L'assemblage de logiciels et l'outil `make`

Guy Tremblay
Professeur

Département d'informatique
UQAM
http://www.labunix.uqam.ca/~tremblay_gu

INF600A
2 octobre 2018

Contenu

1 Introduction

2 Qu’est-ce que l’assemblage de logiciels ?

3 L’outil `make` sur Unix

4 Conclusion

1. Introduction

La première chose à faire pour développer du code de façon professionnelle. . .

«*You need to get the development infrastructure environment in order. That means adopting (or improving) the fundamental Starter Kit practices :*

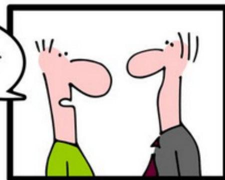
- *Version control*
- *Unit Testing*
- *Build automation*

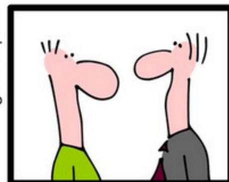
Version control needs to come before anything else. It's the first bit of infrastructure we set up on any project.»

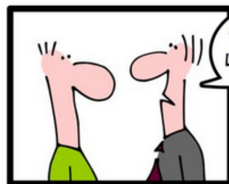
«*Practices of an Agile Developer—Working in the Real World*», Subramaniam & Hunt, 2006.

L’étape de `build` est cruciale

Source : `geek-and-poke.com`







HOW TO BECOME INVALUABLE

2. Qu’est-ce que l’assemblage de logiciels ?

Qu’est-ce que l’assemblage d’un logiciel ?

Historically, build has often referred either to the process of converting source code files into standalone software artifact(s) that can be run on a computer, or the result of doing so.

Qu’est-ce que l’assemblage d’un logiciel ?

Historically, build has often referred either to the process of converting source code files into standalone software artifact(s) that can be run on a computer, or the result of doing so.

The process of building a computer program is usually managed by a build tool, a program that coordinates and controls other programs. [. . .] The build utility needs to compile and link the various files, in the correct order. If the source code in a particular file has not changed then it may not need to be recompiled [. . .].

Source: `https://en.wikipedia.org/wiki/Software_build`

Il faut distinguer entre *composants sources* et *composants dérivés*

Composants sources (*source components*)

Les composants qui sont — et doivent être — créés manuellement par les développeurs.

Exemple pour programmes C : Fichiers `*.[hc]`, `Makefile`.

Il faut distinguer entre *composants sources* et *composants dérivés*

Composants sources (*source components*)

Les composants qui sont — et doivent être — créés manuellement par les développeurs.

Exemple pour programmes C : Fichiers `*.[hc]`, `Makefile`.

Composants dérivés (*derived components*)

Les composants qui peuvent être été créés automatiquement par la machine, sans l'intervention explicite des développeurs.

Exemple pour programmes C : Fichiers `*.o`, exécutables.

Source: «*Essential Open Source Toolset*», Zeller & Krinke

Il faut distinguer entre *composants sources* et *composants dérivés*

Rappel important en lien avec le contrôle du code source :

- Seuls les fichiers pour les composants sources doivent être mis dans le système de contrôle du code source.

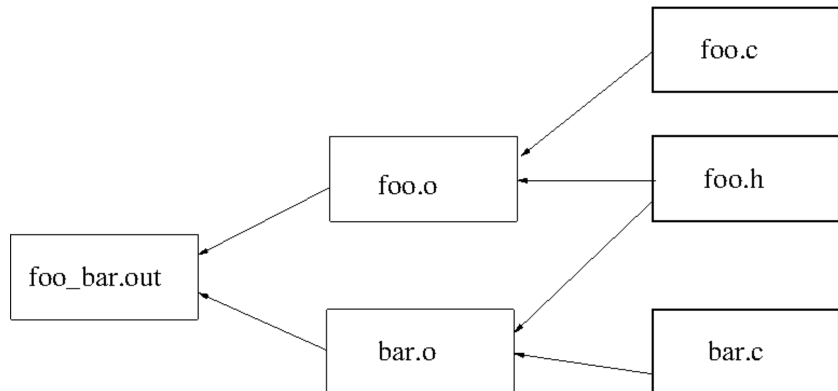
  $\Rightarrow$ Les fichiers pour les composants dérivés ne doivent pas être mis dans le système de contrôle du code source.

Le rôle d’un outil d’assemblage de logiciels

Le rôle d’un outil d’assemblage de logiciels est d’assurer qu’un logiciel soit assemblé. . .

- à partir des bons composants sources
- toujours de la bonne façon
- sans intervention humaine
- rapidement, en regénérant le nombre minimal de composants dérivés
  $\Rightarrow$ on ne recompile que le strict nécessaire

Tous les outils d'assemblage reposent sur un modèle du logiciel à construire = Graphe des dépendances



Le processus d'assemblage repose sur l'analyse du graphe de dépendances

Soit un composant dérivé A qui dépend de $A_1, A_2, \ldots, A_n$.

Alors, le composant A doit être regénéré si une des conditions suivantes s'applique :

1. A n'existe pas
2. un des composants A_i a été modifié
3. un des composants A_i doit être regénéré

Donc : il s'agit donc d'un processus récursif !

Source: «*Essential Open Source Toolset*», Zeller & Krinke

3. L’outil `make` sur Unix

Qu'est-ce que `make` ?

In software development, Make is a utility that automatically builds executable programs and libraries from source code by reading files called makefiles which specify how to derive the target program. [. . .]

Qu'est-ce que `make` ?

In software development, Make is a utility that automatically builds executable programs and libraries from source code by reading files called makefiles which specify how to derive the target program. [. . .]

Besides building programs, Make can be used to manage any project where some files must be updated automatically from others whenever the others change.

Source: *https://en.wikipedia.org/wiki/Make_(software)*

Un exemple simple : Compilation et exécution d'un fichier `hello.c`


```
$ cat hello.c
#include <stdio.h>

int main()
{
  printf( "Hello, World!\n" );
  return 0;
}
```

Un exemple simple : Compilation et exécution d’un fichier `hello.c`


```
$ cat makefile
default: run

run: hello
	./hello

hello: hello.c
	gcc -o hello hello.c

clean:
	rm -f hello hello.o
```

Un exemple montrant que l'outil `make` ne s'applique pas uniquement à la compilation de «programmes»

Dans mon éditeur de texte (`emacs`), la clé «F12» est associée à la macro `save-and-make` :

- Sauve le contenu du *buffer* dans le fichier ;
- Lance l'exécution de `make` dans le répertoire courant.

Deux principaux cas d'utilisation :

a. Traitement de texte avec LaTeX :

b. Écriture de code :

Un exemple montrant que l'outil `make` ne s'applique pas uniquement à la compilation de «programmes»

Dans mon éditeur de texte (`emacs`), la clé «F12» est associée à la macro `save-and-make` :

- Sauve le contenu du *buffer* dans le fichier ;
- Lance l'exécution de `make` dans le répertoire courant.

Deux principaux cas d'utilisation :

a. Traitement de texte avec LaTeX :
   `make` regénère le fichier PDF, mettant à jour le PDF affiché à l'écran.

b. Écriture de code :

Un exemple montrant que l'outil `make` ne s'applique pas uniquement à la compilation de «programmes»

Dans mon éditeur de texte (`emacs`), la clé «F12» est associée à la macro `save-and-make` :

- Sauve le contenu du *buffer* dans le fichier ;
- Lance l'exécution de `make` dans le répertoire courant.

Deux principaux cas d'utilisation :

a. Traitement de texte avec LaTeX :

b. Écriture de code :
`make` lance l'exécution des tests (unitaires et/ou acceptation).

Un autre exemple montrant que l'outil make ne s'applique pas uniquement à la compilation


```
$ cat makefile
MAIN=build

default: $(MAIN)

$(MAIN) $(MAIN).pdf:: $(MAIN).tex macros.tex biblio/*tex
        pdflatex $(MAIN)

etudiants: clean
        make $(MAIN).pdf
        scp $(MAIN).pdf tremblay_gu@java:public_html/INF600A/M

prof:
        $(HOME)/cmd/sans-notes.sh
        make $(MAIN).pdf
        scp $(MAIN).pdf ...
```

Ce que fait `make` : Il analyse le graphe des dépendances entre tâches décrit par un `Makefile`

Un fichier `Makefile` contient la description d'une ou plusieurs tâches (règles), où pour chaque tâche on a ...

- le nom de la tâche = la cible
- les préalables à la tâche = dépendances entre tâches
- les commandes (*shell*) à exécuter pour la tâche

Forme générale d'une règle

$target_1 \quad target_2 \quad \ldots \quad target_n : \quad source_1 \quad \ldots \quad source_m$

$\quad command_1$

$\quad command_1$

$\quad \ldots$

$\quad command_k$

Ce que fait `make` : Il analyse le graphe des dépendances entre tâches décrit par un `Makefile`

- il analyse le graphe de dépendances des tâches
- il détermine les tâches qui sont absolument nécessaires pour construire la cible désirée
- il exécute les commandes associées aux tâches identifiées

Ce que fait `make` : Il analyse le graphe de dépendances entre tâches décrit par un `Makefile`

Processus utilisé :

- Fouille en profondeur (*depth-first search*) du graphe de dépendances ⇒ algorithme récursif

- Utilise la date de dernière modification d'un fichier pour déterminer si le fichier a été modifié :
  - si A dépend de B
    et la date de dernière modification de B est après celle de A
    alors B a été modifié

Contenu d’un Makefile

- Des commentaires :

```
# Un commentaire debute par un diese.
```

- Des déclarations de variables :

```
MAIN = build
```

- Des cibles et leurs dépendances (possiblement avec variables) :

```
$(MAIN).pdf: $(MAIN).tex macros.tex biblio/*tex
```

- Les commandes à exécuter pour une cible :

```
pdflatex $(MAIN)
```

Contenu d’un `Makefile` et caractère de tabulation ★23

Remarque importante : Les lignes indiquant les commandes à exécuter doivent débuter par un caractère de tabulation !

> *«The fact that commands have to be indented by tabs (and not, for example, blank spaces !) is the biggest stumbling block for beginners wanting to create a Makefile. Stuart Feldman [(le concepteur de* `make`*)] himself apparently encountered the same problem a few days after finalizing the original version of* MAKE *;* MAKE *had already been too widely distributed by then for him to be able to correct this design error.»*

Source*: «*Essential Open Source Toolset*», Zeller & Krinke*

Contenu d’un `Makefile`

Définitions de variables

```
MAIN=build

default: $(MAIN)

$(MAIN) $(MAIN).pdf: $(MAIN).tex macros.tex biblio/*tex
	pdflatex $(MAIN)

etudiants: clean
	make $(MAIN).pdf
	scp $(MAIN).pdf tremblay_gu@java:public_html/INF600A/M

prof:
	$(HOME)/cmd/sans-notes.sh
	make $(MAIN).pdf
	scp $(MAIN).pdf ...
```

Contenu d'un Makefile

Définitions de cibles


```
MAIN=build

default: $(MAIN)

$(MAIN) $(MAIN).pdf: $(MAIN).tex macros.tex biblio/*tex
        pdflatex $(MAIN)

etudiants: clean
        make $(MAIN).pdf
        scp $(MAIN).pdf tremblay_gu@java:public_html/INF600A/M

prof:
        $(HOME)/cmd/sans-notes.sh
        make $(MAIN).pdf
        scp $(MAIN).pdf ...
```

Contenu d'un Makefile

Définitions de dépendances


```
MAIN=build

default: $(MAIN)

$(MAIN) $(MAIN).pdf: $(MAIN).tex macros.tex biblio/*tex
        pdflatex $(MAIN)

etudiants: clean
        make $(MAIN).pdf
        scp $(MAIN).pdf tremblay_gu@java:public_html/INF600A/M

prof:
        $(HOME)/cmd/sans-notes.sh
        make $(MAIN).pdf
        scp $(MAIN).pdf ...
```

Contenu d'un Makefile

Définitions d'actions à exécuter

```
MAIN=build

default: $(MAIN)

$(MAIN) $(MAIN).pdf: $(MAIN).tex macros.tex biblio/*tex
        pdflatex $(MAIN)

etudiants: clean
        make $(MAIN).pdf
        scp $(MAIN).pdf tremblay_gu@java:public_html/INF600A/M

prof:
        $(HOME)/cmd/sans-notes.sh
        make $(MAIN).pdf
        scp $(MAIN).pdf ...
```

Lancement de la commande `make`

- On identifie la cible principale :
  - Commande «`make`» :
    - Utilise la première cible définie dans le fichier.
  - Commande «`make foo`» :
    - utilise la cible «`foo`».
  - Commande «`make foo` VAR=bar» :
    - utilise la cible «`foo`» et définit la variable `VAR` comme étant `bar`.

- Récursivement, toutes les cibles dont les dépendances ne sont pas satisfaites sont exécutées

- Ne réexécute que ce qui est nécessaire

Attention : «`make foo VAR=bar`» $\neq$ «`VAR=bar make foo`» !

Lancement de la commande `make`

Note : «`make -n`» montre ce qui serait exécuté, mais sans l'exécuter — utile pour comprendre ce que fait `make` !

```
$ make -n
pdflatex build

$ make -n build.pdf
pdflatex build

$ make -n MAIN=tests-unitaires
pdflatex tests-unitaires
```

Contenu d'un Makefile (suite)

- Des règles de dépendances implicites :

```
# Regle pour compilation des fichiers .c
CC = gcc
CFLAGS = -ggdb -Wall -O

# Regle implicite
.c.o:
	$(CC) -c $(CFLAGS) $<
```

Variables spéciales dans les règles implicites :

- `$<` : Le premier préalable
- `$@` : La cible

Contenu d'un Makefile (suite)

- Une autre forme, plus générale, pour des règles de dépendances implicites (GNU make) :

```
# Regle pour compilation des fichiers .c
CC = gcc
CFLAGS = -ggdb -Wall -O

# Regle implicite avec dependances additionnelles.
%.o: %.c definitions.h
	$(CC) -c $(CFLAGS) -o $@ $<
```

Patron spécial de règles implicites :

- `%` : patron pour une chaine de caractères arbitraire

Écriture d’un Makefile (1)
Manuellement

À la main = on détermine soi-même les dépendances, et on écrit, soi-même, chaque série de commandes

Écriture d'un Makefile (1)
Manuellement

À la main = on détermine soi-même les dépendances, et on écrit, soi-même, chaque série de commandes

Désavantages :

- Cibles omises
  $\Rightarrow$ Éléments manquants dans le logiciel déployé
- Dépendances omises
  $\Rightarrow$ Recompilations inutiles

Écriture d'un Makefile (2)

À l'aide du compilateur

Un logiciel composé de nombreux fichiers :

```
$ ls *[hc]
biblio.c            MiniCUnit.c
bool.h              MiniCUnit.h
communications.c    outils.c
communications.h    outils.h
configuration.h     sequences.c
emprunts.c          sequences.h
emprunts.h          testCommunications.c
gererArguments.c    testEmprunts.c
gererArguments.h    testGererArguments.c
gererBD.c           testGererBD.c
gererBD.h           testGererErreurs.c
gererErreurs.c      testMiniCUnit.c
gererErreurs.h      testOutils.c
```

Écriture d'un Makefile (2)

À l'aide du compilateur

On utilise le compilateur pour générer les dépendances :

```
$ gcc -MM *.[hc]
```

```
biblio.o: biblio.c gererArguments.h gererBD.h emprunts.h\
          bool.h gererErreurs.h communications.h\
          configuration.h outils.h
bool.o: bool.h
communications.o: communications.c outils.h bool.h communicati
...
sequences.o: sequences.c bool.h sequences.h
...
testGererErreurs.o: testGererErreurs.c configuration.h\
                    gererErreurs.h bool.h MiniCUnit.h
testMiniCUnit.o: testMiniCUnit.c MiniCUnit.h bool.h
testOutils.o: testOutils.c outils.h bool.h MiniCUnit.h
```

Écriture d’un Makefile (3)

À l’aide d’un outil spécialisé

À l’aide d’outils tels que (GNU) `autoconf` et `automake`, qui génèrent un fichier `Makefile.in` (cibles, dépendances et règles) à partir d’un fichier de configuration minimale

Pour une introduction, voir notamment
`http://mij.oltrelinux.com/devel/autoconf-automake`

Quelques cibles standards

- `all` : Compile le programme en entier
- `test` : Lance l’exécution des tests
- `install` : Compile le programme et l’installe
- `uninstall` : Désinstalle le programme
- `clean` : Supprime les fichiers temporaires créés par `make`

4. Conclusion

Conclusion

De nombreux autres outils d'assemblage sont disponibles

Généralement associés à un langage de programmation spécifique :

- Ant : Java
- Maven : Java
- Rake : Ruby

Conclusion

De nombreux autres outils d'assemblage sont disponibles

Généralement associés à un langage de programmation spécifique :

- Ant : Java
- Maven : Java
- Rake : Ruby

Outils utiles même s'il n'y a pas de «compilation» !

Ces outils sont utiles... même s'il n'y a pas de processus de compilation et d'édition des liens, e.g., exécution de scripts, Rake avec Ruby !

Conclusion

De nombreux autres outils d'assemblage sont disponibles

Généralement associés à un langage de programmation spécifique :

- Ant : Java
- Maven : Java
- Rake : Ruby

Outils utiles même s'il n'y a pas de «compilation» !

Ces outils sont utiles... même s'il n'y a pas de processus de compilation et d'édition des liens, e.g., exécution de scripts, Rake avec Ruby !

Ce sont des outils «d'automatisation des tâches»... comme les scripts shell !

Références

A. Hunt and D. Thomas.
The Pragmatic Programmer—From Journeyman to Master.
Addison-Wesley, 2000.

V. Subramaniam and A. Hunt.
Practices of an Agile Developer—Working in the Real World.
The Pragmatic Bookshelf, 2006.

A. Zeller and J. Krinke.
Essential Open Source Toolset.
John Wiley & Sons, Ltd, Chichester, UK, 2005.