dei file generati automaticamente, è possibile memorizzare un parametro **dpkg-source** nel file `source/options` del pacchetto sorgente. Il comando di seguito salterà la creazione dei file di patch per `config.sub`, `config.guess` e `Makefile`.

```
extend-diff-ignore = "(^|/)(config\.sub|config\.guess|Makefile)$"
```

5.25 patches/*

Il vecchio formato sorgente `1.0` creava un singolo, grosso file `diff.gz` file che conteneva i file di manutenzione del pacchetto che stavano in `debian` ed i file di patch del sorgente. Un pacchetto così è un po' difficoltoso da ispezionare per capire successivamente la sequenza delle modifiche. Per questo tale formato non è considerato soddisfacente.

Il nuovo formato dei sorgenti `3.0 (quilt)` salva le patch in file `debian/patches/*` utilizzando il comando **quilt**. Queste patch e gli altri dati del pacchetto che sono contenuti nella directory `debian` è pacchettizzato come file `debian.tar.gz`. Poiché il comando **dpkg-source** è in grado di gestire patch formattate con **quilt** nel formato sorgente `3.0 (quilt)` senza il pacchetto `quilt`, non è necessario il avere `quilt` nel campo `Build-Depends`. [8]

Il comando **quilt** è documentato in quilt(1). Esso registra le modifiche ai sorgenti come una coda di file di patch `-p1` nella directory `debian/patches` e l'albero dei sorgenti non varia al di fuori della directory `debian`. L'ordine delle patch è registrato nel file `debian/patches/series`. Si può applicare (=push), rimuovere(=pop), ed aggiornare le patch con facilità. [9]

Per Capitolo 3, sono state create tre patch in `debian/patches`.

Dal momento che le patch di Debian si trovano in `debian/patches`, ci si assicuri di impostare correttamente il comando **dquilt**, come descritto nella Sezione 3.1.

Quando qualcuno (me compreso) fornisce una patch *`foo`*`.patch` per i sorgenti, allora la modifica del pacchetto sorgente di tipo `3.0 (quilt)`è abbastanza semplice:

```
$ dpkg-source -x gentoo_0.9.12.dsc
$ cd gentoo-0.9.12
$ dquilt import ../foo.patch
$ dquilt push
$ dquilt refresh
$ dquilt header -e
... describe patch
```

Le patch salvate nel nuovo formato sorgente `3.0 (quilt)` devono essere prive di *fuffa*. Bisogna quindi assicurarsi di ciò eseguendo `dquilt pop -a; while dquilt push; do dquilt refresh; done`.

[8] Diversi metodi di mantenimento delle patch set sono stati proposti e sono utilizzati per i pacchetti Debian. Il sistema **quilt** è il sistema di manutenzione consigliato ed utilizzato. Altri includono **dpatch**, **dbs** e **cdbs**. Molte di questi tengono le patch come file `debian/patches/*`.

[9] Se si sta chiedendo ad uno sponsor di caricare il proprio pacchetto, questo tipo di chiara separazione e documentazione dei cambiamenti è molto importante per accelerare la revisione del pacchetto da parte dello sponsor.

Capitolo 6

Costruzione del pacchetto

A questo punto, si dovrebbe essere pronti a creare il pacchetto.

6.1 (ri)Creazione completa

Al fine di (ri)creare un pacchetto in modo appropriato, è necessario assicurarsi di installare

- il pacchetto `build-essential`,
- i pacchetti elencati nel campo `Build-Depends` (vedere Sezione 4.1), e
- i pacchetti elencati nel campo `Build-Depends-indep` (vedere Sezione 4.1).

Adesso ci si sposti nella directory dei sorgenti del programma e si lanci il comando:

```
$ dpkg-buildpackage -us -uc
```

Questo comando creerà i pacchetti binari e sorgenti al posto vostro. Eseguirà le seguenti operazioni:

- pulirà l'albero dei sorgenti (`debian/rules clean`)
- costruirà il pacchetto sorgente (`dpkg-source -b`)
- costruirà il programma (`debian/rules build`)
- costruirà il pacchetto binario (`fakeroot debian/rules binary`)
- crea il file `.dsc`
- crea il file `.changes`, utilizzando **dpkg-genchanges**

Se si è soddisfatti del pacchetto generato, si firmino i file `.dsc` e `.changes` con la chiave GPG privata utilizzando il comando **debsign**. È necessario inserire la propria passphrase segreta due volte. [1]

Per un pacchetto Debian non-nativo, per esempio, `gentoo`, si vedano i seguenti file nella directory superiore (`~/gentoo`) dopo la creazione dei pacchetti:

[1] Questa chiave GPG deve essere firmata da uno sviluppatore Debian per collegarsi alla rete di fiducia e bisogna essere registrati al portachiavi Debian (http://keyring.debian.org) . In questo modo i pacchetti caricati potranno essere accettati ed inseriti negli archivi Debian. Vedere Creare una nuova chiave GPG (http://keyring.debian.org/creating-key.html) e Debian Wiki on Keysigning (http://wiki.debian.org/Keysigning) .

- `gentoo_0.9.12.orig.tar.gz`

  Questo è il codice sorgente originale, semplicemente rinominato in modo da aderire allo standard Debian. Da notare che questo è stato creato inizialmente con il comando `dh_make -f ../gentoo-0.9.12.tar.gz`.

- `gentoo_0.9.12-1.dsc`

  Questo è un sommario del contenuto del codice sorgente. Questo file è generato dal file `control`, ed è usato quando si decomprime il sorgente con dpkg-source(1).

- `gentoo_0.9.12-1.debian.tar.gz`

  Questo file compresso contiene il contenuto della directory `debian`. Ogni modifica effettuata al codice sorgente originale, verrà memorizzata come patch di **quilt** in `debian/patches`.

  Se qualcun altro volesse ri-creare il pacchetto da zero, potrebbe farlo facilmente usando i suddetti tre file. La procedura di estrazione è banale: basta copiare i tre file da qualche parte ed eseguire `dpkg-source -x gentoo_0.9.12-1.dsc`. [2]

- `gentoo_0.9.12-1_i386.deb`

  Questo è il pacchetto binario completo. Si può usare **dpkg** per installarlo e rimuoverlo, come per ogni altro pacchetto.

- `gentoo_0.9.12-1_i386.changes`

  Questo file descrive tutte le modifiche effettuate nella revisione corrente del pacchetto; è usata dai programmi di manutenzione dell'archivio FTP di Debian, per installare i pacchetti binari e sorgenti. È generato parzialmente dal contenuto del file `changelog` e dal file `.dsc`.

  Quando si lavora sul pacchetto, potrebbero cambiare il funzionamento del programma, o potrebbero venire introdotte nuove funzionalità. Chi scaricherà il pacchetto, potrà controllare questo file per vedere velocemente quali sono i cambiamenti. I programmi di manutenzione dell'archivio Debian invieranno anche i contenuti di questo file alla mailing list debian-devel-changes-@lists.debian.org (http://lists.debian.org/debian-devel-changes/) .

I file `gentoo_0.9.12-1.dsc` e `gentoo_0.9.12-1_i386.changes` devono essere firmati usando il comando **debsign** con la propria chiave privata GPG presente nella directory `~/.gnupg/`, prima di caricarli nell'archivio FTP Debian. La firma GPG è la prova che questi file sono veramente stati generati da te, utilizzando la chiave GPG pubblica.

Il comando **debsign** può essere eseguito per firmare con l'ID della chiave GPG specificata, nel file `~/.devscripts`, come seguie (utile in caso di sponsoring di pacchetti):

```
DEBSIGN_KEYID=Your_GPG_keyID
```

Le lunghe stringhe di numeri nei file `.dsc` e `.changes` sono codici di controllo SHA1/SHA256 per i file menzionati. Chi scarica questi file, può controllarli con sha1sum(1), o sha256sum(1) e se i numeri non corrispondessero saprebbe che il file relativo è corrotto, o è stato alterato.

6.2 Auto-costruzione

Debian supporta molti port (http://www.debian.org/ports/) tramite la autobuilder network (http://www.debian.org/devel/buildd/) , su cui sono in esecuzione i demoni di **buildd** su molti computer con architetture differenti. Anche se non sarà necessario fare questo da soli, si dovrebbe essere consapevoli di quello che succederà ai pacchetti. Si vedrà, in maniera non approfondita, come i pacchetti vengono ricostruiti per architetture differenti. [3]

I pacchetti con `Architecture: any`, verranno ricostruiti dal sistema di auto-costruzione. Ci si assicuri di avere installato

- il pacchetto `build-essential`, e
- i pacchetti elencati nel campo `Build-Depends` (vedere Sezione 4.1).

[2] È possibile evitare di applicare la patch **quilt**, nel formato sorgente `3.0 (quilt)`, aggiungendo il parametro `--skip-patches` al comando di estrazione. In alternativa, è possibile eseguire `dquilt pop -a` dopo le normali operazioni.

[3] L'attuale sistema di auto-costruzione è molto più complicato di come è qui documentato. Tali dettagli esulano dallo scopo del documento.

Dopo si può eseguire il comando seguente nella directory dei sorgenti:

```
$ dpkg-buildpackage -B
```

Questo comando creerà i pacchetti binari e sorgenti al posto vostro. Eseguirà le seguenti operazioni:

- pulirà l'albero dei sorgenti (`debian/rules clean`)
- costruirà il programma (`debian/rules build`)
- costruirà il pacchetto binario per una specifica architettura (`fakeroot debian/rules binary-arch`)
- firmerà il file sorgente `.dsc` file, usando **gpg**
- creerà e firmerà il file di upload `.changes` file, usando **dpkg-genchanges** e **gpg**

È questo il motivo per il quale si vede il proprio pacchetto per altre architetture.

Anche se i pacchetti sono elencati nel campo `Build-Depends-Indep`, per la normale creazione del pacchetto, devono comunque essere installati (vedere Sezione 6.1), invece per il sistema di auto-costruzione non è necessario installarli dato che costruisce solamente pacchetti binari per una specifica architettura. [4] Questa distinzione tra la normale pacchettizzazione e il sistema di auto-costruzione determina se i pacchetti richiesti devono essere registrati nei campi `Build-Depends` o `Build-Depends-Indep` nel file `debian/control` (vedere Sezione 4.1).

6.3 Il comando debuild

È possibile automatizzare ulteriormente il processo di creazione del pacchetto, eseguito con il comando **dpkg-buildpackage**, utilizzando il comando **debuild**. Vedere debuild(1).

Il comando **debuild** esegue il comando **lintian** per effettuare un controllo statico dopo la creazione del pacchetto Debian. Il comando **lintian** può essere personalizzato come segue, nel file `~/.devscripts`:

```
DEBUILD_DPKG_BUILDPACKAGE_OPTS="-us -uc -I -i"
DEBUILD_LINTIAN_OPTS="-i -I --show-overrides"
```

Si possono ripulire i sorgenti e ricreare il pacchetto da un account utente, semplicemente con:

```
$ debuild
```

È possibile ripulire l'albero dei sorgenti semplicemente con:

```
$ debuild -- clean
```

6.4 Il pacchetto `pbuilder`

Il pacchetto `pbuilder` è molto utile per verificare le dipendenze di creazione del pacchetto da un ambiente (**chroot**) di compilazione sano e minimale. [5] Questo assicura di compilare i sorgenti in maniera pulita, usando la distribuzione `sid` un compilatore automatico (auto-builder) per differenti architetture ed evita i bug FTBFS (Fails To Build From Source) di severità seria, che sono sempre di categoria RC (Critici per il Rilascio). [6]

Si configuri il pacchetto `pbuilder` come segue:

[4] Diversamente dal pacchetto `pbuilder`, l'ambiente **chroot** sotto il pacchetto `sbuild`, usato dal sistema di auto-costruzione, non forza la creazione di un sistema minimale, e potrebbe lasciare installati molti pacchetti.

[5] `pbuilder` è ancora in evoluzione, si dovrebbe controllare l'attuale configurazione consultando la documentazione ufficiale più recente.

[6] Vedere http://buildd.debian.org/ per maggiori informazioni sull'auto-costruzione dei pacchetti Debian.

- impostare il permesso di scrittura per l'utente alla directory `/var/cache/pbuilder/result`.
- creare una directory, ad es. */var/cache/pbuilder/hooks*, con i permessi di scrittura per l'utente per potergli inserire degli script di hook.
- configurare il file `~/.pbuilderrc` o `/etc/pbuilderrc` in modo che includa le seguenti righe.

```
AUTO_DEBSIGN=${AUTO_DEBSIGN:-no}
HOOKDIR=/var/cache/pbuilder/hooks
```

Si avvii `pbuilder` per costruire l'ambiente **chroot** locale, come segue:

```
$ sudo pbuilder create
```

Se si hanno già i pacchetti sorgenti, eseguire i seguenti comandi nella directory in cui si trovano i file *foo*`.orig.tar.gz`, *foo*`.debian.tar.gz`, e *foo*`.dsc` per aggiornare l'ambiente **chroot** di `pbuilder` e per costruirci dentro il pacchetto binario:

```
$ sudo pbuilder --update
$ sudo pbuilder --build foo_version.dsc
```

Il nuovo pacchetto, senza firme GPG, sarà creato nella directory `/var/cache/pbuilder/result/` e non sarà assegnato all'utente root.

Le firme GPG sui file `.dsc .changes` possono essere generate come segue:

```
$ cd /var/cache/pbuilder/result/
$ debsign foo_version_arch.changes
```

Se si ha l'albero dei sorgenti aggiornato, ma non si sono generati i rispettivi pacchetti sorgenti, eseguire i seguenti comandi nella directory dei sorgenti in cui si trova il file `debian`:

```
$ sudo pbuilder --update
$ pdebuild
```

È possibile accedere all'ambiente **chroot** con il comando `pbuilder --login --save-after-login` e configurarlo come si vuole. Questo ambiente può essere salvato, semplicemente uscendo dalla shell con `^D` (Control-D).

L'ultima versione del programma **lintian** può essere eseguita nell'ambiente `chroot`, usando gli script di hook */var/cache/pbuilder/hooks/*`B90lintian`, configurati come segue: [7]

```
#!/bin/sh
set -e
install_packages() {
        apt-get -y --allow-downgrades install "$@"
        }
install_packages lintian
echo "+++ lintian output +++"
su -c "lintian -i -I --show-overrides /tmp/buildd/*.changes" - pbuilder
# use this version if you don't want lintian to fail the build
#su -c "lintian -i -I --show-overrides /tmp/buildd/*.changes; :" - pbuilder
echo "+++ end of lintian output +++"
```

È necessario avere accesso all'ultima versione di `sid` per poter costruire correttamente i pacchetti per `sid`. In pratica `sid` potrebbe avere dei problemi che rendono poco consigliabile migrare l'intero sistema. In questo caso il pacchetto `pbuilder` può essere molto di aiuto.

Potrebbe essere necessario aggiornare i pacchetti `stable` dopo il loro rilascio per `stable-proposed-updates`, `stable/update` ecc. [8] Per questa ragione se si sta utilizzando un sistema `sid` non è una buona scusa per non aggiornarli tempestivamente. Il pacchetto `pbuilder` aiuta ad accedere agli ambienti di quasi tutte le distribuzioni derivate da Debian con la stessa architettura di CPU.

Vedere http://www.netfort.gr.jp/~dancer/software/pbuilder.html, pdebuild(1), pbuilderrc(5), e pbuilder(8).

[7] Ciò presuppone il settaggio `HOOKDIR=/var/cache/pbuilder/hooks`. È possibile trovare numerosi esempi di script di hook, nella directory `/usr/share/doc/pbuilder/examples`.

[8] Ci sono alcune restrizioni per tali aggiornamenti del pacchetto `stable`.

6.5 Il comando git-buildpackage ed altri simili

Se l'autore originale utilizza un sistema di controllo di versione (VCS) [9] per gestire il proprio codice, si dovrebbe prendere in considerazione di usarlo. Questo rende molto più semplice la fusione e la raccolta di patch dai sorgenti originali. Ci sono diversi pacchetti di script adatti alla costruzione di pacchetti Debian per ogni sistema VCS.

- `git-buildpackage`: una suite per aiutare con i pacchetti Debian nei repository Git.
- `svn-buildpackage`: programmi di supporto per mantenere i pacchetti Debian con Subversion.
- `cvs-buildpackage`: una serie di script per i pacchetti Debian per gli alberi di sorgenti sotto CVS.

L'utilizzo di `git-buildpackage` sta diventando molto popolare per gli sviluppatori Debian,questo permette di gestire i pacchetti Debian con il server Git su alioth.debian.org (http://alioth.debian.org/) . [10] Questo pacchetto offre molti comandi che *automatizzano* le procedure di pacchettizzazione:

- gbp-import-dsc(1): import a previous Debian package to a Git repository.
- gbp-import-orig(1): import a new upstream tar to a Git repository.
- gbp-dch(1): generate the Debian changelog from Git commit messages.
- git-buildpackage(1): costruisce i pacchetti Debian da un repository Git.
- git-pbuilder(1): costruisce i pacchetti Debian da un repository Git, utilizzando **pbuilder**/**cowbuilder**.

Questi comandi utilizzano 3 "branch" per tenere traccia dell'attività sulla pacchettizzazione:

- `main` per l'albero dei sorgenti dei pacchetti Debian.
- `upstream` per l'albero dei sorgenti originali.
- `pristine-tar` per i tarball dei sorgenti originali generati dall'opzione `--pristine-tar`.[11]

È possibile configurare `git-buildpackage` utilizzando il file `~/.gbp.conf`. Vedere gbp.conf(5). [12]

6.6 Ricostruzione veloce

Con un pacchetto di grandi dimensioni, si potrebbe non voler ricostruire tutto da zero, ogni volta che si modifica un dettaglio in `debian/rules`. Per effettuare delle prove, si può creare un file `.deb`, senza ricompilare i sorgenti originali, come segue: [13]:

```
$ fakeroot debian/rules binary
```

[9] Vedere Version control systems (http://www.debian.org/doc/manuals/debian-reference/ch10#_version_control_systems) per più informazioni.

[10] Debian wiki Alioth (http://wiki.debian.org/Alioth) spiega come usare il servizio alioth.debian.org (http://alioth.debian.org/) .

[11] L'opzione `--pristine-tar` esegue il comando **pristine-tar** che può rigenerare un copia esatta del tarball dei sorgenti originali, utilizzando solo un piccolo file binario delta e i contenuti del tarball, che normalmente sono conservati nel branch `upstream` del VCS.

[12] Ecco alcune risorse web, per gli utenti esperti.

- Creare pacchetti Debian Packages con git-buildpackage (`/usr/share/doc/git-buildpackage/manual-html/gbp.html`)
- debian packages in git (https://honk.sigxcpu.org/piki/development/debian_packages_in_git/)
- Using Git for Debian Packaging (http://www.eyrie.org/~eagle/notes/debian/git.html)
- git-dpm: Debian packages in Git manager (http://git-dpm.alioth.debian.org/)
- Using TopGit to generate quilt series for Debian packaging (http://git.debian.org/?p=collab-maint/topgit.git;a=blob_plain;f=debian/HOWTO-tg2quilt;hb=HEAD)

[13] Le variabili d'ambiente, che sono normalmente impostate con dei valori corretti, non sono utilizzati in questa modalità. Mai creare dei pacchetti, che poi andranno caricati, utilizzando il metodo **veloce**.

Oppure, semplicemente controllando se si costruisce o no:

```
$ fakeroot debian/rules build
```

Una volta completati i vari aggiustamenti, bisogna ricordarsi di ricostruire il pacchetto usando la giusta procedura. Si potrebbe non essere in grado di caricare il pacchetto correttamente se si prova con dei file `.deb` creati in questo modo.

6.7 Struttura gerarchica del comando

Ecco un breve riassunto di come molti comandi per creare i pacchetti si incastrano nella gerarchia di comando. Ci sono molti modi per fare la stessa cosa.

- `debian/rules` = script responsabile della costruzione del pacchetto
- **dpkg-buildpackage** = il cuore dello strumento di costruzione del pacchetto
- **debuild** = **dpkg-buildpackage** + **lintian** (creare il pacchetto utilizzando un ambiente con le variabili controllate)
- **pbuilder** = il cuore dello strumento chroot dell'ambiente Debian
- **pdebuild** = **pbuilder** + **dpkg-buildpackage** (costruito in chroot)
- **cowbuilder** = velocizza l'esecuzione di **pbuilder**
- **git-pbuilder** = una sintassi a linea di comando facilitata per **pdebuild** (usato da **gbp buildpackage**)
- **gbp** = gestisce il sorgente Debian utilizzando un repository git
- **gbp buildpackage** = **pbuilder** + **dpkg-buildpackage** + **gbp**

Sebbene l'uso di comandi di alto livello come **gbp buildpackage** e **pbuilder** assicura l'ambiente ideale per la costruzione del pacchetto, è essenziale comprendere come i comandi di basso livello `debian/rules` e **dpkg-buildpackage** vengono richiamati da loro.

Capitolo 7

Controllare il pacchetto per errori

Ci sono alcune tecniche da sapere per controllare se un pacchetto ha degli errori prima di caricarlo negli archivi pubblici.

Effettuare dei test su altre macchine oltre a quella con cui si è sviluppato è una buona idea. Si deve inoltre fare attenzione agli avvisi ed agli errori per tutti i test che verranno qui descritti.

7.1 Modifiche sospette

Se si trova un nuovo file di patch generato automaticamente, come `debian-changes-*` nella directory `debian/patches` dopo aver costruito un pacchetto Debian non-nativo nel formato `3.0 (quilt)`, è probabile che è stato cambiato qualche file per caso o che gli script di build hanno modificato il sorgente originale. Se è si tratta di un errore del genere, lo si risolva. Se è causato dallo script di build, si cerchi la causa principale del problema con **dh-autoreconf** come mostrato in Sezione 4.4.3 o si cerchi di aggirare il problema con `source/options` come mostrato in Sezione 5.24.

7.2 Verifica dell'installazione di un pacchetto

Bisogna verificare che il pacchetto si installi senza problemi. Il comando debi(1) aiuta a testare l'installazione di tutti i pacchetti binari generati.

```
$ sudo debi gentoo_0.9.12-1_i386.changes
```

Per prevenire eventuali problemi di installazione su sistemi diversi, bisogna assicurarsi che non ci siano file in conflitto con altri pacchetti esistenti utilizzando il file `Contents-`*`i386`* scaricato dall'archivio Debian. Il comando **apt-file** può tornare utile a questo scopo. Se ci sono file che si sovrappongono, si prega di prendere delle misure per evitare l'insorgere del problema, rinominando il file, spostando il file in un pacchetto separato e configurarlo come dipendenza sui vari pacchetti che lo richiedono, utilizzando meccanismi alternativi (si veda update-alternatives(1)) che permettendo di coordinarsi con i manutentori degli altri pacchetti interessati o settano la voce `Conflicts` nel file `debian/control`.

7.3 Verifica degli script del manutentore di un pacchetto

Tutti gli script del manutentore (ad esempio i file, `preinst`, `prerm`, `postinst`, e `postrm`) sono difficili da scrivere correttamente a meno che non siano stati generati automaticamente dai programmi di `debhelper`. Si consiglia pertanto di non utilizzarli se non si ha sufficiente esperienza come manutentore (si veda Sezione 5.18).

Se il pacchetto utilizza questi particolari script del manutentore, ci si assicuri di effettuare delle prove non solo per l'operazione di install, ma anche per il remove, purge, e l'upgrade. Molti bug degli script del manutentore vengono fuori quando i pacchetti sono rimossi o viene applicato il purge. Si utilizzi il comando **dpkg** nel seguente modo per testarli:

```
$ sudo dpkg -r gentoo
$ sudo dpkg -P gentoo
$ sudo dpkg -i gentoo_version-revision_i386.deb
```

Questa operazione si dovrebbe effettuare con delle sequenze di questo tipo:

- installazione della versione precedente (se necessaria).
- aggiornamento dalla versione precedente.
- ritorno alla versione precedente (opzionale).
- applicazione del purge.
- installazione del nuovo pacchetto.
- rimozione del pacchetto.
- reinstallazione del pacchetto.
- applicazione del purge.

Se si sta creando il primo pacchetto, andrebbero creati dei pacchetti fittizi con diversi numeri di versione per testare il pacchetto originale in anticipo e prevenire problemi futuri.

Si tenga in mente che se il pacchetto è stato già rilasciato in Debian, le persone spesso effettueranno un aggiornamento a quest'ultimo a partire dall'ultima versione disponibile su Debian. Si ricordi di testare gli aggiornamenti anche a partire da questa versione.

Anche se il ritorno ad una versione precedente non è ufficialmente supportato, sarebbe buona abitudine supportarlo.

7.4 Utilizzare `lintian`

Si esegua lintian(1) sul file `.changes`. Il comando **lintian** esegue molti script di test alla ricerca dei più comuni errori di pacchettizzazione. [1]

```
$ lintian -i -I --show-overrides gentoo_0.9.12-1_i386.changes
```

Ovviamente va rimpiazzato il nome con quello del file `.changes` generato per il pacchetto. I risultati del comando **lintian** vengono qui elencati di seguito:

- `E:` errore; una violazione certa di una policy o un errore di pacchettizzazione.
- `W:` attenzione; una possibile violazione di policy o un errore della pacchettizzazione.
- `I:` informazione; un'informazione su alcuni aspetti della pacchettizzazione.
- `N:` nota; un messaggio dettagliato per aiutare nell'analisi degli errori.
- `O:` per sovrascrivere: il messaggio verrà sovrascritto dal file `lintian-overrides`, ma potrà essere visualizzato con l'opzione `--show-overrides`.

Quando vengono generati degli avvertimenti, si imposti il pacchetto in modo tale da evitarli o si verifichi che tali avvertimenti non siano indicativi di un errore. In quest'ultimo caso, si impostino i file `lintian-overrides` come descritto in Sezione 5.14.

Si noti che si può costruire il pacchetto con **dpkg-buildpackage** ed eseguire **lintian** su di esso in una sola volta con debuild(1) o con pdebuild(1).

[1] Non c'è bisogno di fornire l'opzione **lintian** `-i -I --show-overrides` se si è personalizzato il file `/etc/devscripts.conf` o il file `~/.devscripts` come descritto in Sezione 6.3.

7.5 Il comando debc

Si possono elencare i file nel pacchetto binario Debian con il comando debc(1).

```
$ debc package.changes
```

7.6 Il comando debdiff

Si può confrontare il contenuto dei file in due pacchetti sorgente Debian con il comando debdiff(1).

```
$ debdiff old-package.dsc new-package.dsc
```

Si possono anche confrontare le liste di file in due set di pacchetti binari Debian con il comando debdiff(1).

```
$ debdiff old-package.changes new-package.changes
```

Questi comandi sono utili per vedere cosa sia cambiato nei pacchetti sorgente, se un file sia stato spostato inavvertitamente o rimosso dai pacchetti, e se altri cambiamenti non intenzionali siano stati fatti durante l'aggiornamento dei pacchetti binari.

7.7 Il comando interdiff

Si possono confrontare due file `diff.gz` con il comando interdiff(1). Questo è utile per verificare che il manutentore non abbia inavvertitamente fatto dei cambiamenti ai sorgenti durante il processo di aggiornamento dei pacchetti nel vecchio formato sorgente `1.0`.

```
$ interdiff -z old-package.diff.gz new-package.diff.gz
```

Il nuovo formato sorgente `3.0` salva i cambiamenti in file di patch multipli come descritto in Sezione 5.25. È possibile tracciare i cambiamenti di ogni file `debian/patches/*` usando anche **interdiff**.

7.8 Il comando mc

Molte delle operazioni di ispezione dei file possono essere rese più semplici utilizzando un gestore dei file come mc(1), che permette di navigare non solo il contenuto dei pacchetti in formato `*.deb` ma anche degli `*.udeb`, `*.debian.tar.gz`, `*.diff.gz`, e dei file `*.orig.tar.gz`.

Si faccia attenzione ad ulteriori file non necessari o vuoti, sia nel pacchetto binario che in quello sorgente. Spesso non vengono ripuliti correttamente; si aggiusti il file `rules` per riparare a questo problema.

Capitolo 8

Aggiornamento del pacchetto

Una volta rilasciato un pacchetto, ci sarà presto bisogno di aggiornarlo.

8.1 Nuova revisione Debian

Si supponga che sia stato compilato il bug report `#654321` per il pacchetto creato, e che questo descriva un problema che si può risolvere. Qui è descritto come creare una nuova revisione del pacchetto Debian:

- Se la modifica deve essere registrata come una nuova patch, si seguano queste istruzioni:
  - `dquilt new` *`bugname.patch`* per impostare il nome della patch;
  - `dquilt add` *`buggy-file`* per indicare i file modificati;
  - Correggere il problema evidenziato dal bug report nel pacchetto sorgente;
  - `dquilt refresh` per registrare le modifiche in *`bugname.patch`*;
  - `dquilt header -e` per aggiungere una descrizione;
- Se la modifica è un aggiornamento ad una patch esistente, si seguano queste istruzioni:
  - `dquilt pop` *`foo.patch`* per richiamare la patch *`foo.patch`* esistente;
  - Correggere il problema nella vecchia patch *`foo.patch`*;
  - `dquilt refresh` per aggiornare *`foo.patch`*;
  - `dquilt header -e` per aggiornarne la descrizione;
  - `while dquilt push; do dquilt refresh; done` per applicare tutte le patch mentre si sta rimuovendo il *fuzz*;
- Aggiungere una nuova revisione in cima al file di `changelog` Debian, per esempio con `dch -i`, o esplicitamente con `dch -v` *`version-revision`* e poi inserire i commenti utilizzando l'editor preferito. [1]
- Includere nella nuova voce del changelog una breve descrizione del problema e della relativa soluzione, seguita da `Closes: #654321`. In questo modo, il report del problema verrà *automaticamente* chiuso dal programma di manutenzione dell'archivio Debian nel momento stesso in cui il pacchetto viene accettato.
- Si ripetano questi passaggi per risolvere ulteriori problemi e si ricordi nel frattempo di aggiornare il file Debian di `changelog` con `dch` a seconda della necessità.
- Si ripetano i passi fatti in Sezione 6.1, Capitolo 7.

[1] Per impostare la data nel formato corretto, si usi `LANG=C date -R`.

- Quando si è soddisfatti, si può cambiare il valore del campo distribuzione nel file `changelog` da `UNRELEASED` a `unstable` (o anche `experimental`).[2]
- Caricare i pacchetti come Capitolo 9. La differenza è che questa volta, l'archivio del sorgente originale non sarà incluso, dal momento che non è stato modificato e già esiste nell'archivio Debian.

Un caso complicato può verificarsi quando si crea un pacchetto locale per sperimentare la pacchettizzazione prima di caricare la versione normale nell'archivio ufficiale, ad esempio *`1.0.1-1`*. Per facilitare gli aggiornamenti è consigliabile creare una voce con la stringa della versione come *`1.0.1-1~rc1`* nel file `changelog`. Per il pacchetto ufficiale si può riordinare il file `changelog` consolidando la modifiche locali in una singola voce. Si veda Sezione 2.6 per l'ordine delle stringhe di versione.

8.2 Controllo della nuova distribuzione

Quando si stanno preparando i pacchetti della nuova release per l'archivio Debian, bisogna controllare prima la nuova release del pacchetto originale.

Si comincia leggendo i file originali `changelog`, `NEWS`, e qualsiasi altra documentazione che possa essere stata rilasciata con la nuova versione.

Successivamente si controllano i cambiamenti tra i vecchi sorgenti originali e quelli nuovi, come mostrato in seguito, alla ricerca di qualsiasi modifica sospetta:

```
$ diff -urN foo-oldversion foo-newversion
```

I cambiamenti effettuati ad alcuni file generati automaticamente dagli Autotools come `missing`, `aclocal.m4`, `config.guess`, `config.h.in`, `config.sub`, `configure`, `depcomp`, `install-sh`, `ltmain.sh`, e `Makefile.in` possono essere ignorati. Possono anzi venire cancellati prima di eseguire **diff** per controllare i sorgenti.

8.3 Nuova distribuzione

Se un pacchetto *`foo`* è stato impacchettato correttamente nei nuovi formati `3.0 (native)` o `3.0 (quilt)`, si rende necessario impacchettare anche la versione originale muovendo la directory `debian` nella nuova sorgente. Questo può essere fatto eseguendo `tar xvzf` *`/path/to/foo_oldversion`*`.debian.tar.gz` nella nuova directory sorgente scompattata. [3] Ovviamente bisognerà effettuare alcuni passaggi:

- Creare una copia dei sorgenti originali nel file `foo_`*`newversion`*`.orig.tar.gz`.
- Aggiornare il file di `changelog` Debian con `dch -v` *`newversion-1`*.
  - Aggiungere una voce che dica `New upstream release`.
  - Descrivere brevemente i cambiamenti *nella nuova distribuzione* che correggono i bug riportati e ne chiudono altri aggiungendo `Closes:` *`#numero_bug`*.
  - Descrivere brevemente i cambiamenti *nella nuova distribuzione*, effettuati dal manutentore, che correggono i bug riportati e li chiudono aggiungendo `Closes:` *`#numero_bug`*.
- `while dquilt push; do dquilt refresh; done` per applicare tutte le patch mentre si sta rimuovendo il *fuzz*.

Se la patch/merge non è stata applicata in maniera corretta, si controlli la situazione (gli indizi vengono lasciati nei file `.rej`).

- Se una patch applicata ai sorgenti è stata integrata nei sorgenti originali,

[2] Se si utilizza il comando `dch -r` per effettuare quest'ultima modifica, ci si assicuri che l'editor salvi il file con il nome `changelog`.

[3] Se un pacchetto *`foo`* è stato pacchettizzato nel vecchio formato `1.0`, si deve invece eseguire `zcat` *`/path/to/foo_oldversion`*`.diff.gz|patch -p1` nella nuova directory sorgente scompattata.

  - si rimuova con `dquilt delete`.

- Se una patch applicata alla sorgente è andata in conflitto con i nuovi cambiamenti nei sorgenti originali,
  - si esegua `dquilt push -f` per applicare le vecchie patch forzando gli scarti nel file *`baz`*`.rej`.
  - Modificare manualmente il file *`baz`* per applicare gli effetti desiderati presenti nel file *`baz`*`.rej`.
  - `dquilt refresh` per aggiornare la patch.
- Ritornare alla procedura `while dquilt push; do dquilt refresh; done`.

Questo processo può essere automatizzato utilizzando il comando uupdate(1) come segue:

```
$ apt-get source foo
...
dpkg-source: info: extracting foo in foo-oldversion
dpkg-source: info: unpacking foo_oldversion.orig.tar.gz
dpkg-source: info: applying foo_oldversion-1.debian.tar.gz
$ ls -F
foo-oldversion/
foo_oldversion-1.debian.tar.gz
foo_oldversion-1.dsc
foo_oldversion.orig.tar.gz
$ wget http://example.org/foo/foo-newversion.tar.gz
$ cd foo-oldversion
$ uupdate -v newversion ../foo-newversion.tar.gz
$ cd ../foo-newversion
$ while dquilt push; do dquilt refresh; done
$ dch
... document changes made
```

Se si imposta il file `debian/watch` come descritto in Sezione 5.21, si può saltare il comando **wget**. Basterà eseguire uscan(1) nella directory *`foo-oldversion`* al posto del comando **uupdate**. Questo cercherà *auto-magicamente* i sorgenti corretti, li scaricherà ed eseguirà il comando **uupdate**. [4]

Si possono rilasciare questi sorgenti aggiornati ripetendo ciò che si è fatto in Sezione 6.1, Capitolo 7, ed Capitolo 9.

8.4 Aggiornare lo stile di pacchettizzazione

L'aggiornamento dello stile del pacchetto non è una attività obbligatoria nel processo di aggiornamento di quest'ultimo. Tuttavia facendo ciò si può sfruttare appieno la capacità del moderno sistema `debhelper` ed il formato sorgente `3.0`. [5]

- Se per qualsiasi motivo c'è la necessità di aggiungere file di template cancellati, si può eseguire nuovamente il comando **dh_make** nella stessa directory del sorgente del pacchetto Debian con l'opzione `--addmissing`. Fatto questo si potrà modificarlo di conseguenza.
- Se il pacchetto non è stato aggiornato per utilizzare la sintassi v7+ di **dh** del pacchetto `debhelper` per il file `debian/rules`, si deve effettuare un aggiornamento per poter usare **dh**. Si aggiorni di conseguenza anche il file `debian/control`.
- Se si desidera aggiornare il file `rules` creato con il meccanismo di inclusione "Common Debian Build System" (`cdbs`) del `Makefile` con la sintassi **dh**, si veda più avanti per capire le variabili di configurazione `DEB_*`.
  - copia locale di `/usr/share/doc/cdbs/cdbs-doc.pdf.gz`
  - The Common Debian Build System (CDBS), FOSDEM 2009 (http://meetings-archive.debian.net/pub/debian-meetings/-2009/fosdem/slides/The_Common_Debian_Build_System_CDBS/)

[4] Se il comando **uscan** scarica il sorgente aggiornato ma non esegue il comando **uupdate**, si può modificare il file `debian/watch` inserendo `debian uupdate` alla fine dell'URL.

[5] Se il proprio sponsor o altri manutentori non ritengono sia necessario aggiornare lo stile del pacchetto, allora non vale la pena perderci molto tempo. Ci sono cose più importanti da fare.

- Se si ha un pacchetto sorgente nella versione `1.0` senza il file *foo*`.diff.gz`, si può aggiornare al nuovo formato sorgente `3.0 (native)` creando il file `debian/source/format` contenente `3.0 (native)`. Gli altri file `debian/*` possono essere semplicemente copiati.
- Se si ha un pacchetto sorgente nella versione `1.0` con il file *foo*`.diff.gz`, si può aggiornare al nuovo formato sorgente `3.0 (quilt)` creando il file `debian/source/format` contenente `3.0 (quilt)`. Gli altri file `debian/*` possono essere semplicemente copiati. Se necessario, si importi il file `big.diff` generato dal comando `filterdiff -z -x '*/debian/*' foo.diff.gz > big.diff` nel sistema **quilt**.[6]
- Si nel pacchetto è stato utilizzato un altro sistema di patch come `dpatch`, `dbs`, o `cdbs` con `-p0`, `-p1`, o `-p2`, lo si può convertire con il comando `quilt` utilizzando `deb3` presente su http://bugs.debian.org/581186.
- Se il pacchetto è stato creato con il comando **dh** con l'opzione `--with quilt` o con i comandi **dh_quilt_patch** e **dh_quilt_unpatch**, si rimuovano i file prodotti e li si sostituisca usando il nuovo formato dei sorgenti `3.0 (quilt)`.

Bisogna controllare DEP - Debian Enhancement Proposals (http://dep.debian.net/) e adottare le proposte ACCEPTED.

Ci sarà anche bisogno di eseguire ulteriori operazioni descritte in Sezione 8.3.

8.5 Conversione UTF-8

Se i documenti originali sono codificato con vecchi schemi di codifica, è buona norma convertirli in UTF-8.

- Utilizzare iconv(1) per convertire le codifiche dei file di testo.

```
iconv -f latin1 -t utf8 foo_in.txt > foo_out.txt
```

- Utilizzare w3m(1) per convertire da file HTML a file di testo UTF-8. Ci si assicuri di utilizzarlo in un ambiente con localizzazione UTF-8.

```
LC_ALL=en_US.UTF-8 w3m -o display_charset=UTF-8 \
        -cols 70 -dump -no-graph -T text/html \
        < foo_in.html > foo_out.txt
```

8.6 Note per l'aggiornamento dei pacchetti

Vengono qui presentate alcune note per l'aggiornamento dei pacchetti:

- Si preservino le vecchie voci del `changelog` (sembra ovvio, ma a volte si sono verificati problemi per aver scritto `dch` mentre si sarebbe dovuto scrivere `dch -i`.)
- Cambiamenti preesistenti devono essere ricontrollati; si scarti tutto ciò che è stato incorporato in upstream (in una forma o nell'altra) e ci si ricordi di mantenere tutto quello che non è stato incorporato dall'upstream, a meno che non ci sia una buona ragione per non farlo.
- Se è stato fatto qualsiasi cambiamento nel sistema di costruzione del pacchetto (sperabilmente ce se ne renderà conto controllando i cambiamenti dall'originale) allora si aggiorni il file `debian/rules` e le dipendenze di `debian/control` se necessario.
- Si controlli nel Debian Bug Tracking System (BTS) (http://www.debian.org/Bugs/) se qualcuno ha reso disponibili delle patch ai bug che sono attualmente aperti.
- Si controlli il contenuto del file `.changes` per essere sicuri che si stia caricando per la distribuzione corretta, le risoluzioni ai bug vengono listate nel campo `Closes`, i campi `Maintainer` e `Changed-By` corrispondono, il file è firmato con GPG, ecc.

[6] È possibile dividere il file `big.diff` in tante piccole patch incrementali utilizzando il comando **splitdiff**.

Capitolo 9

Caricamento del pacchetto

Una volta testato il nuovo pacchetto approfonditamente, si potrà rilasciarlo in un archivio pubblico per condividerlo.

9.1 Caricamento nell'archivio Debian

Una volta diventati sviluppatori ufficiali, [1] si dovrà caricare il pacchetto nell'archivio Debian. [2] Si potrebbe fare manualmente, ma è più semplice usare i tool automatici che vengono forniti, come dupload(1) o dput(1). Verrà qui descritto come tutto ciò può essere fatto utilizzando **dupload**. [3]

Innanzitutto andrà impostato il file di configurazione di **dupload**. Si può modificare il file `/etc/dupload.conf` per l'intero sistema, oppure far si che il file `~/.dupload.conf` sovrascriva le poche cose che si vogliono cambiare.

Si legga la pagina del manuale dupload.conf(5) per capire cosa significhino tutte le opzioni.

L'opzione `$default_host` determina quali tra le code di caricamento vengano utilizzate in modo predefinito. `anonymous-ftp-ma` è quella primaria, ma è possibile utilizzarne anche altre. [4]

Quando si è connessi ad Internet, si può caricare il proprio pacchetto con il comando:

```
$ dupload gentoo_0.9.12-1_i386.changes
```

dupload controlla che i checksum SHA1/SHA256 dei file corrispondano con quelli del file `.changes`. Se non dovessero corrispondere si verrà avvertiti di ricostruire il pacchetto come descritto in Sezione 6.1 per poi poterlo ricaricare.

Se si incontrano problemi nel caricamento su ftp://ftp.upload.debian.org/pub/UploadQueue/, si può porre rimedio caricando automaticamente il file `*.commands` firmato con GPG via **ftp**. [5] Per esempio, si prenda il file `hello.commands`:

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1
Uploader: Foo Bar <Foo.Bar@example.org>
Commands:
 rm hello_1.0-1_i386.deb
 mv hello_1.0-1.dsx hello_1.0-1.dsc
-----BEGIN PGP SIGNATURE-----
```

[1] Si veda Sezione 1.1.

[2] Ci sono archivi accessibili al pubblico, come http://mentors.debian.net/ che lavorano quasi allo stesso modo dell'archivio Debian e forniscono una zona di caricamento per i non-DD. È possibile impostare, autonomamente, un archivio equivalente utilizzando gli strumenti elencati in http://wiki.debian.org/HowToSetupADebianRepository . Quindi questa sezione è utile per anche per i non-DD.

[3] Il pacchetto `dput` sembra avere più funzionalità e sta divenendo più popolare del pacchetto `dupload`. Questo utilizza il file `/etc/dput` per la sua configurazione globale ed il file `~/.dput.cf` per quella dei singoli utenti. Inoltre supporta nativamente anche i servizi relativi ad Ubuntu.

[4] Per maggiori informazioni, si consulti la Guida di riferimento per lo sviluppatore 5.6. "Uploading a package" (http://www.debian.org/doc/devel-manuals#devref) .

[5] Si veda ftp://ftp.upload.debian.org/pub/UploadQueue/README. Alternativamente, si può utilizzare il comando **dcut** del pacchetto `dput`.

```
Version: GnuPG v1.4.10 (GNU/Linux)

[...]
-----END PGP SIGNATURE-----
```

9.2 Includere `orig.tar.gz` per il caricamento

Quando si carica per la prima volta il pacchetto nell'archivio, si deve includere il file dei sorgenti originali `orig.tar.gz`. Se il numero di revisione Debian del pacchetto non è `1` o `0`, si deve eseguire il comando **dpkg-buildpackage** con l'opzione `-sa`.

Per il comando **dpkg-buildpackage**:

```
$ dpkg-buildpackage -sa
```

Per il comando **debuild**:

```
$ debuild -sa
```

Per il comando **pdebuild**:

```
$ pdebuild --debbuildopts -sa
```

D'altra parte, l'opzione `-sd` forzerà l'esclusione del sorgente originale `orig.tar.gz`.

9.3 Aggiornamenti scartati

Se si creano più voci nel file `debian/changelog` tralasciando gli aggiornamenti, è necessario creare il file `*_.changes` che include tutte le modifiche dall'ultimo caricamento. Questo può essere fatto specificando, al comando **dpkg-buildpackage**, l'opzione `-v` con la versione, ad esempio *`1.2`*.

Per il comando **dpkg-buildpackage**:

```
$ dpkg-buildpackage -v1.2
```

Per il comando **debuild**:

```
$ debuild -v1.2
```

Per il comando **pdebuild**:

```
$ pdebuild --debbuildopts "-v1.2"
```

Appendice A

Pacchettizzazione avanzata

Sono qui riportati alcuni suggerimenti e riferimenti sulle cose più comuni riguardanti la pacchettizzazione avanzata. Si consiglia vivamente di leggere tutti i riferimenti qui riportati.

Può essere necessario modificare manualmente il file del template del pacchetto generato dal comando **dh_make** per affrontare gli argomenti trattati in questo capitolo. Il nuovo comando **debmake** potrebbe trattare questi temi in modo migliore.

A.1 Librerie condivise

Prima di pacchettizzare una libreria condivisa, si dovrebbero leggere attentamente i seguenti riferimenti principali:

- Manuale delle policy di Debian, 8 ”Shared libraries” (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html)
- Manuale delle policy di Debian, 9.1.1 ”File System Structure” (http://www.debian.org/doc/debian-policy/ch-opersys.html#s-fhs)
- Manuale delle policy di Debian, 10.2 ”Libraries” (http://www.debian.org/doc/debian-policy/ch-files.html#s-libraries)

Di seguito alcuni semplici suggerimenti per iniziare:

- Le librerie condivise sono file oggetto ELF che contengono del codice compilato.
- Le librerie condivise sono distribuite come file `*.so`. (Né come file `*.a` né come file `*.la`)
- Le librerie condivise sono utilizzate principalmente per condividere codice tra più eseguibili, utilizzando il sistema **ld**.
- Le librerie condivise sono a volte utilizzate per fornire plugin a più di un file eseguibile con il sistema **dlopen**.
- Le librerie condivise esportano i symbols che rappresentano gli oggetti compilati, come le variabili, le funzioni e le classi; e consentono l’accesso ad essi dagli eseguibili collegati.
- Il SONAME di una libreria condivisa `lib`*`foo`*`.so.`*`1`*: `objdump -p lib`*`foo`*`.so.`*`1` `| grep SONAME` [1]
- Il SONAME di una libreria condivisa di solito corrisponde al nome del file della libreria (ma non sempre).
- Il SONAME delle librerie condivise collegate a */usr/bin/foo*: `objdump -p` */usr/bin/foo* `| grep NEEDED` [2]
- `lib`*`foo1`*: il pacchetto libreria per la libreria condivisa `lib`*`foo`*`.so.`*`1` con la versione SONAME ABI *1*.[3]

[1] In alternativa: `readelf -d lib`*`foo`*`.so.`*`1` `| grep SONAME`
[2] In alternativa: `readelf -d lib`*`foo`*`.so.`*`1` `| grep NEEDED`
[3] See Manuale delle policy di Debian, 8.1 ”Run-time shared libraries” (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-runtime) .

- Gli script dei maintainer riguardanti i pacchetti libreria devono richiamare **ldconfig** in circostanze specifiche per creare i necessari collegamenti simbolici per SONAME.[4]
- `lib`*`foo1`*`-dbg`: i pacchetti di simboli di debugging che contengono i simboli di debugging per il pacchetto della libreria condivisa `lib`*`foo1`*.
- `lib`*`foo`*`-dev`: il pacchetto di sviluppo che contiene i file header etc, per la libreria condivisa `lib`*`foo`*`.so.`*`1`*.[5]
- Un pacchetto Debian, di norma, non deve contenere file di archivi Libtool `*.la`.[6]
- Un pacchetto Debian, di norma, non deve usare RPATH.[7]
- Anche se un po' datato, ed è solo un riferimento secondario, Debian Library Packaging Guide (http://www.netfort.gr.jp/~dancer/column/libpkg-guide/libpkg-guide.html) può ancora essere utile.

A.2 Gestire `debian/`*`package`*`.symbols`

Quando si pacchettizza una libreria condivisa, si deve creare il file `debian/`*`package`*`.symbols` per gestire la versione minima associata ad ogni simbolo per le modifiche ABI compatibili con le versioni precedenti, utilizzando lo stesso SONAME della libreria per lo stesso nome del pacchetto della libreria condivisa.[8] Si dovrebbero leggere, con attenzione, i seguenti riferimenti:

- Manuale delle policy di Debian, 8.6.3 "The symbols system" (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-symbols) [9]
- dh_makeshlibs(1)
- dpkg-gensymbols(1)
- dpkg-shlibdeps(1)
- deb-symbols(5)

Di seguito un esempio di massima, per creare il pacchetto `libfoo1` alla versione originale (upstream) `1.3` con il file `debian/libfoo1.symbols` corretto:

- Preparare lo scheletro dell'albero del sorgente debianizzato utilizzando il file originale `libfoo-1.3.tar.gz`.
  - Se è il primo pacchetto per `libfoo1`, bisogna creare il file vuoto `debian/libfoo1.symbols`.
  - Se la versione originale (upstream) precedente `1.2` è stata pacchettizzata come `libfoo1` con il file `debian/libfoo1.symbols` nei propri sorgenti del pacchetto, lo si utilizzi.
  - Se la versione originale (upstream) precedente `1.2` non è stata pacchettizzata con il file `debian/libfoo1.symbols`, è necessario creare il file `symbols` per tutti i pacchetti binari disponibili con lo stesso nome del pacchetto della libreria condivisa che contiene lo stesso SONAME della libreria, ad esempio, le versioni `1.1-1` e `1.2-1`. [10]

```
$ dpkg-deb -x libfoo1_1.1-1.deb libfoo1_1.1-1
$ dpkg-deb -x libfoo1_1.2-1.deb libfoo1_1.2-1
$ : > symbols
$ dpkg-gensymbols -v1.1 -plibfoo1 -Plibfoo1_1.1-1 -Osymbols
$ dpkg-gensymbols -v1.2 -plibfoo1 -Plibfoo1_1.2-1 -Osymbols
```

[4] See Manuale delle policy di Debian, 8.1.1 "ldconfig" (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-ldconfig) .

[5] See Manuale delle policy di Debian, 8.3 "Static libraries" (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-static) e Manuale delle policy di Debian, 8.4 "Development files" (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-dev) .

[6] Si veda Debian wiki ReleaseGoals/LAFileRemoval (http://wiki.debian.org/ReleaseGoals/LAFileRemoval) .

[7] Si veda Debian wiki RpathIssue (http://wiki.debian.org/RpathIssue) .

[8] Le modifiche ABI incompatibili con le versioni precedenti, normalmente rendono necessario aggiornare il SONAME della libreria e il nome del pacchetto della libreria condivisa a quelli nuovi.

[9] Per le librerie C++ e per gli altri casi in cui il tracciamento dei singoli simboli è troppo difficile, si consulti invece Manuale delle policy di Debian, 8.6.4 "The shlibs system" (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-shlibdeps) , instead.

[10] Tutte la versioni precedenti del pacchetto Debian packages sono disponibili su http://snapshot.debian.org/ (http://snapshot.debian.org/) . La revisione Debian viene eliminata dalla versione per rendere più facile il backport del pacchetto: `1.1` << `1.1-1~bpo70+1` << `1.1-1` and `1.2` << `1.2-1~bpo70+1` << `1.2-1`

- È possibile provare a costruire l'albero dei sorgenti utilizzando dei programmi come **debuild** e **pdebuild**. (Se questo non riesce a causa della mancanza di simboli, ecc, ci sono stati dei cambiamenti ABI incompatibili con le versioni precedenti, che richiedono di cambiare il nome del pacchetto della libreria condivisa a qualcosa di simile `libfoo1a` e si dovrebbe ricominciare di nuovo da capo.)

```
$ cd libfoo-1.3
$ debuild
...
dpkg-gensymbols: warning: some new symbols appeared in the symbols file: ...
 see diff output below
--- debian/libfoo1.symbols (libfoo1_1.3-1_amd64)
+++ dpkg-gensymbolsFE5gzx        2012-11-11 02:24:53.609667389 +0900
@@ -127,6 +127,7 @@
  foo_get_name@Base 1.1
  foo_get_longname@Base 1.2
  foo_get_type@Base 1.1
+ foo_get_longtype@Base 1.3-1
  foo_get_symbol@Base 1.1
  foo_get_rank@Base 1.1
  foo_new@Base 1.1
...
```

- Se si vede il diff stampato da **dpkg-gensymbols** qui sopra, bisogna estrarre i file `symbols` aggiornati correttamente dal pacchetto binario generato dalla libreria condivisa. [11]

```
$ cd ..
$ dpkg-deb -R  libfoo1_1.3_amd64.deb libfoo1-tmp
$ sed -e 's/1\.3-1/1\.3/' libfoo1-tmp/DEBIAN/symbols \
       >libfoo-1.3/debian/libfoo1.symbols
```

- Costruire la release dei pacchetti con programmi come **debuild** e **pdebuild**.

```
$ cd libfoo-1.3
$ debuild -- clean
$ debuild
...
```

In aggiunta agli esempi sopra riportati, è necessario controllare ulteriormente la compatibilità ABI e cambiare le versioni di qualche simbolo manualmente come richiesto. [12]

Anche se è solo un riferimento secondario,, Debian wiki UsingSymbolsFiles (http://wiki.debian.org/UsingSymbolsFiles) e i suoi collegamenti possono essere utili.

A.3 Multiarch

La funzionalità multiarch introdotta in Debian wheezy integra il supporto per l'installazione dei pacchetti binari cross-architettura (in particolare `i386`<->`amd64`, ma anche altre combinazioni) in `dpkg` e `apt`. Si consiglia di leggere attentamente i seguenti riferimenti:

- Ubuntu wiki MultiarchSpec (https://wiki.ubuntu.com/MultiarchSpec) (upstream)
- Debian wiki Multiarch/Implementation (http://wiki.debian.org/Multiarch/Implementation) (Debian situation)

Esso utilizza la tripletta come `i386-linux-gnu` e `x86_64-linux-gnu` per il percorso d'installazione delle librerie condivise. La tripletta del percorso reale è impostata con il valore dinamico `$(DEB_HOST_MULTIARCH)` da dpkg-architecture(1) per ogni costruzione. Ad esempio, il percorso per installare le librerie multiarch viene modificato come segue.[13]

[11] La revisione Debian viene eliminata dalla versione per rendere più facile il backport del pacchetto: `1.3` << `1.3-1~bpo70+1` << `1.3-1`

[12] Si veda Manuale delle policy di Debian, 8.6.2 "Shared library ABI changes" (http://www.debian.org/doc/debian-policy/ch-sharedlibs.html#s-sharedlibs-updates) .

[13] Vecchi percorsi di libreria, per scopi speciali, come `/lib32/` and `/lib64/` non sono più utilizzati.

Vecchio percorso	percorso i386 multiarch	percorso amd64 multiarch
`/lib/`	`/lib/i386-linux-gnu/`	`/lib/x86_64-linux-gnu/`
`/usr/lib/`	`/usr/lib/i386-linux-gnu/`	`/usr/lib/x86_64-linux-gnu/`

Qui di seguito alcuni esempi tipici di pacchetti multiarch divisi per scenario:

- sorgente di libreria `lib`*`foo`*`-1.tar.gz`
- sorgente di programma *`bar`*`-1.tar.gz` scritto con un linguaggio compilato
- sorgente di programma *`baz`*`-1.tar.gz` scritto con un linguaggio interpretato

Pacchetto	Architettura:	Multi-Arch:	Contenuto del pacchetto
`lib`*`foo1`*	qualsiasi	uguale	la libreria condivisa, co-installabile
`lib`*`foo1`*`-dbg`	qualsiasi	uguale	i simboli di debug della libreira condivisa, co-installabile
`lib`*`foo`*`-dev`	qualsiasi	uguale	i file di header, ecc, della libreira condivisa, co-installabile
`lib`*`foo`*`-tools`	qualsiasi	straniero	il programma di supporto run-time, non co-installabile
`lib`*`foo`*`-doc`	tutti	straniero	i file di documentazione della libreria condivisa
`bar`	qualsiasi	straniero	i file del programma compilato, non co-installabile
`bar``-doc`	tutti	straniero	i file di documentazione del programma
`baz`	tutti	straniero	i file del programma interpretato

Si prega di notare che il pacchetto di sviluppo dovrebbe contenere un link simbolico per la libreria condivisa associata **senza un numero di versione**. Ad es.: `/usr/lib/x86_64-linux-gnu/libfoo.so` -> `libfoo.so.1`

A.4 Costruzione del pacchetto della libreria condivisa

Si può costruire il pacchetto Debian delle libreria, abilitando il supporto multiarch utilizzando dh(1) come di seguito:

- Aggiornare `debian/control`.
  - Aggiungere `Build-Depends: debhelper (>=10)` per la sezione del sorgente del pacchetto.
  - Aggiungere `Pre-Depends: ${misc:Pre-Depends}` per ogni pacchetto binario di una libreria condivisa.
  - Aggiungere `Multi-Arch:` per ogni sezione del pacchetto binario.
- Impostare `debian/compat` a ”10”.
- Regolare il percorso dal normale `/usr/lib/` a quello multiarch `/usr/lib/$(DEB_HOST_MULTIARCH)/` per tutti gli script di pacchettizzazione.
  - Invocare `DEB_HOST_MULTIARCH ?= $(shell dpkg-architecture -qDEB_HOST_MULTIARCH)` nel file `debian/rules` per impostare la variabile `DEB_HOST_MULTIARCH`.
  - Sostituire `/usr/lib/` con `/usr/lib/$(DEB_HOST_MULTIARCH)/` nel file `debian/rules`.
  - Se `./configure` viene utilizzato nella parte target di `override_dh_auto_configure` in `debian/rules`, ci si assicuri di sostituirlo con `dh_auto_configure -- .` [14]
  - Sostituire tutte le occorrenze di `/usr/lib/` con `/usr/lib/*/` nei file `debian/`*`foo`*`.install`.
  - Genera dinamicamente dei file come `debian/`*`foo`*`.links` da `debian/`*`foo`*`.links.in` aggiungendo un script al target di `override_dh_auto_configure` in `debian/rules`.

[14] In alternativa, si possono aggiungere gli argomenti `--libdir=\$${prefix}/lib/$(DEB_HOST_MULTIARCH)` e `--libexecdir=\$${prefix}/lib/$(DEB_HOST_MULTIARCH)` a `./configure`. Si noti che `--libexecdir` indica il percorso predefinito d’installazione dei programmi eseguibili gestiti da altri programmi e non dagli utenti. Il percorso predefinito di Autotools è `/usr/libexec/` mentre quello di Debian è `/usr/lib/`.

```
override_dh_auto_configure:
        dh_auto_configure
        sed 's/@DEB_HOST_MULTIARCH@/$(DEB_HOST_MULTIARCH)/g' \
                debian/foo.links.in > debian/foo.links
```

Ci si assicuri di verificare che il pacchetto di libreria condivisa contenga solo i file attesi, e che il pacchetto -dev continui a funzionare.

Tutti i file installati contemporaneamente come pacchetto multiarch con lo stesso percorso del file devono avere esattamente lo stesso contenuto. È necessario prestare attenzione alle differenze generate dall'ordine dei byte nei dati e dall'algoritmo di compressione.

A.5 Pacchetto nativo Debian

Se il pacchetto è mantenuto solo per Debian o per uso locale, il suo sorgente potrebbe contenere tutti i file in `debian/*`. In questo caso, ci sono 2 modi per pacchettizzarlo.

È possibile creare l'archivio originale escludendo i file in `debian/*` e pacchettizandolo come pacchetto Debian non nativo, come descritto in Sezione 2.1. Questo è il metodo normale che alcune persone incoraggiano ad utilizzare.

L'alternativa è utilizzare lo stesso metodo usato dai pacchetti Debian nativi.

- Creare un pacchetto sorgente nativo di Debian nel formato `3.0 (native)` usando un singolo file tar compresso che include tutti i file.
  - *pacchetto_versione*`.tar.gz`
  - *pacchetto_versione*`.dsc`
- Costruire pacchetti binari Debian dal pacchetto sorgente nativo di Debian.
  - *pacchetto_versione_arch*`.deb`

Per esempio, se si hanno i file sorgenti in `~/mypackage-1.0` senza i file `debian/*`, si può creare un pacchetto nativo Debian, utilizzando il comando **dh_make** come segue:

```
$ cd ~/mypackage-1.0
$ dh_make --native
```

La directory `debian` e il suo contenuto, sono creati proprio come Sezione 2.8. Questo non crea un archivio poiché si tratta di un pacchetto Debian nativo. Questa è l'unica differenza. Il resto delle attività di pacchettizzazione sono praticamente le stesse.

Dopo l'esecuzione del comando **dpkg-buildpackage**, si possono vedere i seguenti file nella directory principale:

- `mypackage_1.0.tar.gz`

  Questo è l'archivio del codice sorgente creato dalla directory `mypackage-1.0` dal comando **dpkg-source**. (Il suffisso non è `orig.tar.gz`.)
- `mypackage_1.0.dsc`

  Questo è un sommario del contenuto del codice sorgente, come per i pacchetti non-nativi Debian. (non c'è revisione Debian.)
- `mypackage_1.0_i386.deb`

  Questo è il pacchetto binario completo, come per i pacchetti non-nativi Debian. (non c'è revisione Debian.)
- `mypackage_1.0_i386.changes`

  Questo file descrive tutte le modifiche apportate nella versione attuale del pacchetto, come per i pacchetti Debian non-nativi. (non c'è revisione Debian.)