Laboratorio di Metodi di Programmazione
Sviluppo di programmi Java

Luca Padovani

lpadovan@cs.unibo.it

http://www.cs.unibo.it/~lpadovan/didattica/LaboratorioProgrammazione2004/

Sommario

1. materiale di riferimento
2. l'editor Emacs
3. l'editor Vim
4. compilatori per Java
5. `make`

Materiale di riferimento

- Software GNU `http://www.gnu.org/`
- Java `http://java.sun.com/`
- Libreria standard Java `http://java.sun.com/j2se/1.5.0/docs/api/`
- GCJ `http://gcc.gnu.org/java/`
- Manuale GCJ `http://gcc.gnu.org/onlinedocs/gcj/`
- Manuale Make `http://www.gnu.org/software/make/manual/make.html` (`info make`)
- Emacs `http://www.gnu.org/software/emacs/emacs.html` (`info emacs`)
- Vim `http://www.vim.org/`

Emacs

- autore: Richard Stallman
- licenza: GPL
- architettura: C + Lisp
- editor sostanzialmente non modale
- esteso supporto per linguaggi di programmazione (syntax highlighting, shortcut, completamento stringhe, . . .)
- estendibile con Lisp
- embedding di altri processi (shell, compilatori, debugger, . . .)

Emacs: primi passi

Invocazione

- `$ emacs`
  GUI se eseguito in ambiente grafico (ad es. terminale X), altrimenti interfaccia testuale
- `$ emacs -nw`
  per forzare l'interfaccia testuale
- `$ emacs <nome-file>`
  per iniziare a lavorare su un file (esistente o meno)

Terminazione

- `C-x C-c`
  esce senza salvare, chiede conferma se esistono buffer modificati

Emacs: help

- `C-h t`
  lancia il tutorial
- `M-x info`
  esegue `info` dentro Emacs

Il tasto delle emergenze:

- `C-g`

Emacs: file di configurazione

`~/.emacs`

(è un file nascosto, usate `ls -la` per vederlo)

La sintassi è Lisp-like, per informazioni `info elisp`

Emacs: movimento del "punto"

Di un carattere/riga:

- con i tasti cursore
- `C-p C-n C-b C-f` (su giù sx dx)

Di una "parola":

- `M-` e tasti cursore sx/dx
- `M-b M-f` (sx dx)

Di una "pagina":

- tasti "page up" e "page down"
- `C-v M-v` (giù su)

Di un "paragrafo":

- `C-` e tasti cursore su/giù

Altri movimenti:

- `C-A` o `<HOME>`, inizio riga
- `C-E` o `<END>`, fine riga
- `C-<HOME>` inizio buffer
- `C-<END>` fine buffer

Iterazione comando:

- `M-`n `<comando>`

Emacs: editing di base

- inserimento e cancellazione come nei normali word processor
- copia & incolla, terminologia:
  - kill (taglia, cut)
  - yank (incolla, paste)
- copia & incolla, comandi:
  - `C-k` kill line (dal punto fino alla fine della riga)
  - `C-y` oppure `S-<INS>` yank
  - copia = taglia + incolla, `M-x copy-region-as-kill`, oppure `C-<INS>`
- selezione (region):
  - `C-<SPC>` inizia selezione (`transient-mark-mode` per visualizzarla)
  - `C-w` oppure `S-<DEL>` kill selezione
- `C-x u` oppure `C-_` undo

Emacs: gestione file

- `C-x C-f`
  apri file (è possibile usare la history)
- `C-x C-s`
  salva file
- `C-x C-w`
  salva con nome

Emacs: ricerca e sostituzione

Ricerca

- `C-s <testo-da-cercare>` ricerca incrementale in avanti
- `C-r <testo-da-cercare>` ricerca incrementale all'indietro
- `C-M-s` ricerca espressione regolare, per esempio

  `{[^}]*}`
- case sensitivity automatica

Sostituzione

- `M-x replace-string <RET> <testo-da-cercare> <RET>` `<testo-da-sostituire>` sostituzione di testo
- `M-x replace-regexp`, oppure `C-M-%` sostituzione di espressioni regolari

  `{\([^}]*\)}` `(\1)`

Emacs: supporto alla programmazione

- riconoscimento automatico del tipo di file sorgente (dall'estensione o da opportuni commenti)
- syntax highlighting
  - `M-x global-font-lock-mode` per attivarla e disattivarla
  - aggiungere `(global-font-lock-mode 1)` nel `.emacs`
- indentazione (semi-)automatica
  - attiva di default
  - `<TAB>` indenta la riga corrente
  - `C-c C-q` indenta il "blocco" corrente (funzione, metodo, ...)
  - `M-q` formatta un paragrafo
- matching delle parentesi
  - `M-x show-paren-mode`

Emacs: finestre multiple

Gestione buffer

- `C-x b` cambia buffer
- `C-x k` chiudi un buffer

Gestione finestre

- `C-x 2` dividi orizzontalmente
- `C-x 3` dividi verticalmente
- `C-x 0` chiudi la finestra corrente (merge)
- `C-x 1` chiudi tutto tranne la finestra corrente
- `C-x o` vai ad altra finestra ("other")

Emacs: compilazione

Emacs è in grado di eseguire un comando di compilazione esterno e di interpretare eventuali messaggi di errore

- `M-x compile <RET> <comando-di-compilazione> <RET>`
  compila
- `javac <nome-file>`
  per i programmi Java

Al termine della compilazione, i messaggi vengono memorizzati in un buffer `*compilation*`

- `C-x ‘`
  vai al messaggio di errore successivo

Vim

- autore: Bram Moolenaar
- licenza: proprietaria (compatibile con GPL)
- architettura: C + vari interpreti
- editor modale
- esteso supporto per linguaggi di programmazione (syntax highlighting, shortcut, completamento stringhe, . . .)
- estendibile con vimscript (o altri linguaggi di scripting)

Vim: attenzione ai modi

Vim è un editor modale

- in ogni istante l'editor si trova in un solo modo
- l'interfacciamento con l'editor cambia a seconda del modo, ovvero:
  lo stesso comando può avere effetti diversi a seconda del modo
- modi principali:
  - normale (command mode)
  - inserimento (insert mode)
  - selezione (visual mode)
- per cambiare modo:
  - `i` command mode $\longrightarrow$ insert mode
  - `<ESC>` insert mode $\longrightarrow$ command mode
  - `v` oppure `V` command mode $\longrightarrow$ visual mode

Vim: primi passi

Invocazione

- `$ vim` interfaccia testuale
- `$ gvim` GUI (in ambiente grafico, ad es. terminale X)
- `$ vim <nome-file>`
  `$ gvim <nome-file>`
  per iniziare a lavorare su un file (esistente o meno)

Terminazione (occorre essere in command mode)

- `:wq<RET>` oppure `ZZ` salva ed esci
- `:q` esci senza salvare (ma solo se il buffer non ha modifiche)
- `:q!` esci senza salvare (anche se il buffer ha modifiche)

Vim: help

- `$ vimtutor`
  tutorial nella lingua di default
- `$ vimtutor it`
  tutorial in italiano

Il tasto delle emergenze:

- `<ESC>`

Vim: file di configurazione

`~/.vimrc`

(è un file nascosto, usate `ls -la` per vederlo)

Contiene una sequenza di comandi "ex"

Ogni comando "ex" viene utilizzato all'interno di Vim con la sintassi `:<nome-comando>` e può essere aggiunto al file di configurazione senza `:`

Vim: movimento del "cursore"

Di un carattere/riga:

- con i tasti cursore (in qualsiasi modo)
- `h j k l` (sx, giù, su, dx, solo in command mode)

Di una "parola":

- `w b` (avanti, indietro, solo in command mode)

Di una "pagina":

- `C-f` (command mode) oppure "page down" (ogni modo) avanti
- `C-b` (command mode) oppure "page up" (ogni modo) indietro

Cambiamento di modo + movimento:

- `I` insert mode a inizio riga
- `A` insert mode a fine riga (append)
- `a` append (insert mode al carattere successivo)

Iterazione comando (in command mode)

- n `<comando>`

Vim: editing di base I

- inserimento e cancellazione: prima occorre passare in insert mode, poi come nei normali word processor
- cancellazione
  - `dw` delete word
  - `x` oppure `<DEL>` delete char (dopo il cursore)
- copia & incolla, terminologia:
  - delete (cancella)
  - yank (copia)
  - put (incolla)
- copia & incolla, comandi:
  - `dd` delete line
  - `p` (in command mode, incolla dopo il carattere corrente)
  - `P` (in command mode, incolla a partire dalla linea corrente)

Vim: editing di base II

- selezione:
  - `v` o `V` per selezionare (in visual mode)
  - `d` (cancella selezione)
  - `y` (yank)
- `u` undo
- `C-R` redo

Vim: gestione file

- `:e <nome-file>` apri file (e chiudi quello corrente)
- `:w` salva il file corrente
- `:w <nome-file>` salva con nome

Vim: ricerca e sostituzione

Ricerca

- tutte le ricerche sono di espressioni regolari
- `/ <regexp-da-cercare>` ricerca
- `:set incsearch` abilita ricerca incrementale
- case sensitive a meno di `\c` in fondo al testo da cercare
- `n` trova successivo
- `N` trova precedente

Sostituzione

- `:%s/<regexp-da-cercare>/<testo-da-sostituire>`

Vim: supporto alla programmazione

- riconoscimento automatico del tipo di file sorgente (dall'estensione)
- syntax highlighting
  - `:syntax on` per attivarla (`off` per disattivarla)
  - aggiungere `:syntax on` nel `.vimrc`
- indentazione automatica
  - è automatica e attiva di default
- matching delle parentesi
  - `:set showmatch` per attivarlo (aggiungerlo nel `.vimrc`)
  - `%` passa da una parentesi aperta (chiusa) alla corrispondente che la chiude (apre), solo in command mode

Vim: finestre multiple

- `:q` chiudi la finestra corrente
- `:sp` dividi orizzontalmente la finestra corrente
- `C-w j` vai alla finestra sotto
- `C-w k` vai alla finestra sopra

Vim: compilazione

Vim è in grado di eseguire un comando di compilazione esterno e di interpretare eventuali messaggi di errore

- `:make` compila
- il comando di compilazione è definito nella variabile di Vim `makeprg`. Per settarla `:set makeprg=<comando>`
- nel nostro caso `:set makeprg=javac\ %`

Al termine della compilazione:

- `:copen` apre la finestra di compilazione
- `:cclose` chiude la finestra di compilazione
- `:cn` (`:cN`) vai all'errore successivo (precedente)

Compilatori Java (I)

Compilatore bytecode: Java $\longrightarrow$ bytecode

- `javac <nome-classe>.java`
  compila una classe Java e produce un file `<nome-classe>.class`
- `java <nome-classe>`
  esegue una classe compilata

Creazione di archivi `.jar`

- `jar cvf A.jar A.class B.class ...`
  crea un archivio contenente un insieme di classi compilate
- `java -classpath A.jar B ...`
  esegue il metodo `B.main` contenuto in una classe nell'archivio `A.jar`

Compilatori Java (II)

Compilatore nativo: Java $\longrightarrow$ programma eseguibile

- `gcj -c <nome-classe>.java`
  compila una classe Java e produce un file `<nome-classe>.o`
- `gcj --main=<nome-classe> <nome-classe>.java`
  compila una classe Java e produce un programma eseguibile `a.out`
- usare l'opzione `-o <nome-file>` per definire il nome del file eseguibile generato
- `gcj --main=<nome-classe> <nome-classe>.o...`
  effettua il linking dei file `.o` e genera un programma eseguibile

make e javac

Il programma `make` consente di specificare una volta per tutte le operazioni che portano alla compilazione di un programma

Tali operazioni vengono specificate in un `Makefile` (o `makefile`):

```
A.class: A.java
        javac A.java
clean:
        rm -f *.class
```

`A.class` e `clean` sono i target delle regole specificate

Dopo il `:` seguono le dipendenze

Nella riga successiva, preceduti da un carattere di tabulazione, compaiono i comandi che consentono di costruire il target

make e gcj

Un esempio di Makefile per compilare un programma Java con gcj:

```
hello: A.o
        gcj --main=A -o hello A.o

A.o: A.java
        gcj -c A.java

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

make: esecuzione

In presenza di un file `Makefile` o `makefile`, il comando `make` cerca di ottenere il primo target

Con `make -f` è possibile specificare un `Makefile` diverso

La variante `make` *`target`* cerca di ottenere il target specificato

Con `make -n` è possibile esaminare la sequenza di comandi che `make` eseguirebbe, senza eseguirli davvero

```
> make -n
gcj -c A.java
gcj --main=A -o hello A.o
```

make: variabili

Spesso è conveniente parametrizzare le regole in base a variabili

```
SOURCES = A.java B.java C.java
OBJS = A.o B.o C.o
hello: $(SOURCES)
        gcj --main=A -o hello $(SOURCES)
clean:
        rm -f hello $(OBJS)
```

Il riferimento `$(SOURCES)` viene espanso in `A.java B.java C.java`

Il riferimento `$(OBJS)` viene espanso in `A.o B.o C.o`

make: variabili calcolate

Le variabili possono essere definite in modo dipendente da altre variabili, usando delle sostituzioni:

```
SOURCES = A.java B.java C.java
OBJS = $(SOURCES:%.java=%.o)
hello: $(SOURCES)
        gcj --main=A -o hello $(SOURCES)
clean:
        rm -f hello $(OBJS)
```

Vedi "Functions for transforming text" nel manuale di make

make: raffinamento

In genere si specifica una regola diversa per ogni file sorgente:

```
SOURCES = A.java B.java C.java
OBJS = $(SOURCES:%.java=%.o)
hello: $(OBJS)
        gcj --main=A -o hello $(OBJS)
A.o : A.java
        gcj -c A.java
B.o : B.java
        gcj -c B.java
C.o : C.java
        gcj -c C.java
```

In questo modo le dipendenze sono associate più puntualmente ad ogni target

Come funziona `make`

Al momento dell'invocazione `make` *`target`*, il programma `make` calcola il grafo (aciclico!) delle dipendenze che occorrono per ottenere il target

- per ottenere `hello` servono i file `A.o B.o C.o`
- per ottenere `A.o` serve `A.java`
- per ottenere `A.java` non occorre nulla, il file è già disponibile
- i pre-requisiti di `A.o` sono soddisfatti, esegui il comando associato al target `A.o`
- ripeti per tutti i file `.o`
- i pre-requisiti di `hello` sono soddisfatti, esegui il comando associato al target `hello`

Come funziona `make`: tempo

`make` è pigro

Prima di eseguire il comando associato ad un target t, `make` verifica due cose:

1. il tempo T_t di ultima modifica del target t
2. il tempo T_d più recente di ultima modifica delle dipendenze del target t

Se $T_t > T_d$, il target è più recente di ciascuna delle sue dipendenze

Dunque, il target t non necessita di essere ricreato, ed il comando ad esso associato viene ignorato

Variabili automatiche

L'uso di variabili automatiche semplifica la scrittura delle regole, e si rende necessario nel momento in cui si usano regole implicite:

```
NAME = hello
SOURCES = A.java B.java C.java
OBJS = $(SOURCES:%.java=%.o)
$(NAME): $(OBJS)
        gcj --main=A -o $@ $(OBJS)
A.o : A.java
        gcj -c $<
B.o : B.java
        gcj -c $<
C.o : C.java
        gcj -c $<
```

Vedi sezione "Automatic Variables" nel manuale di `make`

Regole implicite

Una regola implicita definisce un pattern di regole

```
NAME = hello
SOURCES = A.java B.java C.java
OBJS = $(SOURCES:%.java=%.o)
$(NAME) : $(OBJS)
        gcj --main=A -o $@ $(OBJS)
%.o : %.java
        gcj -c $<
A.o : A.java
B.o : B.java
C.o : C.java
clean:
        rm -f $(NAME) $(OBJS)
```

make e directory multiple

Un progetto complesso strutturato in sotto-directory prevede normalmente un Makefile diverso per ogni directory

I Makefile possono "propagare" l'esecuzione in directory diverse tramite l'opzione -C di make

Supponendo di avere una sotto-directory src:

```
all:
        make -C src all
clean:
        rm -f hello
        make -C src clean
```