I nostri “IDE”: `emacs`, `vim`, `make`



Stefano Zacchiroli
<zack@cs.unibo.it>

5 Marzo 2004

Ma chi ca. . . volo sei?

(uny)identikit

- Dottore in Informatica
- Studente di Dottorato di Ricerca (sempre in Informatica . . .)
- Interessi di ricerca: Mathematical Knowledge Management, linguaggi di markup (XML e compagnia), Web Services, linguaggi di programmazione funzionali, . . . free software

e che ci fai qui?

- assistente alla didattica (cane da guardia, progetti)

Come contattarmi

- via mail: <zack@cs.unibo.it>
- ricevimento (programmato): da concordarsi . . . via mail!
- ricevimento (occasionale): cercatemi in "ufficio" dottorandi (ex-underlab, piano sottorraneo del dipartimento di scienze dell'informazione)
- homepage: `http://www.cs.unibo.it/~zacchiro/`
  in particolare
  `http://www.cs.unibo.it/~zacchiro/courses/labprog0304/`

Sommario

IDE vs Editor

Siamo programmatori (almeno quando ci tocca di realizzare progetti universitari) . . . ci servono strumenti adatti a ******** "programmi"!

******** =

- scrivere/modificare
- leggere
- compilare
- eseguire
- debuggare
- valutare le prestazioni
- . . .

Taaaaanti strumenti di sviluppo

La "vecchia" maniera =

Scrivere, modificare, leggere	`editor`
	+
Compilare	`compilatore`
	+
Eseguire	`shell`
	+
Debuggare	`debugger`
	+
Valutare le prestazioni	`profiler`
	+
...	...

The IDE way

Da `foldoc` (Free On-line Dictionary of Computing):

- IDE: integrated/interactive development environment
- interactive development environment:

  A system for supporting the process of writing software. Such a system may include a {syntax-directed editor}, graphical tools for program entry, and integrated support for compiling and running the program and relating compilation errors back to the {source}.

  Such systems are typically both interactive and integrated, hence the ambiguous acronym. They are interactive in that the developer can view and alter the execution of the program at the level of statements and variables. They are integrated in that, partly to support the above interaction, the source code editor and the execution environment are tightly coupled.

I(mprobably)DE

PROs:

+ integrazione (manco a dirlo) (i.e. nesun problema di comunicazione fra tool)
+ RAD (Rapid Application Development)
+ syntax directed editing molto potente

CONs:

- "pesantezza" (burp)
- troppo language specific (se in una settimana cambiate più linguaggi di programmazione che calzini . . . all'università vi capiterà!)
- troppo goal specific

Editor "dopati"

La "nuova vecchia" maniera =

editor in grado di orchestrare gli altri tool di sviluppo

Due esempi (dal mondo `*nix`):

- `emacs` 
  `http://www.gnu.org/software/emacs/emacs.html`
- `vim` 
  `http://www.vim.org/`

Che la guerra di religione abbia inizio!

Sommario

- IDE vs Editor
- introduzione a GNU Emacs
- introduzione a Vim
- introduzione a GNU Make
- a quando il progetto?

Emacs: highlights

- autore: Richard Stallman
- licenza: GPL
- architettura sw: C + Lisp
- editor non modale
- completamente (ri)programmabile in Lisp
- major mode
- minor mode
- embedding di altri processi

Emacs: primi passi

Invocazione:

- `$ emacs`
  GUI se possibile (e.g. eseguito da un terminale X) altrimenti interfaccia testuale
- per forzare l'interfaccia testuale: `$ emacs -nw`
  mnemonic: no windows
- editing di un file (esistente o meno): `$ emacs <nome_file>`

Uscita:

- esci senza salvare `C-x C-c`
  chiede conferma se esistono modifiche non salvate

Emacs: autoapprendimento

The Emacs tutorial:

- `$ emacs`
- `C-h t`

Emacs: prima dei primi passi

Notazione: C-, M-, SPC

Emacs: movimento del cursore (motion)

- the easy way: usa le frecce, Luke!
- the powerful way:
  - a piccoli passi: `C-p`, `C-n`, `C-b`, `C-f`
    mnemonic: previous, next, backward, forward
  - parola per parola: `M-b`, `M-f`
  - schermate: `C-v`, `M-v`
- tip of the day:
  provate `C-u <numero> <motion>` (e.g. `C-u 8 C-f`)

Emacs: file di configurazione

Nella vostra home: `.emacs`
(attenzione: dato che inizia con il carattere "." è un file nascosto, dovete usare `ls -a` per vederlo)

È un dialetto di Lisp chiamato elisp, per maggiori info `$ info elisp`

Emacs: editing di base

- iserimento, cancellazione: a la word processor
- cut and paste, terminologia:
  - killing (taglia)
  - yanking (incolla)
- cut and paste, comandi:
  - kill line: `C-k`
  - selezione: `C-SPC` (tip: `(transient-mode-mark 1)` nel .emacs)
  - kill selezione corrente: `C-w`
  - yank: `C-y`
  - copia = kill + yank
- undo: `C-x u` o `C-_`

Emacs: gestione dei file

- find file (apri): `C-x C-f`
  funziona il completamento con TAB
- salva file corrente: `C-x C-s`
- write file (salva con nome): `C-x C-w`

Emacs: ricerca e sostituzione

Ricerca

- ricerca incrementale: `C-s <testo_da_cercare>`, `C-r <testo_da_cercare>`
- ricerca di espressioni regolari: `C-M-s`
- case sensitivity automatica

Sostituzioni

- sostituzione di testo:
  `M-x replace-string <RET> <testo_da_cercare> <RET> <testo_da_sostituire> <RET>`
- sostituzione di regexp:
  `M-x replace-regexp <RET> <regexp_da_cercare> <RET> <testo_da_sostituire> <RET>`

Emacs: piccole gioie del programmatore

- riconoscimento automatico del tipo di file sorgente (dall'estensione)
- syntax highlight
  - non attivo di default, per attivarla aggiungete `(global-font-lock-mode 1)` al vostro .emacs
- indentazione automatica
  - attiva di default
  - in emacs è in realta semi automatica, per invocarla usate `<TAB>` (indenta la riga corrente)
- matching delle parentesi
  - attivo di default: provare . . . per credere

Emacs: cenni sulle finestre

- chiudi la finestra corrente: `C-x 0`
- nascondi tutte le finestre tranne la corrente: `C-x 1`
- dividi in due (orizzontalmente) la finestra corrente: `C-x 2`
- passa ad un'altra delle finestre visibili: `C-x o`
  mnemonic: other

Emacs: compilazione

Emacs è in grado di eseguire un comando di compilazione esterno e di interpretare gli eventuali errori di compilazioni

- compila: `M-x compile <RET> <comando_di_compilazione>`
- nel nostro caso il comando di compilazione è `javac <nome_del_file>`

al termine della compilazione, l'output della stessa viene riportato nella finestra di compilazione `*compilation*` e interpretato per eventuali errori di compilazione

- vai al successivo errore di compilazione: `C-x ‘`

Emacs: auto. . . approfondimento

Help in linea: Menu Help → Read the Emacs Manual

Sommario

- IDE vs Editor
- introduzione a GNU Emacs
- introduzione a Vim
- introduzione a GNU Make
- a quando il progetto?

Vim: highlights

- autore: Bram Moolenaar
- licenza: proprietaria (GPL compatibile)
- architettura sw: C (+ intepreti vari)
- editor modale
- programmabile in vimscript (+perl, +python, +tcl, +ruby, ...)

Vim: prima dei primi passi

Vim è un editore modale!:

- in ogni istante Vim è in una e una sola modalità (mode)
- l'interfacciamento con l'editor cambia a seconda della modalità
- modalità principali:
  - Modalità normale (command mode)
  - Modalità di inserimento (insert mode)
  - Modalità di rimpiazzo (replace mode)
  - Modalità visuale (visual mode)

PROs: shortcut più semplici

CONs: context switch

Vim: prima dei primi passi

Notazione: <ESC>, <RET>, CTRL-, SHIFT-, ...

- command mode → insert mode: `i`
  (o altro comando di inserimento)
- insert mode → command mode: `<ESC>`

Vim: primi passi

Invocazione:

- GUI: `$ gvim`
  mnemonic: GUI
- interfaccia testuale: `$ vim`
- editing di un file (esistente o meno): `$ vim <nome_file>`, `$ gvim <nome_file>`

Uscita:

- esci e salve: (command mode) `:wq<RET>` o `ZZ`
- esci senza salvare: (command mode) `:q`
  per forzare l'uscita nel caso di modifiche non salvate: `:q!`

Vim: autoapprendimento

The Vim tutor:

- `$ vimtutor` (nella lingua di default)
- `$ vimtutor it` (in italiano)

Vim: movimento del cursore (motion)

- the easy way: usa le frecce, Luke! (funzionano in qualsiasi modalità)
- the powerful way:
  - a piccoli passi: `h`, `j`, `k`, `l` (command mode)
    mnemonic: la `j` "punta" verso il basso
  - parola per parola: `w`, `b` (command mode)
    mnemonic: word, backword
  - schermate: `CTRL-f`, `CTRL-b`
- tip of the day: provate `<numero>` `<motion>` (e.g. `8 l`)

Vim: file di configurazione

Nella vostra home: `.vimrc`

È una sequenza di comandi Ex (Ex command).

Ogni comando Ex viene utilizzato all'intero di Vim con la sintassi `:<nome_comando>` e può essere aggiunto al file di configurazione senza i ":" iniziali.

Vim: editing di base

- inserimento, cancellazione: prima si passa alla modalità di inserimento, poi a la word processor
- cut and paste, terminologia:
  - delete (taglia)
  - yank (copia)
  - put (incolla)
- cut and paste, comandi:
  - delete line: `dd`
  - (visual mode) selezione: `v`, `V`, `CTRL-v`
  - azioni sulla selezione: `d` (delete), `y` (yank)
  - put (command mode): `p` (incolla dopo il carattere corrente)
- undo (command mode): `u`

Vim: gestione dei file

- edit file (apri): `:e <nome_file>` (chiude anche il file corrente) funziona il completamento con TAB
- save (salva file corrente): `:w`
- salva con nome: `:w <nome_file>`

Vim: ricerca e sostituzione

Ricerca

- tutte le ricerche sono di espressioni regolari
- ricerca: `/ <regexp_da_cercare>`
- ricerca incrementale: aggiungete `set incsearch` al .vimrc
- le ricerche sono case sensitive di default, aggiungete `\c` in fondo al testo da cercare per ricerche case insensitive

Sostituzioni

- sostituzione (di regexp:)
  `:%s/<regexp_da_cercare>/<testo_da_sostituire>`

Vim: piccole gioie del programmatore

- riconoscimento automatico del tipo di file sorgente (dall'estensione)
- syntax highlight
  - non attivo di default, per attivarla aggiungete `syntax on` al vostro .vimrc
- indentazione automatica
  - è automatica e attiva di default
- matching delle parentesi
  - non attivo di default: aggiungete `set showmatch` al vostro .vimrc
  - motion da una parentesi alla parentesi che la "completa" (command mode): `%`

Vim: cenni sulle finestre

- chiudi la finestra corrente: `:q`
- dividi in due (orizzontalmente) la finestra corrente: `:sp`
- passa alla finestra al di sotto della corrente: `C-w j`
- passa alla finestra al di sopra della corrente: `C-w k`

Vim: compilazione

Vim è in grado di eseguire un comando di compilazione esterno e di interpretare gli eventuali errori di compilazioni.

- compila: `:make`
- 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, l'output della stessa viene interpretato per eventuali errori di compilazione.

- apri la finestra di compilazione: `:copen`
- chiudi la finestra di compilazione: `:cclose`
- vai al successivo/precedente errore di compilazione: `:cn`, `:cN`

Vim: auto. . . approfondimento

Help in linea: `:help`

Sommario

- IDE vs Editor
- introduzione a GNU Emacs
- introduzione a Vim
- introduzione a GNU Make
- a quando il progetto?

Processo di compilazione

Un tipico processo di compilazione:

- creazione dei sorgenti
- creazione degli oggetti (tipicamente 1 oggetto per ogni sorgente)
- linking (generando un solo eseguibile da tanti oggetti)

Qual è il problema? . . . le dipendenze!

Nel caso in cui un sorgente venga modificato, non solo l'oggetto corrispondente deve essere ricreato, ma anche tutti gli oggetti che (transitivamente) dipendono da esso.

GNU Make si prefigge la semplificazione di questo processo, (ri)compilando ad ogni invocazione solo i sorgenti che necessitano di (ri)compilazione (o perché sono stati modificati o perché alcuni dei file da cui dipendono lo sono stati).

GNU Make: macro

Ad ogni esecuzione make cerca un Makefile, tipicamente questo deve essere trovato nella directory corrente in un file denominato `Makefile`.

Un Makefile è un file testuale contenente una lista di definizione. Esistono due tipi di definizioni: le macro e le regole (rule).

Le macro vengono definite con la sintassi: `nome = valore`. L'azione di ottenere il valore associato ad una macro prende il nome di sostituzione di macro e si effettua con la sintassi: `$(nome)`. La sostituzione di una macro può essere effettuata in qualsiasi posizione del Makefile, compresa la definizione di altre macro.

Esempio di macro:

```
JAVADIR = /usr/bin
JAVAC = $(JAVADIR)/javac
```

GNU Make: regole

Le regole stabiliscono l'effettivo comportamento di make ad ogni sua invocazione. Ogni regola definisce uno o più obiettivi (target).

All'atto dell'esecuzione, make viene invocato specificando a riga di comando uno o più target. L'esecuzione di make processa sequenzialmente le regole associate ad ognuno degli obiettivi specificati.

Esempio di invocazione:

```
$ make Hello.class
```

GNU Make: regole

Ogni regola è composta da tre componenti:

- obiettivi
- dipendenze
- comandi

Esse sono specificabile all'interno di un Makefile con la seguente sintassi:

```
obiettivo ...:  [dipendenza ...]
<TAB>comando
...
```

Ogni comando inizia sempre con un carattere di tabulazione.

GNU Make: come funge

Il caso semplice: obiettivi senza dipendenze

Ogni volta che un obiettivo viene considerato, make controlla se esiste un file su disco corrispondente all'obiettivo (i.e. con lo stesso nome). Se tale file esiste il processo di compilazione termina (l'obiettivo è già stato raggiunto).

Se tale file non esiste i comandi associati all'obiettivo vengono eseguiti.

Ai fini del corretto funzionamento, si assume quindi che la sequenza di comandi associati ad un obiettivo crei un file corrispondente all'obiettivo stesso.

Esempio di makefile senza dipendenze:

```
Hello.class:
	$(JAVAC) Hello.java
```

GNU Make: come funge

Il caso complesso: obiettivi con dipendenze

Ogni volta che un obiettivo viene considerato, l'algoritmo descritto in precedenza viene ricorsivamente eseguito sulle dipendenze dell'obiettivo (secondo una politica depth-first) fino a quando non si incontra un obiettivo senza dipendenze (i cui comandi associati vengono eseguiti) o un caso base della ricorsione.

Un caso base è costituito da un obiettivo che ha come dipendenze uno o più file presenti su disco. Se tali file sono più recenti (secondo la data di ultima modifica del file) del file associato all'obiettivo corrente, i comandi associati vengono eseguiti, altrimenti no.

Una volta terminata la ricorsione, se almeno una delle dipendenze ha richiesto esecuzione del comando associato, allora i comandi associati all'obiettivo corrente vengono eseguiti per ricreare il file associato.

GNU Make: come funge

Esempio di makefile con dipendenze:

```
Hello.class:  Hello.java
	$(JAVAC) Hello.java
```

GNU Make: obiettivi effimeri

Risulta spesso comodo avere obiettivi che non sono associati a nessun file su disco, tali obiettivi prendono il nome di obiettivi effimeri (phony target).

Ogni volta che make processa una regola associata ad uno di questi obiettivi, processa ricorsivamente le dipendenze ed esegue i comandi associati indipendentemente dalla necessita di ricreare l'obiettivo. Esempi classici di obiettivi effimeri sono: `all` e `clean`.

Per segnalare a make quali obiettivi sono effimeri occorre aggiungerli come dipendenze dell'obiettivo predefinitio `.PHONY`.

Esempio di makefile con obiettivi effimeri:

```
all:  Hello.class
clean:
	rm -f Hello.class
.PHONY: all clean
```

GNU Make: make & Java

Il caso di Java è più semplice, dato che il compilatore java effettua automaticamente (o almeno ci prova) il controllo della necessità di ricompilare un file `.class` o meno.

Un Makefile tipo per il progetto potrebbe quindi essere:

```
...
all: Main.class
Main.class: *.java
        $(JAVAC) main.java
clean:
        rm -f *.class
run:
 $(JAVA) -cp . Main
.PHONY: all clean run
```

Sommario

- IDE vs Editor
- introduzione a GNU Emacs
- introduzione a Vim
- introduzione a GNU Make
- a quando il progetto?

Informazioni preliminari sul progetto

- specifiche prima delle vacanze pasquali

nel frattempo:

- il progetto dovrà (ovviamente) essere implementato in Java ...
- ... utilizzando gli strumenti che volete, ma sono fortemente consigliati quelli che abbiamo visto in questa lezione. In sede di discussione del progetto sarete valutati anche sulla vostra abilità nell'uso di tali strumenti (un editor a scelta + make)
- il progetto sarà svolto a gruppi di 3 persone

È tutto