Sviluppo software in gruppi di lavoro complessi[1]

Mattia Monga

Dip. di Informatica
Università degli Studi di Milano, Italia
mattia.monga@unimi.it

Anno accademico 2018/19, I semestre

Lezione IX: Sistemi di *build automation*

Dove siamo?

Come ci si organizza? *"The tar pit"* Sviluppare *software* necessita sforzi collettivi coordinati: gruppi di lavoro complessi con obiettivi in rapida evoluzione e innumerevoli *concern* intrecciati rendono molto difficile la divisione del lavoro

Come si gestiscono i manufatti? La produzione del *software* consiste principalmente nella modifica di *file*: i sistemi di *configuration management* permettono di tenere sotto controllo l'evoluzione delle revisioni

Build

Costruire (assemblare) un prodotto software fatto di molti componenti è tutt'altro che banale:

- dipendenze da componenti che non controlliamo (**dependency hell**)
- dipendenze fra componenti che stiamo sviluppando

Dependency hell (cont.)

Versionamento semantico
`http://semver.org/spec/v2.0.0.html`

- Numero di versione con tre token MAJOR.MINOR.PATCH
- MAJOR cambia quando ci sono cambiamenti incompatibili nelle API
- MINOR cambia con nuove funzionalità (ma *backwards-compatible*)
- PATCH solo bugfix

Make

Stuart Feldman, 1977 at Bell Labs.
Permette di specificare **dipendenze** fra processi di generazione.
Dipendenze: se cambia (secondo la data dell'ultima modifica) un prerequisito, allora il processo di generazione deve essere ripetuto.

```
helloworld.o: helloworld.c
        cc -c -o helloworld helloworld.c

helloworld: helloworld.o
        cc -o $@ $<

.PHONY: clean
clean:
        rm helloworld.o helloworld
```


Make: come funziona

- Le dipendenze definiscono un grafo aciclico che ammette un unico *ordinamento topologico* (in quanto si passa una sola volta da ogni *target*)
- I processi di generazione (*receipt*) sono eseguiti seguendo l'ordinamento topologico
- Nei `make` moderni è possibile eseguire processi di generazione indipendenti in parallelo (`make -j`)

Parte del materiale che segue è preso da: `http://www.lrde.epita.fr/~adl/autotools.html`

Standard Makefile Targets

`make all`	Build programs, libraries, documentation, etc. (Same as `make`.)
`make install`	Install what needs to be installed.
`make install-strip`	Same as `make install`, then strip debugging symbols.
`make uninstall`	The opposite of `make install`.
`make clean`	Erase what has been built (the opposite of `make all`).
`make distclean`	Additionally erase anything `./configure` created.
`make check`	Run the test suite, if any.
`make installcheck`	Check the installed programs or libraries, if supported.
`make dist`	Create `PACKAGE-VERSION.tar.gz`.

Standard File System Hierarchy

Directory variable	Default value
`prefix`	`/usr/local`
`exec-prefix`	`prefix`
`bindir`	`exec-prefix/bin`
`libdir`	`exec-prefix/lib`
...	
`includedir`	`prefix/include`
`datarootdir`	`prefix/share`
`datadir`	`datarootdir`
`mandir`	`datarootdir/man`
`infodir`	`datarootdir/info`
...	

Portabilità della costruzione

Il modello di `make` assume un ambiente di *build* fisso.

- L'ipotesi è irrealistica perfino nel mondo dello sviluppo anni '70 (C/UNIX)
- Compilatori, librerie cambiano molto anche nell'ambito degli standard

Non-Portability in C

Funzioni di libreria che:

- non esistono ovunque (es. `strtod()`)
- hanno nomi diversi (es. `strchr()` vs. `index()`)
- hanno prototipi differenti (es. `int setpgrp(void);` vs. `int setpgrp(int, int);`)
- hanno comportamenti diversi (e.g., `malloc(0);`)
- richiedono diverse dipendenze transitive (`pow() in libm.so or in libc.so?`)
- richiedono diverso trattamento (`string.h` vs. `strings.h` vs. `memory.h`)

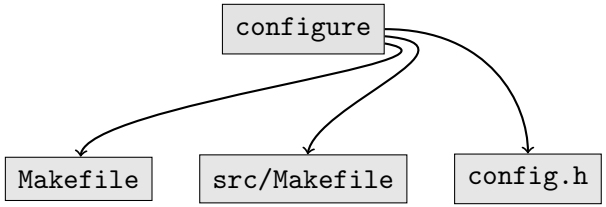
Le soluzioni in C

- `#if/#else`
- substitution macros
- substitution functions

Code Cluttered with #if/#else

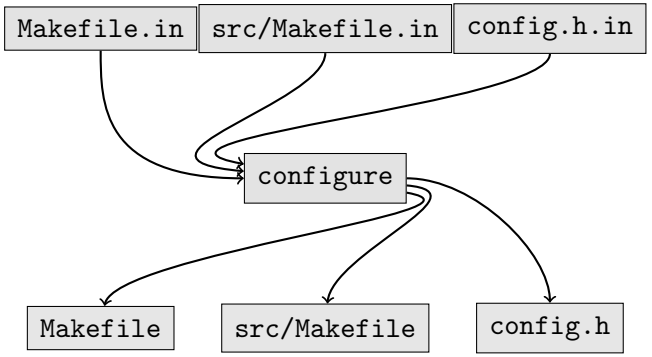
```
#if !defined(CODE_EXECUTABLE)
  static long pagesize = 0;
#if defined(EXECUTABLE_VIA_MMAP_DEVZERO)
  static int zero_fd;
#endif
  if (!pagesize) {
#if defined(HAVE_MACH_VM)
      pagesize = vm_page_size;
#else
      pagesize = getpagesize();
#endif
#if defined(EXECUTABLE_VIA_MMAP_DEVZERO)
      zero_fd = open("/dev/zero", O_RDONLY,0644);
      if (zero_fd < 0) {
        fprintf(stderr, "trampoline: Cannot open /dev/zero!\n");
        abort();
      }
#endif
  }
#endif
```


configure



- `configure` verifica le caratteristiche dell'ambiente di costruzione.
- genera un `config.h` con le `#define` giuste
- e un `Makefile`

configure process



`*.in` sono configuration templates da cui `configure` genera lo script di verifica dell'ambiente.

Ant

- Build automation: dipendenze + processi di generazione + riconfigurazione all'ambiente di *build*
- Java e XML
- Plugin in Java

Ant

```
<?xml version="1.0"?>
<project name="Hello" default="compile">
    <target name="clean" description="remove intermediate files">
        <delete dir="classes"/>
    </target>
    <target name="clobber" depends="clean" description="remove all artifact files">
        <delete file="hello.jar"/>
    </target>
    <target name="compile" description="compile the Java source code to class files">
        <mkdir dir="classes"/>
        <javac srcdir="." destdir="classes"/>
    </target>
    <target name="jar" depends="compile" description="create a Jar file for the application">
        <jar destfile="hello.jar">
            <fileset dir="classes" includes="**/*.class"/>
            <manifest>
                <attribute name="Main-Class" value="HelloProgram"/>
            </manifest>
        </jar>
    </target>
</project>
```


Gradle

- *Domain specific language* basato su Groovy
- *build-by-convention* (eventualmente configurabile)
- Supporta cataloghi di componenti
- Supporto per il test
- Reportistica

Make

```
plugins {
        id "java"
        id "eclipse"
        id "jacoco"
}

repositories {
    jcenter()
}

dependencies {
  testCompile "junit:junit:4.11"
        testCompile 'org.assertj:assertj-core:3.5.2'
  compile 'commons-io:commons-io:2.5'
}

jacoco {
        jacocoTestReport.reports.xml.enabled = true
}
```


Continuous Integration

Configuration Management

Esplicitazione delle dipendenze

Test d'unità, d'integrazione, d'accettazione

Build automatizzate

Deployment simulato o automatizzato (domani)

$\rightsquigarrow$ *Continuous Integration & Delivery*

Continuous Integration

- Martin Fowler 2001-6 (`http://www.martinfowler.com/articles/continuousIntegration.html`)
- Tradizionalmente, l'integrazione è una delle parti più lunghe e rischiose dei progetti *software*
- CI $\rightsquigarrow$ integrazione continua (incrementale) per minimizzare il rischio di fallimento
  - *Reduced Deployment Risk*
  - *Believable Progress*
  - *User Feedback*

Continuous Integration

1. Lavoro su una copia locale sulla *macchina di sviluppo*
2. *Build* vs. *Compile*: oltre alla costruzione esecuzione di test
3. *build* funzionante sulla *macchina di sviluppo*
4. Caricamento sulla *macchina d'integrazione*
5. *build* funzionante sulla *macchina d'integrazione*

Almeno una volta al giorno.

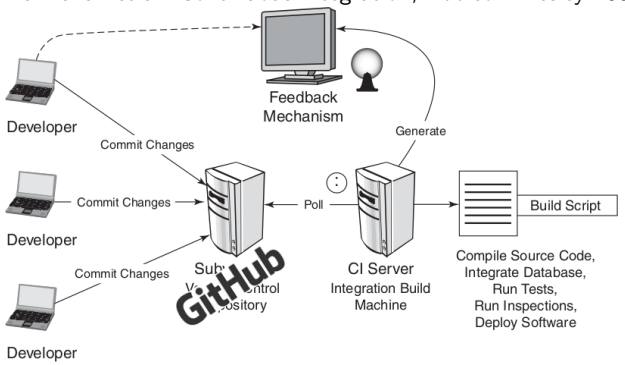
Continuous Integration

Cambiato drasticamente negli ultimi 10 anni. Fowler:

- "The current open source repository of choice is Subversion. (The older open-source tool CVS is still widely used, and is much better than nothing, but Subversion is the modern choice.)"
- "One of the features of version control systems is that they allow you to create multiple branches, to handle different streams of development. This is a useful, nay essential, feature — but it's frequently overused and gets people into trouble. Keep your use of branches to a minimum." (ma... suggerisce tecniche di *pending head* simili a quelle usate da `gitflow`)

Continuous Integration

Da Duvall et al. Continuous Integration, Addison Wesley 2007



Continuous Integration

Il *build* non dovrebbe impiegare più di 10 minuti, altrimenti si perde il *feedback* immediato. Che fare se è necessario più tempo?

- *Deployment pipeline:* il *build* è spezzato in più fasi, *commit build*, *slower tests build*, ecc.
- Il *commit build* impiega meno di 10 min, il resto parte poi

Continuous Delivery

- Il sistema dove avviene l'**integrazione** dovrebbe essere il più possibile "simile" a quello di **produzione**
- Il *deployment* verso l'ambiente di produzione può essere automatizzato (garantendo così un maggior controllo sulla effettiva configurazione in uso!)

Sottomoduli Git

`https://git-scm.com/book/en/v2/Git-Tools-Submodules`

Permettono di tenere sotto controllo una 'dipendenza'

- una *sottodirectory* contiene un altro *repository*
- `git submodule add repo dirname`
- il *file* `.gitmodules` viene versionato
- `git clone --recursive`
- (`git submodule init`) `git submodule update` per ottenere la versione attuale del modulo
- ma nel progetto **congelo** una specifica versione `git add submoduledirname`

Cancellare un sottomodulo

Non c'è un modo diretto, occorrono più operazioni:

1. Cancellare a mano in `.gitmodules`
2. Cancellare a mano in `.git/config`
3. `git rm --cached submoduledirname`
4. *commit* (e cancellare il sottomodulo)