



UNIVERSITÀ DEGLI STUDI
DI MILANO

SVILUPPO SOFTWARE IN GRUPPI DI LAVORO COMPLESSO

prof. Carlo Bellettini

prof. Mattia Monga

a.a. 2021-22



UNIVERSITÀ DEGLI STUDI
DI MILANO

8. Intro Software configuration management

Source code management (versioning)

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.

M. FOWLER

Linus presenta git a Google

Tech Talk: Linus Torvalds on git



Software Configuration Management

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

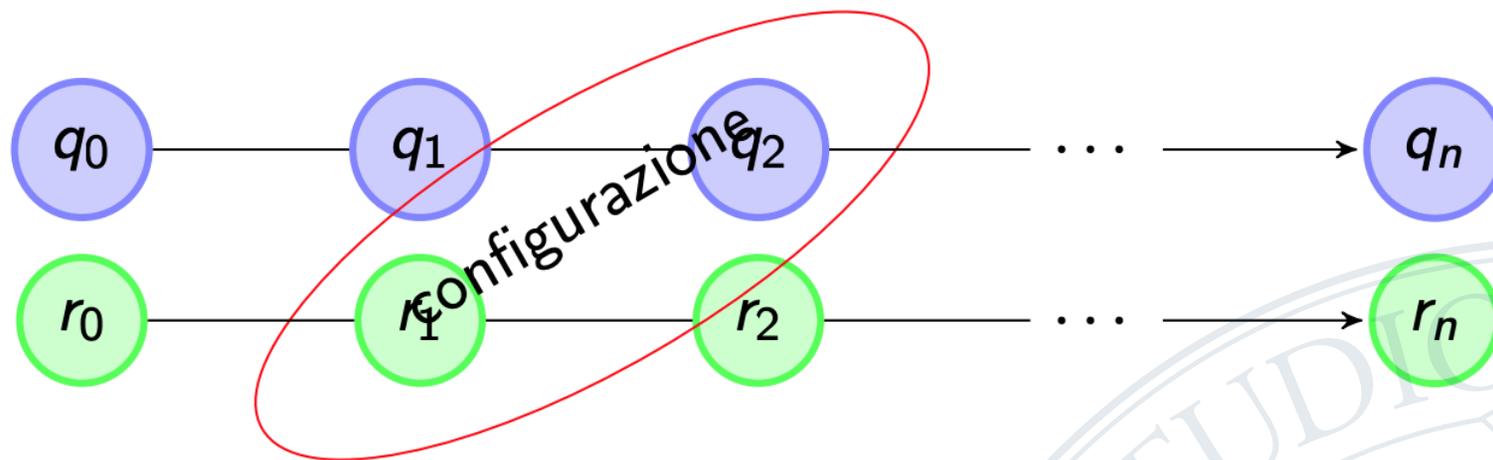
Pratiche che hanno l'obiettivo di rendere sistematico il processo di sviluppo, **tenendo traccia dei cambiamenti** in modo che il prodotto sia in ogni istante in uno stato (configurazione) ben definito.

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

Tre scenari di esempio

- Programmatore solo
- Collaborazione Studente professore
- Blaming

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

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 calcola la differenza fra due revisioni (R0, R1), calcolata per righe, cercando di minimizzare il numero di inserimenti e cancellazioni (ricerca della sottosequenza più lunga)

R0 = a b c d f g h j q z

R1 = 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

+ - + - + + + +

Il diff è diviso in “fette” **hunk**

e, hi, q, krxxy

patch è il programma che permette di applicare il diff non solo a R0 per ottenere R1, ma anche a un qualunque R0' “vicino” a R0 (che diventerà un R1'), 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* uguale nelle tre revisioni: inalterato
- *hunk* uguale in due delle tre revisioni
 - A' e A'' uguali: merge (A')
 - A e A' uguali: merge A''
 - (A e A'' uguali: merge A')
- *hunk* diverso nelle tre revisioni deve essere valutato a mano



UNIVERSITÀ DEGLI STUDI
DI MILANO

8b. Git

Versioning distribuito

- Internals
- Architettura
- Alcuni comandi un po' più avanzati: History rewriting
- Alcuni workflow

Formato degli objects

```
def put_raw_object(content, type)
  size = content.length.to_s

  header = "#{type} #{size}\0" # type(space)size(null byte)
  store = header + content

  sha1 = Digest::SHA1.hexdigest(store)
  path = @git_dir + '/' + sha1[0...2] + '/' + sha1[2..40]

  if !File.exists?(path)
    content = Zlib::Deflate.deflate(store)

    FileUtils.mkdir_p(@directory+'/' + sha1[0...2])
    File.open(path, 'w') do |f|
      f.write content
    end
  end
  return sha1
end
```