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 III: Gruppi di lavoro agili

Software Configuration Management (SCM)

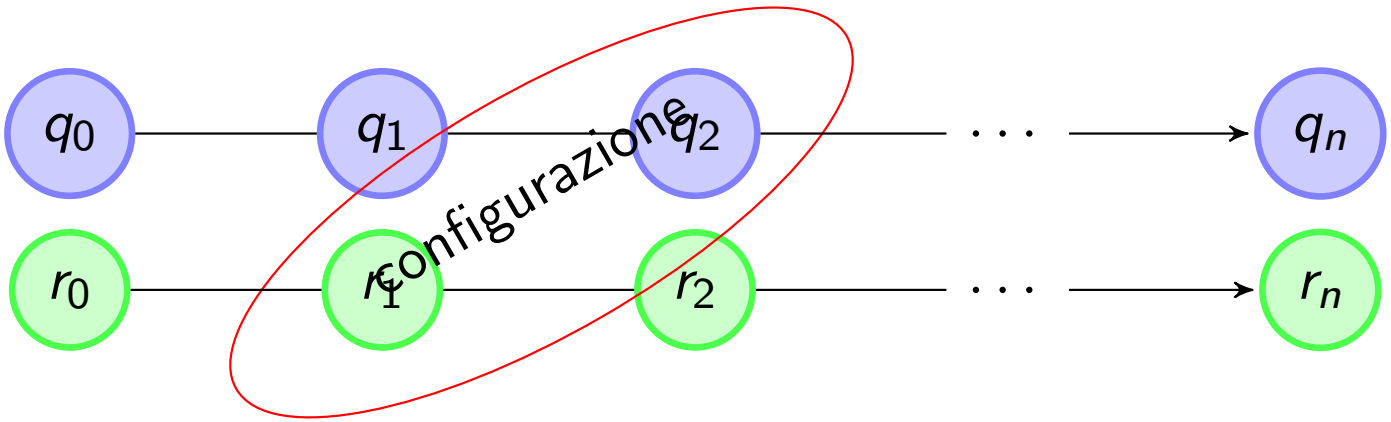
Il *Configuration Management* nasce nell'industria aerospaziale negli anni '50. Alla fine degli anni '70 inizia a essere applicato nella produzione del *software*.

Software Configuration Management

Pratiche che hanno l'obiettivo di rendere sistematico il processo di sviluppo, tenendo traccia dei cambiamenti in modo che il prodotto sia in ogni istante in uno stato (configurazione) ben definito.

Gli "oggetti" di cui si controlla l'evoluzione sono detti *configuration item* o (in ambito *sw*) *artifact*.

SCM: di cosa si occupano



- Gli *artifact* sono *file* o più raramente *directory*
- l'SCM permette di tracciare/controllare le revisioni degli *artifact* e le versioni delle risultanti configurazioni
- a volte fornisce supporto per la generazione del prodotto a partire da una ben determinata configurazione

Gli strumenti di SCM

Gli SCM sono per lo più indipendenti da linguaggi di programmazione e applicazioni (una notevole eccezione è Monticello di Smalltalk): lavorano genericamente su *file*, preferibilmente fatti di righe di testo.

- anni '80: strumenti locali (SCCS, rcs, . . .)
- anni '90: strumenti *client-server* centralizzati (cvs, subversion, . . .)
- anni 2000: strumenti distribuiti *peer-to-peer* (git, mercurial, ...)

Due decisioni importanti

Qualunque sistema si usi, occorre prendere due decisioni importanti, che influenzano la replicabilità della produzione. In entrambi i casi la risposta più comune è no, ma in questo caso la perfetta replicabilità è perduta.

1. Si traccia l'evoluzione anche di componenti fuori dal nostro controllo? (librerie, compilatori, ecc.)
2. Si archiviano i *file* che costituiscono il prodotto?

(1) è molto costoso, (2) è spesso poco pratico.

Il meccanismo base

Il meccanismo di base per controllare l'evoluzione delle revisioni è che ogni cambiamento è regolato da:

1. check-out dichiara la volontà di cambiare un determinato *artifact*
2. check-in (o commit) dichiara la volontà di registrare un determinato *change-set*

Queste operazioni vengono attivate rispetto a un'applicazione di *repository*

La regolazione del lavoro concorrente

Quando il *repository* è condiviso da un gruppo di lavoro, nasce il problema di gestirne l'accesso concorrente:

- *modello "pessimistico"* (rcs) il sistema gestisce l'accesso agli *artifact* in mutua esclusione attivando un *lock* al *check-out*
- *modello "ottimistico"* (cvs e successivi) il sistema si disinteressa del problema e fornisce supporto per le attività di merge di *change-set* paralleli.

Il modello pessimistico è, nello sviluppo *software*, tanto irrealistico e ideale quanto il processo a "cascata". Il modello ottimistico può però essere parzialmente regolato tramite i rami paralleli di sviluppo (*branch*).

Il *merge*

Il *merge* rimane un'operazione delicata. Generalmente vengono trattati con strategie diverse:

- lavoro parallelo su *artifact* diversi
- lavoro parallelo sullo stesso *artifact*: *hunk* differenti
- lavoro parallelo sullo stesso *artifact*: *hunk* uguali

L'ultimo caso necessita *sempre* di lavoro intelligente. Nel resto dei casi, dipende.

La terminologia per i *merge*

La terminologia più usata fa riferimento alla coppia di programmi POSIX `diff` e `patch`

diff o *patch* la differenza fra due revisioni (R, R'), calcolata per righe, cercando di minimizzare il numero di inserimenti e cancellazioni (ricerca della sottosequenza più lunga)

$R =$	a	b	c	d	f	g	h	j	q	z				
$R' =$	a	b	c	d	e	f	g	i	j	k	r	x	y	z

L'intersezione è a b c d f g j z

e	h	i	q	k	r	x	y
+	-	+	-	+	+	+	+

hunk Il diff è diviso in "fette" (nell'esempio sarebbero `e,hi,q,krxy`)

`patch` è il programma che permette di applicare il diff non solo a R per ottenere R', ma anche a un qualunque R^* "vicino" a R (che diventerà un $R^{*\prime}$), applicando alcune euristiche per ogni *hunk*.

3-way merge

Quando, come nel caso di lavoro parallelo sullo stesso *artifact*, le due revisioni hanno un antenato comune (per esempio la revisione da cui entrambi sono partiti) si può facilitare il lavoro di *merge*. Siano A' e A^* due revisioni, con antenato comune A.

- *hunk* comuni a tre delle tre revisioni: inalterato
- *hunk* comuni a due delle tre revisioni
  - comuni a A' e A^*: *merge*
  - comuni a A e A' (o A^*): *merge* A^* (o A')
- il resto deve essere valutato a mano