Sistemi Cooperativi:
Sistemi di Sviluppo Cooperativo

Gabriele D'Angelo
<gda@cs.unibo.it>
http://www.cs.unibo.it/~gdangelo

Università degli Studi di Bologna
Dipartimento di Scienze dell'Informazione

Aprile, 2005

Scaletta della lezione

- I sistemi di sviluppo cooperativo: cosa si intende?
- Caso pratico: lo sviluppo del kernel Linux
- SCM:
  - alternative Open Source
  - chi è il migliore? Definiamo i requisiti
  - modelli architeturali (centralizzato o distribuito?)
  - pessimistico o ottimistico?
- Strumenti di SCM: CVS, Subversion
- GForge
- Bug Tracking Systems (es. Debian, Bugzilla)
- Importanza del building / testing / profiling

Sistemi di sviluppo cooperativo

- Cerchiamo di definire il problema con una buona visione d'insieme: si tratta solamente di **condividere** codice?

- Alcuni problemi:
  - gestione del codice, delle revisioni e dei rami di sviluppo
  - comunicazione tra sviluppatori
  - segnalazione, gestione e risoluzione dei bug
  - sistemi automatici di building/testing/profiling
  - strumenti integrati per lo sviluppo cooperativo

Una provocazione: kernel Linux

- Iniziamo immediatamente con una provocazione:

  qual'è l'ultima versione stabile del kernel Linux?

Una provocazione: kernel Linux

- Iniziamo immediatamente con una provocazione:

  qual'è l'ultima versione stabile del kernel Linux?

> finger @kernel.org

Una provocazione: kernel Linux

- Iniziamo immediatamente con una provocazione:

  qual'è l'ultima versione stabile del kernel Linux?

> finger @kernel.org **2.6.11.7**

Una provocazione: kernel Linux, ieri, oggi e domani

- Iniziamo immediatamente con una provocazione:

  qual'è l'ultima versione stabile del kernel Linux?

\> finger @kernel.org **2.6.11.7**

- Major, minor e revision non erano più sufficienti per rappresentare il modello attuale di sviluppo!
- Nel caso pratico di Linux, quale tool viene utilizzato per il Source Control Management (SCM)?
  - Prima nulla
  - Poi Bitkeeper ©
  - E domani?

Source Control Management (SCM)

- Quali sono le principali alternative Open Source disponibili?

  - CVS — http://www.cshome.org
  - Subversion — http://subversion.tigris.org
  - GNU Arch — http://www.gnu.org/software/gnu-arch/
    - Bazaar — http://bazaar.canonical.com
    - Bazaar-NG (bzr) — http://www.bazaar-ng.org
    - Monotone — http://www.venge.net/monotone/

Requisiti: le regole del gioco

Non esiste la "silver bullet", definiamo con attenzione i requisiti richiesti e lo scenario di utilizzo:

- Numero di sviluppatori coinvolti (scalabilità del sistema)
- Dinamicità degli sviluppatori e modello di comportamento
- Livello di coordinamento tra sviluppatori
- Centralizzazione / decentralizzazione
- Fault tolerance
- Problemi di sicurezza / confidenzialità del codice e dei bug
- Modifiche contemporanee agli stessi componenti, parallelismo nello sviluppo

SCM: modelli architetturali

- Approccio **centralizzato**

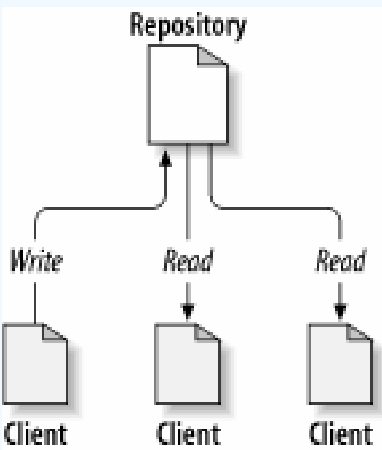
  La versione "ufficiale" del repository viene mantenuta da un server principale. I client si collegano al server per ottenere gli aggiornamenti e per sottomettere le modifiche

- Approccio **distribuito**

  Ogni client mantiene una propria versione del repository. Quando opportuno versioni diverse vengono sincronizzate e le modifiche propagate. Lo sviluppo può procedere con modalità peer-to-peer (P2P)

Approccio centralizzato

- Centralizzato: architettura client/server



I vari client richiedono la versione attuale del codice ad un server centrale e sottomettono le modifiche

Possono sorgere vari problemi:

- come gestire la concorrenza?
- come mantenere coerenti le varie versioni senza sovrascrivere alcune modifiche?

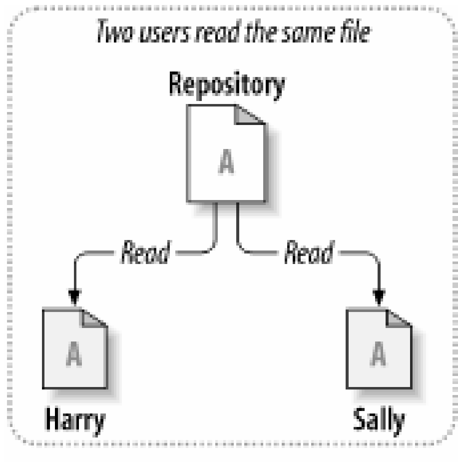
È sufficiente sincronizzare?

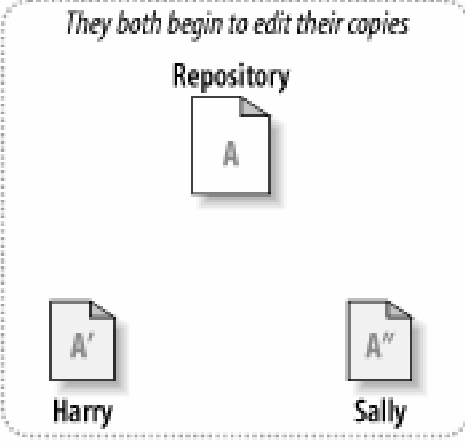
Sovrascrittura accidentale

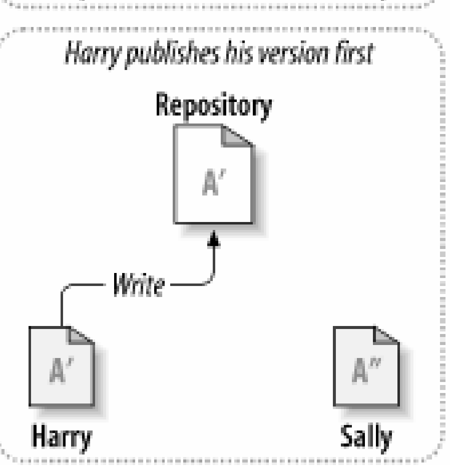
Analizziamo un problema classico: una versione valida del codice viene sovrascritta da un'altra versione. Il primo insieme di modifiche viene perso in modo definitivo

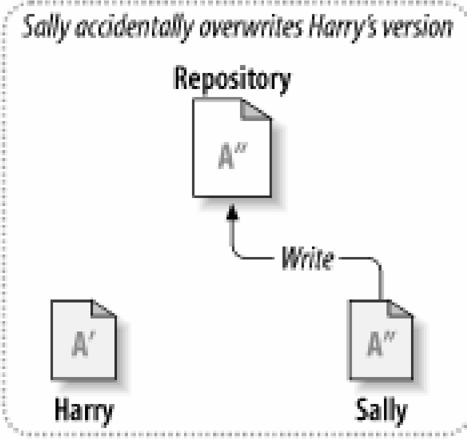
Come è possibile risolvere questo problema?

Esiste una sola soluzione?







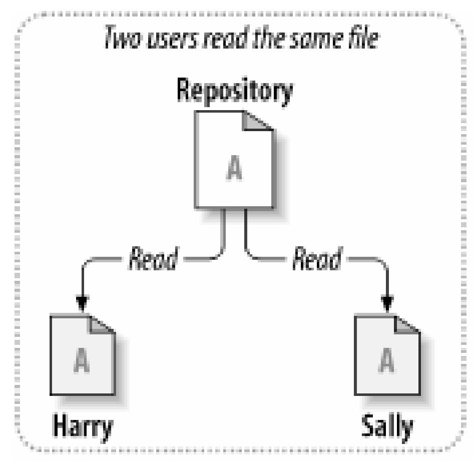


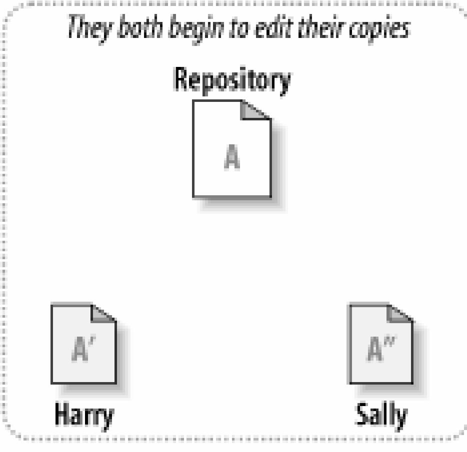
Sovrascrittura accidentale

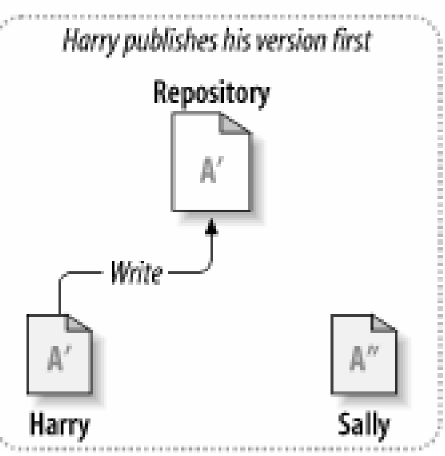
Due approcci molto diversi tra di loro:

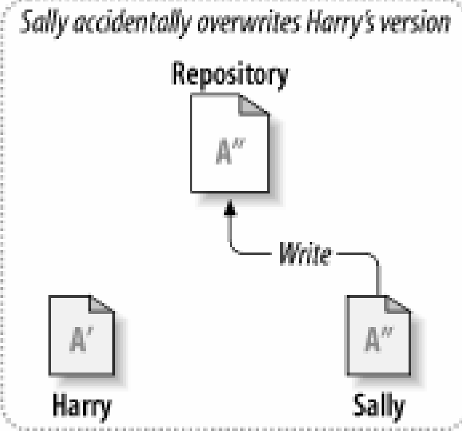
- **Approccio pessimistico**
- **Approccio ottimistico**

Nel primo caso facciamo in modo che il problema sia evitato a priori, nel secondo viene ricostruito uno stato coerente solo se necessario



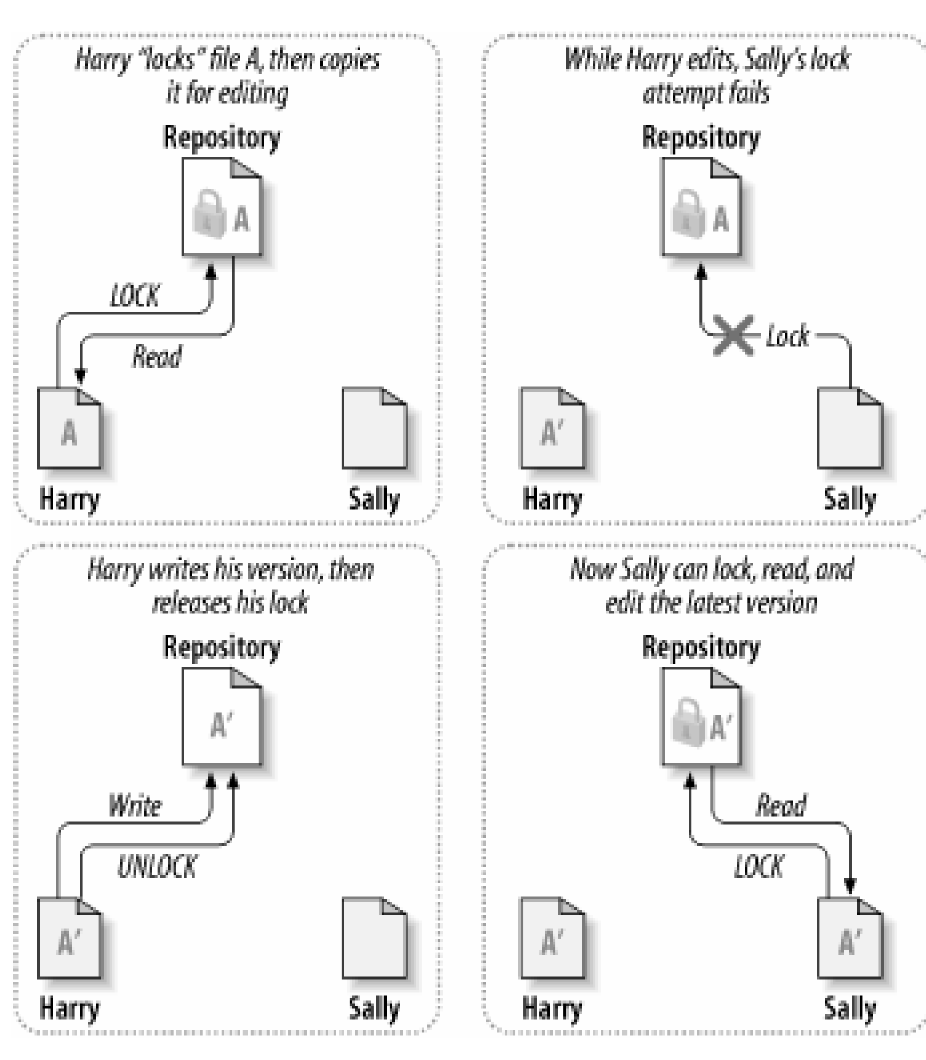






Soluzione pessimistica: lock based

Il problema viene evitato imponendo una **serializzazione** su tutte le operazioni di modifica. Prima di procedere alla modifica l'utente deve richiedere un lock su una parte del repository, effettuare le modifiche e propagarle alla copia principale. Solo a questo punto il lock può essere rilasciato



Vantaggi e svantaggi della soluzione pessimistica

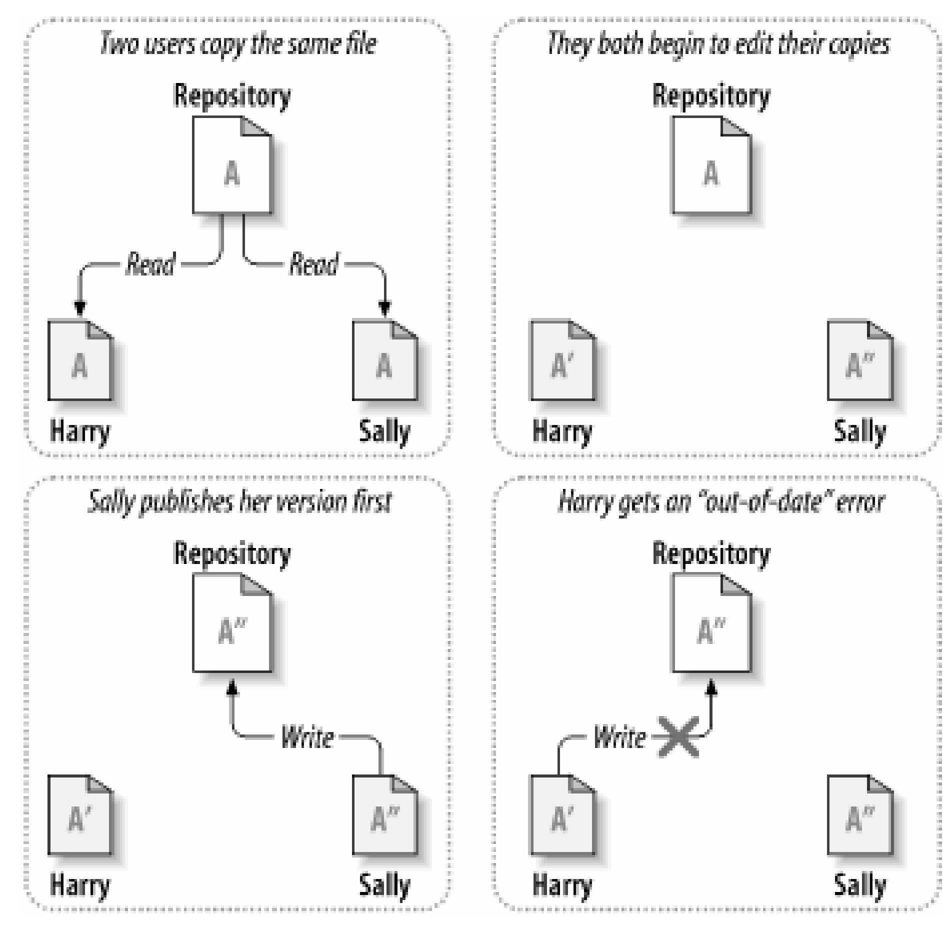
Apparentemente si tratta di una soluzione accettabile e comoda, visto che previene la formazione di conflitti, ma:

- cosa accade se un lock non viene rilasciato?
- la serializzazione rischia di limitare in modo significativo la concorrenza del lavoro su alcune parti del codice
- è una soluzione poco scalabile
- richiede un ottimo livello di comunicazione tra gli sviluppatori (in alcuni casi questo è del tutto impossibile o estremamente scomodo/costoso)

Soluzione ottimistica: copy-modify-merge

I client sono liberi di modificare a piacimento la propia copia locale. Al momento di pubblicare la versione modificata il client verifica la situazione sul server. Se non ci sono state altre modifiche al codice interessato può semplicemente pubblicare la sua versione.

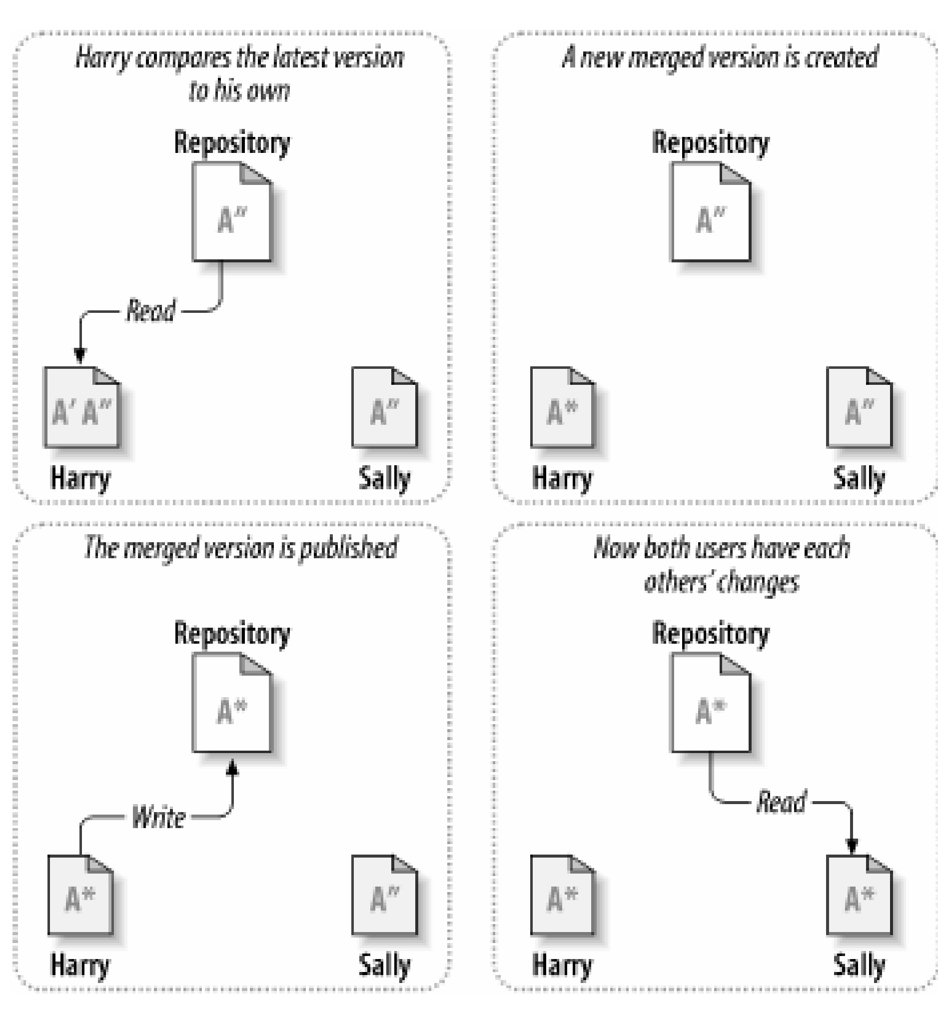
Altrimenti …



Soluzione ottimistica: copy-modify-merge

Altrimenti …

prima di pubblicare le proprie modifiche è costretto a sincronizzarsi con la versione attualmente disponibile sul server, risolvendo eventuali conflitti. Solo dopo questa operazione potrà aggiungere le proprie modifiche che saranno poi eventualmente propagate anche agli altri client



Vantaggi e svantaggi della soluzione ottimistica

- Apparentemente caotica, ma nella pratica funziona molto bene
- Il livello di concorrenza nello sviluppo può essere massimizzato
- Costringe alla risoluzione di conflitti, non sempre risolvibili in modo automatico, a volte è necessario l'intervento umano
- Alcuni conflitti possono essere tutt'altro che banali da risolvere
- Se un client rimane molto tempo senza aggiornarsi durante la sua fase di modifica del sorgente rischia di incorrere in conflitti molto ampi e complicati da gestire

Introduzione a CVS

CVS nasce nel 1989 come collezione di script per facilitare l'uso di RCS. L'approccio che implementa è centralizzato e basato sul paradigma copy-modify-merge

Supporta varie forme di accesso:

- su disco locale
- da remoto (via pserver)
- da remoto + ssh

L'autenticazione degli utenti è solitamente basata su username + password, è necessario prestare attenzione a come viene installato il server (accesso al filesystem)

Supporta tagging e branches

CVS

Breve tutorial di utilizzo:

- export CVSROOT=/var/lib/cvs *(bash + debian)*
- cvsinit *(nel caso debian viene già fatto installando il pacchetto)*
- ls -la /var/lib/cvs (notiamo groupid, modifichiamo /etc/group)
- cd /home/gda/src/project

- cvs import -m "Sample Program" projectname vendortag releasetag

  N project/Makefile

  N project/main.c

  N project/bar.c

  N project/foo.c

CVS

- cvs checkout projectname

  cvs checkout: Updating project

  U project/Makefile

  U project/bar.c

  U project/foo.c

  U project/main.c

- cvs commit -m "Modifica…" main.c
- cvs update
- cvs status
- cvs tag releasename
- cvs checkout -r releasename projectname

- … creare nuovi file, nuove directory, cancellare file…

Subversion

Lo sviluppo di subversion è iniziato nel Febbraio 2000 con il preciso obiettivo di costruire un sostituto di CVS che mantenesse intatto il modello di sviluppo, risolvendo i problemi principali ma senza dover mantenere compatibilità con il passato

Caratteristiche principali:

- Versionamento delle directory (è possibile cancellarle)
- Diversa gestione delle versioni
- Maggiore presenza di network layers utilizzabili (es. ssh, http, https)
- Interoperabilità con Apache 2
- Branching e tagging implementati in modo efficiente
- Presenza di un database per lo storage (usualmente Berkeley DB)

SCM con Eclipse

Resource - Makefile - Eclipse Platform

File Edit Navigate Search Project Run Window Help

Resource

Navigator

- >artis [caronte.pads.cs.unibo.it]
  - CMB
    - cmb.c 1.1.1.1 (ASCII -kkv)
    - cmb.h 1.1.1.1 (ASCII -kkv)
    - Makefile 1.1.1.1 (ASCII -kkv)
  - EXAMPLES
  - MODELS
  - MP
  - TEST
  - TS
  - UTILS
  - >.project (ASCII -kkv)
  - Makefile 1.1.1.1 (ASCII -kkv)

cmb.c | Makefile

```
INCLDIR      = $(ROOT)/INCLUDE
LIBDIR       = $(ROOT)/LIB
OBJDIR       = $(ROOT)/OBJ

LIBCMB       = $(LIBDIR)/libcmb.a
LIBOBJS      = cmb.o
EXTOBJ       = $(OBJDIR)/heap.o
EXPORT_HEADERS  = cmb.h


#-------------------------------------------------------------------------------
CC       = gcc
DEFS            = -DDROPIN -DDROPOUT
CFLAGS          = -g -Wall -O -I. -I$(INCLDIR) -L$(LIBDIR)/librticomm.a  $(DEFS)


#-------------------------------------------------------------------------------
all:     $(LIBCMB)


#-------------------------------------------------------------------------------
$(LIBCMB): $(EXTOBJ) $(LIBOBJS)
	$(AR) crv $(LIBCMB) $(LIBOBJS) $(EXTOBJ)
	@sh -c 'cd ${INCLDIR}; for i in ${EXPORT_HEADERS}; do \
		if [ ! -f $$i ]; then \
			ln -s $(ROOT)/CMB/$$i $$i; \
			echo Installed $$i in include directory; \
		fi \
		done'
	@sh -c 'cd ${OBJDIR}; for i in ${LIBOBJS}; do \
			if [ ! -f $$i ]; then \
				ln -s $(ROOT)/CMB/$$i $$i; \
				echo Installed $$i in object directory; \
			fi \
		done'


#-------------------------------------------------------------------------------
clean :
	@sh -c ' \
			echo cd ${OBJDIR}; \
			cd ${OBJDIR}; \
			for i in ${LIBOBJS}; do \
				if [ -f $$i ]; then \
					echo rm -f $$i; \
```

Writable | Insert | 1 : 1

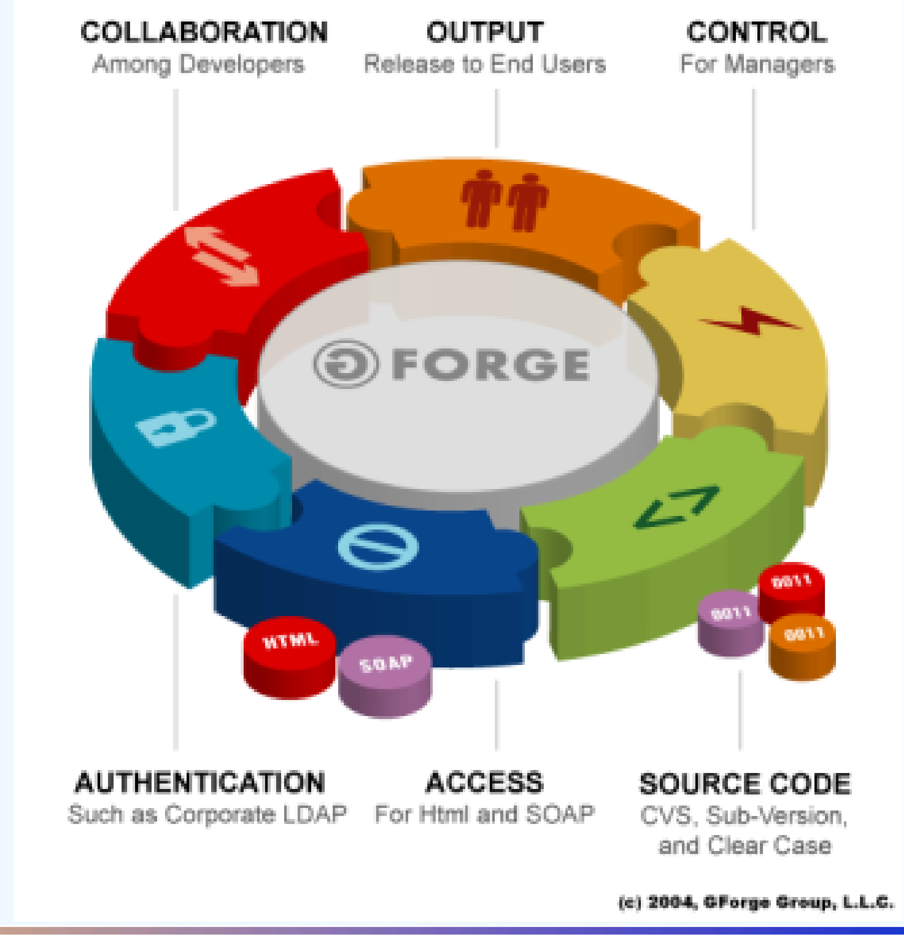
SCM: consigli utili

Alcuni consigli (a volte banali) possono risultare molto utili:

- I commenti da associare ai check-in sono estremamente importanti: è un pessimo settore dove risparmiare tempo
- Sarebbe opportuno avere check-in diversi per ogni "**gruppo logico**" di modifiche, piuttosto che relativo a "**sessioni di lavoro**"
- Durante lo sviluppo conviene cercare di mantenersi sincronizzati con la versione "ufficiale", se rimaniamo sconnessi per molto aumentano le probabilità di un check-in difficoltoso (la versione ufficiale **diverge** molto)
- Non è necessario avere un repository unico per tutto il lavoro
- Gli SCM sono principalmente pensati per codice e configurazioni, non per tutto quello che potremmo voler inserire

GForge

Un buon SCM a volte non è sufficiente, è necessaria un'infrastruttura molto più articolata e completa, un Collaborative Development Environment (CDE)



Esempi:

- SourceForge

  www.sourceforge.net

- GForge

  www.gforge.org

Sistemi di bug tracking

Se si scrive codice, si creano bug

I bug non andrebbero nascosti / ignorati ma:

- segnalati / identificati ed assegnati in modo corretto
- riprodotti / valutati per l'impatto sulle funzionalità
- valutati rispetto all'impatto sulla **sicurezza**
- problema della diffusione delle "**informazioni a rischio**"
- ... eventualmente risolti

Bug triage: i bug tendono ad accumularsi, spesso sono segnalazioni sbagliate / obsolete / incomplete. La valutazione è estremamente importante e non deve essere necessariamente fatta da uno sviluppatore

Sistemi di bug tracking: esempi pratici

- Debian
  - http://packages.debian.org/unstable/gnome/evolution/
  - http://bugs.debian.org/cgi-bin/pkgreport.cgi?pkg=evolution
  - http://www.debian.org/Bugs/
  - http://www.debian.org/Bugs/Reporting/
  - Uso di reportbug [-d] (notare che tutto il sistema è basato sul mail)
  - Ovviamente Open Source, pacchetto Debian debbugs
- BugZilla – Defect Tracking System – Bug Tracking System
  - Sviluppato in Perl + database, nasce come progetto Mozilla
  - www.bugzilla.org
  - Esempio pratico: http://www.mozilla.org/bugs/

Building / Testing / Profiling

- In un modello di sviluppo del codice “caotico” come vengono compilate / distribuite le nuove versioni binarie?
- Importanza di “suite” di test automatiche da affiancare alla verifica umana
- Profiling: verifica delle prestazioni del sistema nella sue singole parti, analisi dei colli di bottiglia ecc. Durante lo sviluppo accade che le prestazioni di alcuni moduli siano inaccettabili, è bene rilevarlo immediatamente, preferibilmente in modo automatico

Note e bibliografia

- Molte delle figure utilizzate sono tratte da: “Version Control with Subversion, For Subversion 1.0” di B. Collins-Sussman, B.W. Fitzpatrick, C.M. Pilato. http://svnbook.red-bean.com/en/1.0/svn-book.html
- Source Control HOWTO, E. Sink.

  http://software.ericsink.com/scm/source_control.html
- Open Source Development with CVS. M. Bar, K. Fogel. Paraglyph Press
- CVS Cederqvist Manual. https://www.cvshome.org/docs/manual/