Source Code Management Systems

Ingegneria del Software L-A

Contesto

- Sviluppo distribuito (es. SourceForge)
- Software House
- Singolo sviluppatore
- Studente…

Sinonimi

- Version Control
- Source Code Management (SCM)
- Source Control

→ Gestione dei codici sorgenti e delle versioni

In poche parole

Un SCM:

- Fornisce supporto alla memorizzazione dei codici sorgenti
- Fornisce uno storico di ciò che è stato fatto
- Può fornire un modo per lavorare in parallelo su diversi aspetti dell'applicazione in sviluppo
- Può fornire un modo per lavorare in parallelo senza intralciarsi a vicenda

→ Fornisce un modello di sviluppo
→ Aumenta la produttività

Best Practice

Usare un sistema di Source Control anche per lo sviluppo personale…

…a volte salva "la vita".

Concetti di base (I)

- ***Repository***: un posto dove memorizzare i sorgenti
  - Tipicamente si trova su una macchina remota affidabile e sicura
  - Tutti gli sviluppatori condividono lo stesso repository
- ***Working folder***: cartella di lavoro
  - Ogni sviluppatore ne ha una collocata sulla propria macchina
  - Contiene una copia del codice sorgente relativo al progetto

Concetti di base (II)

- Sia il *Repository*, sia la *Working Folder* sono una gerarchia di cartelle o *directory*
- Il *workflow* di uno sviluppatore è tipicamente:
  - Copiare i contenuti del *repository* nella *working folder*
  - Modificare o aggiungere codice nella *working folder*
  - Aggiornare il *repository* incorporando i cambiamenti e le aggiunte
  - Ricominciare da capo…

Concetti di base (III)

Evitare di rompere il ciclo di sviluppo (***brake the tree*** – letteralmente rompere la gerarchia di cartelle)

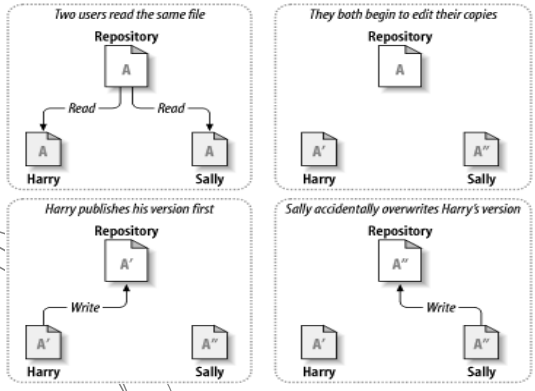
→ Il codice che viene memorizzato sul *repository* deve portare il progetto in uno stato che consenta a chiunque (nel team) di proseguire nello sviluppo

→ Evitare di memorizzare codice che non compila o che non passa i test altrimenti tutto il team è costretto ad interrompere il ciclo di sviluppo

Repository = Time Machine

- Il *Repository* è un archivio di ogni versione di ogni file di codice sorgente
- Contiene la "storia" del progetto
- Rende possibile navigare indietro nel tempo e recuperare versioni vecchie dei file
  - Capire perché sono state fatte certe scelte
  - Capire perché sono stati introdotti nuovi bug

Il problema da evitare



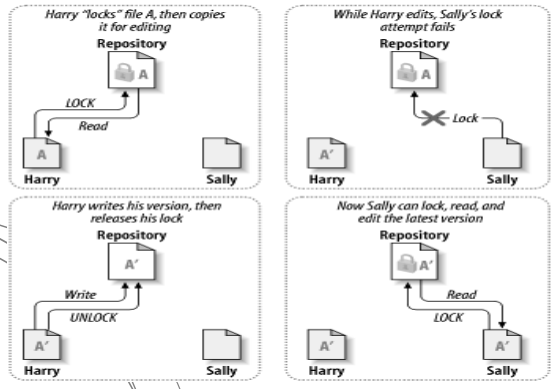
Modello *Lock-Modify-Unlock* (I)

- Tutti i file nella *working folder* sono inizialmente *read-only*
- L'utente A scarica o aggiorna un file sorgente nella sua *working folder*, lo rende *writable* e pone un *lock* su quel file nel *repository* → *checkout*
- L'utente A modifica la sua copia del file (che è *writable*)
- L'utente B non può a sua volta porre un *lock* sul file, al più può ottenerne una copia *read-only*
- L'utente A termina le modifiche sul file, salva il file sul *repository*, rende *read-only* il file nella *working folder* e toglie il *lock* → checkin
- L'utente B può ora porre un suo *lock* sul file per modificarlo

Modello *Lock-Modify-Unlock* (II)

- Uno sviluppatore deve effettuare un *checkout* prima di modificare un file → tutti sanno chi sta modificando cosa
- I *checkout* sono effettuati con *lock* esclusivi → solo uno sviluppatore alla volta può modificare un file

Modello *Lock-Modify-Unlock* (III)



Modello *Lock-Modify-Unlock* (IV)

Problemi:

- Problemi di amministrazione: un utente pone un *lock* poi se ne dimentica → ritardi e tempo perso
- Serializzazione non necessaria: un utente blocca un file e ne modifica una parte → impedisce ad un altro utente di modificare lo stesso file in una parte scorrelata dalla prima modifica
- Falso senso di sicurezza: un utente blocca un file e lo modifica, un altro utente blocca un altro file che dipende dal primo e lo modifica → le modifiche effettuate saranno compatibili?
- Altri problemi: come lavorare *offline*?

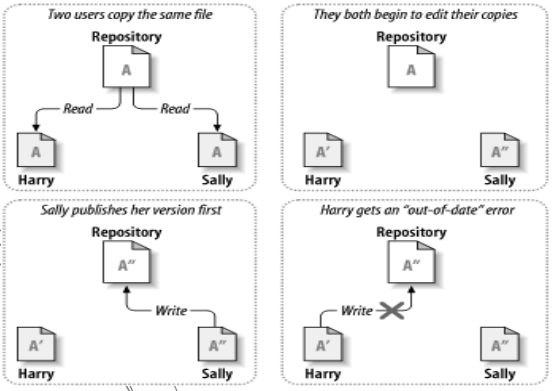
Modello *Copy-Modify-Merge* (I)

- Non esistono *lock* e tutti i file sono *scrivibili*
- L'utente copia i file del *repository* nella propria *working folder*
- Modifica i file che desidera nella propria *working folder*
- Richiede un *update* della propria *working folder* (nel frattempo, qualcun altro potrebbe aver modificato il *repository*)
  - I file modificati vengono "fusi" (*merge*) con quelli contenuti nel *repository* nella versione finale del file
  - L'operazione di *merge* è tipicamente fatta in modo automatico e il risultato è tipicamente positivo: dopo un *merge* occorre verificare il corretto funzionamento dell'applicazione (*testing*…)
  - In alcuni casi il *merge* non può essere fatto automaticamente (*conflitti* – modifiche sulla stessa linea da parte di diversi sviluppatori); in questo caso i *conflitti* vengono *risolti* con strumenti che aiutano ad evidenziare le modifiche proprie e di altri
- Il salvataggio nel *repository* consente di pubblicare le modifiche effettuate (*commit*)

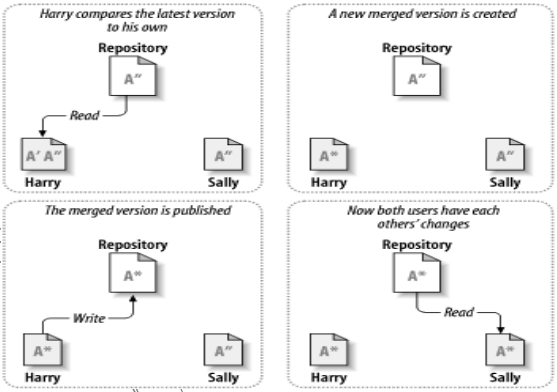
Modello *Copy-Modify-Merge* (II)

- Nessuno usa i *lock* quindi non si sa chi stia facendo cosa
- Il sistema si accorge se è necessario un *merge* oppure se è possibile salvare il file direttamente
- E' buona norma effettuare un *update* del contenuto della propria *working folder* prima di cominciare a lavorare
- Quando uno sviluppatore effettua un *commit,* è sua responsabilità assicurarsi che i cambiamenti siano stati effettuati sull'ultima versione contenuta nel *repository* (v. slide precedente)

Modello *Copy-Modify-Merge* (III)



Modello *Copy-Modify-Merge* (IV)



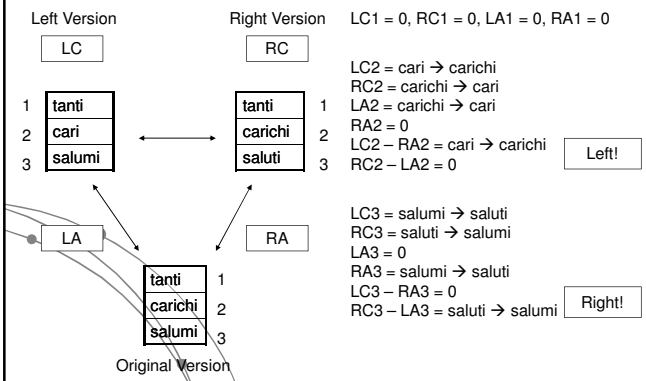
Come funziona il merge?

- Tre file in gioco:
  1. File contenuto nel repository (ultima versione)
  2. File con modifiche dello sviluppatore
  3. File originale prima delle modifiche dello sviluppatore (in *working folder*)
- Se 1. e 3. sono uguali non serve il merge – si copia la nuova versione sul repository
- Si considera 3. come il "file originale", 1. e 2. come file modificati rispetto all'originale

Algoritmo Codeville (semplificato senza numeri di versione)

- Esistono due versioni diverse dello stesso file che devono essere fuse: queste versioni siano etichettate con *left* e *right;* il file originale contenuto in working folder, è etichettato con *original*
- Come prima operazione si esegua un algorimo di *difference* a due vie che individui le differenze fra i due file *{left, right}* e che dia come risultato un insieme di sezioni che corrispondono (*matching*) o non corrispondono (*non matching*) nelle due diverse versioni – per ogni sezione *non matching* occorre determinare se vince la *left*, la *right* o se c'è un conflitto
- Siano LC (*Left Changes*) l'insieme dei cambiamenti nella versione *left* di una sezione *non matching* e RC (*Right Changes*) l'insieme dei cambiamenti nella corrispondente sezione della versione *right*
- Si riapplica il difference a due vie, questa volta su *{left, original}* e *{right, original}* e si determinano LA (*Left Changes Applied*) l'insieme dei cambiamenti applicati nella versione *left* e RA (*Right Changes Applied*) l'insieme dei cambiamenti applicati nella versione *right*
- Se LC – RA non è vuoto allora vince la versione *left*
- Se RC – LA non è vuoto allora vince la versione *right*
- Se entrambe le versioni dovrebbero vincere allora c'è un conflitto (modifica in entrambe le versioni) – richiesto l'intervento dell'utente

Codeville – RUN!



LC1 = 0, RC1 = 0, LA1 = 0, RA1 = 0

LC2 = cari → carichi
RC2 = carichi → cari
LA2 = carichi → cari
RA2 = 0
LC2 – RA2 = cari → carichi
RC2 – LA2 = 0

Left!

LC3 = salumi → saluti
RC3 = saluti → salumi
LA3 = 0
RA3 = salumi → saluti
LC3 – RA3 = 0
RC3 – LA3 = saluti → salumi

Right!

Dopo un *merge*…

- Dopo un *merge* il codice si compila sempre?
- Dipende: i cambiamenti di diversi sviluppatori, pur non entrando in conflitto secondo l'algoritmo di *merge*, possono generare inconsistenze finali
- Però… è abbastanza raro che ciò avvenga…
- Comunque… mai fidarsi!

La guerra dei mondi

- La dottrina "*lock-modify-unlock*" è più tradizionale e conservativa
- La dottrina "*copy-modify-merge*" è più rischiosa e libertaria
  - → Il rischio è che l'operazione di *merge* possa causare problemi e perdite di tempo
  - → L'accettazione del rischio (relativo) consente di avere uno stile di sviluppo concorrente che permette agli sviluppatori di interagire molto meno

Quale scegliere?

- Dipende dal progetto, dall'organizzazione e dagli sviluppatori
- Entrambi i modelli hanno pro e contro
- La leggenda dice che chi si è spostato dal mondo dei *lock* al mondo dei *merge* non torna più indietro

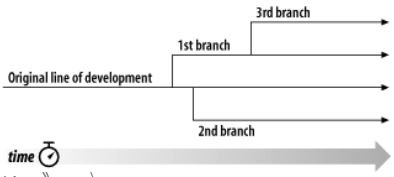
Trunk, Branch, Tag

Un modo ragionevole per organizzare un *repository* è fare in modo che contenga:

- Un tronco principale di sviluppo → *Trunk*
- Un luogo dove memorizzare le linee di sviluppo alternative → *Branch, Branches*
- Un luogo dove memorizzare le release stabili → *Tag, Tags*

Branch (I)

- Relativamente ad un progetto, un *branch* è una linea di sviluppo indipendente dalle altre
- Viene inizialmente generato come copia completa → condivide parte della storia



Branch (II)

- I *Branch*, ad esempio, consentono di iniziare lo sviluppo di una nuova *release* quando la precedente è ancora in fase di consolidamento
- Una volta terminato il consolidamento è possibile effettuare il *merge* fra il *branch* ed il *trunk*

  → I *bug* risolti in fase di consolidamento saranno fusi insieme alle modifiche apportate per incorporare le funzionalità della nuova *release*

Tag

- *Tag = Release*
- Viene memorizzato separatamente in modo da avere a portata di mano tutti i sorgenti relativi ad una certa *release*
- In questo modo non è necessario andare a ripescare dal *main trunk* i sorgenti andando indietro con le versioni dei file ← a quale versione di ogni file corrisponde una certa *release*?

Branch Merge

- Quanto può essere complicato fare il *merge* di un *branch*?
- Dipende

  → In media i *merge* possono essere abbastanza semplici e quasi automatici

  → In alcuni casi si sceglie di ripartire da una versione precedente ed incorporare le modifiche in modo più opportuno

Subversion & C.

- SCM Open Source di tipo *Copy-Modify-Merge*
  - http://subversion.tigris.org/
- TortoiseSVN → Client grafico per Windows
  - http://tortoisesvn.net/
- RapidSVN → Client grafico multipiattaforma
  - http://rapidsvn.tigris.org/
- AnkhSVN → Plugin per Visual Studio
  - http://ankhsvn.open.collab.net/
- Subclipse → Plugin per Eclipse
  - http://subclipse.tigris.org