

Progetto esame di Reti di elaboratori

Mattioli Gioele

matricola: 254833

Sessione autunnale 2015/16

Sistema di controllo del codice sorgente

(pyVersioning)

Docente: prof. A. Della Selva

Indice

[Sistema di controllo del codice sorgente](#)

[Indice](#)

[Scopo del progetto](#)

[Sviluppo](#)

[Linguaggio](#)

[Compatibilità](#)

[Dipendenze](#)

[Strumenti di sviluppo](#)

[Software aggiuntivi](#)

[Concetti base e operazioni](#)

[Concetti](#)

[Operazioni lato server](#)

[Operazioni lato client](#)

[Interfaccia fra client e server](#)

[Ulteriori operazioni](#)

[Comunicazione](#)

[RPC](#)

[Concetti](#)

[Implementazione](#)

[Realizzazione lato server](#)

[Realizzazione lato Client](#)

[Metodi esposti dal Server](#)

[Trasferimento file](#)

[Realizzazione lato server](#)

[Realizzazione lato Client](#)

[Sviluppo](#)

[Server](#)

[Client](#)

[Commit e versioni](#)

[Pending Changes](#)

[Test](#)

[Conclusioni](#)

[Tecniche utilizzate](#)

[Sviluppi futuri](#)

[Stime e sforzo dello sviluppo](#)

[Appendice 1](#)

[Codice](#)

[Client.py](#)

[Server.py](#)

[Repository.py](#)

[Branch.py](#)

[Changeset.py](#)

[uti.py](#)

[consts.py](#)

Scopo del progetto

Il progetto si ispira agli ormai numerosi e diffusi sistemi di controllo del codice sorgente. Tali sistemi permettono a gruppi di sviluppatori o singoli individui di sviluppare prodotti software e mantenere una copia costantemente sincronizzata e comune oltre a permettere la gestione delle versioni.

I concetti basilari e le nomenclature cui si farà riferimento si ispirano a sistemi quali GitHub, Microsoft-TFS e VS-TeamService in quanto familiari al sottoscritto. Lo sviluppo e le logiche del progetto sono di libera invenzione e adattate al caso specifico.

Lo scopo è quello di sviluppare un ridotto sistema di controllo sorgente, in cui più utilizzatori (la cui porzione di software va intesa come client) comunichino con un sistema comune ospitato su una terza entità (da intendere come server) comune.

Sviluppo

Linguaggio

Il progetto è stato realizzato interamente nel linguaggio Python, in particolare nella versione 3.4.4 del linguaggio (<https://www.python.org/>).

Compatibilità

Il codice è compatibile con le versioni di Python 3.4.x, 3.5.x e successive.

Il progetto è compatibile con il sistema operativo Microsoft Windows (7, 8, 8.1, 10)

Dipendenze

Sono state utilizzate le seguenti librerie Python non standard:

- [rpyc](#)
- [natsort](#)

Strumenti di sviluppo

Si è fatto uso dei seguenti strumenti di sviluppo:

- Microsoft Visual Studio 2015 Community (<https://www.visualstudio.com/it-it/products/visual-studio-community-vs.aspx>)
- Microsoft Visual Studio Team Services (<https://www.visualstudio.com/it-it/products/visual-studio-team-services-vs.aspx>)
- Microsoft Visual Studio Code (<https://code.visualstudio.com/>)
- Python Tools for Visual Studio (<https://microsoft.github.io/PTVS/>)

- JetBrains PyCharm (<https://www.jetbrains.com/pycharm/>)
- Notepad++ (<https://notepad-plus-plus.org/>)
- Python IDLE
- Python pip

Software aggiuntivi

Il software sviluppato si integra con i seguenti software aggiuntivi:

- WinMerge (<http://winmerge.org/>)

Concetti base e operazioni

Concetti

- *Repository*: entità associata ad un componente software
- *Branch*: i branch costituiscono più rami di un repository
- *Trunk*: per “trunk” si intende il ramo principale di ogni repository (in altre nomenclature “Master”), ogni repository deve avere un ramo trunk mentre può avere un numero arbitrario di branch (ogni nuovo branch viene originato dal trunk).
- *Changeset*: costituisce l’insieme di modifiche associate ad un’operazione di commit (spiegato meglio in seguito). Per come è stato strutturato il progetto, ogni changeset è costituito da un’insieme di files che hanno subito modifiche. Solo i file modificati vengono inseriti in un changeset, in questo modo si riduce notevolmente l’occupazione di memoria e la ridondanza.
- *Backup*: per “backup” si intende un’operazione richiesta in questo progetto per poter creare un particolare changeset in cui venga memorizzata una copia completa del progetto. Il changeset di backup viene ottenuto copiando tutte le modifiche contenute nei changeset successivi all’ultimo changeset di backup. Questa operazione permette di disporre di un’ultima versione completa del codice senza dover mantenere copie ridondanti in tutti i changeset.

Operazioni lato server

Le operazioni che il server deve svolgere sono le seguenti:

- Creazione e gestione di repositories
- Creazione e gestione di branches
- Creazione e gestione di changesets
 - Creazione di changeset di backup
 - Ripristino di una vecchia versione (di seguito "*Rollback*")

Tutte le operazioni del server devono essere eseguite a seguito di comandi ricevuti dal client. Il server non dispone di operazioni autonome lato-utente e di interazione con lo stesso.

Operazioni lato client

Fra le operazioni tipiche di cui il client deve poter disporre ci sono:

- “Mappare” in locale una versione del software presente sul server
 - Mapping del repository
 - Mapping del branch
- Acquisizione di una determinata versione
 - Copiare l’ultima versione del software (di seguito “*GetLatestVersion*”)
 - Copiare una specifica versione del software (di seguito “*GetSpecificVersion*”)
 - Operazioni di confronto fra file di diversi changeset (di seguito “*Compare*”)
- Creare un repository per un nuovo componente software
- Creare un branch da un determinato repository
- Effettuare modifiche sulla versione locale (di seguito “*check-out*”)
- Inviare le modifiche e quindi aggiornare la versione del server (di seguito “*commit*” o “*check-in*”)
 - Verificare differenze fra le versioni di uno o più file
 - Invio di uno o più file
 - Disporre di mantenere la versione del server
 - Disporre di sovrascrivere la versione del server con quella locale
- Visualizzare una lista dei file modificati localmente (di seguito “*Pending changes*”)
 - lista di tutte le modifiche catalogate per tipo di modifica
 - Nuovo file
 - Vecchio file
 - File aggiunto
 - File rimosso
 - Escludere/includere file dai pending (singoli file o tutti i file di un determinato tipo)
- Annullamento delle modifiche fatte in locale su uno o più file (di seguito “*Undo*”)
- Rimuovere il mapping di repository o branch (di seguito “*Drop*”).

Interfaccia fra client e server

Il server deve inoltre esporre un'interfaccia al client che fornisca la possibilità di svolgere operazioni quali:

- Listato dei repository presenti
- Listato dei branch di un determinato repository (con il numero di changeset da cui sono stati originati)
- Listato dei changesets di un determinato branch
 - lista di tutte le modifiche catalogate per tipo di modifica
 - Nuovo file
 - Vecchio file
 - File aggiunto
 - File rimosso
 - data della modifica e della versione precedente
- Creazione di nuovi repository
- Creazione di nuovi branch
- Rimozione di repository
- Rimozione di branch
- GetLatestVersion / GetSpecificVersion
- Commit
- Undo

Ulteriori operazioni

Altre operazioni a disposizione dell'utente sono:

- Guida per programma
- Visualizzazione del percorso corrente
- Listato dei file del branch corrente (sul server)
- Apertura rapida della cartella corrente (in locale)
- Uso di un software di comparazione avanzato e munito di GUI (in questo caso è stato integrata una versione portable del software free [WinMerge](#))

Comunicazione

RPC

Concetti

La comunicazione fra client e server è stata implementata tramite RPC (Remote Procedure Call).

Il server espone al client una serie di metodi con i quali il client può interagire in modo controllato con il server stesso. A seguito di una delle chiamate di cui sopra il server esegue una serie di operazioni nascoste al client, il quale a sua volta fa uso dei metodi esposti dal server in maniera trasparente, astruendo da come sono implementati.

Implementazione

Per fare uso di chiamate rpc si sfrutta la libreria [rpyc](#) nella versione 3.3.0 per Python 3.3 o successivi.

In particolare si fa riferimento modalità [NewStyle](#), che rispetto alla modalità *Classic* permette operazioni più avanzate e svincola lo sviluppatore da vincoli stilistici. Inoltre, questa modalità, orientata ai servizi, permette di implementare la comunicazione fra client e server con un servizio attivo sul server, il quale espone un determinato insieme di funzionalità al client e non vincola il server alla condizione di "slave".

Per garantire maggiore separazione fra metodi di "interfaccia" del server e metodi di "logica" sottostante, i metodi esposti dal server incapsulano chiamate più complesse appartenenti alla logica del server stesso e articolate in più classi.

Anche se possibile, non si è resa necessaria una comunicazione bi-direzionale. Il server svolge sempre un ruolo passivo, limitandosi ad assolvere alle richieste del client. Tuttavia, non è da precludersi un ruolo più attivo del server, specie nel caso in cui dovesse assolvere richieste da parte di un numero arbitrario di client, o se si rendesse necessario aumentarne le funzionalità.

Realizzazione lato server

```
if __name__ == "__main__":  
    print("Benvenuto su pyVersioning (Server)")  
  
    #creo la cartella di installazione se non presente
```

```

root = "C:\pyV\pyVServer"
if (path.isdir(root) == False):
    os.makedirs(root)

#avvio il servizio rpyc
print("Avvio servizio in corso...")
server = ThreadedServer(Server, port = 18812)
print("Server attivo.")
server.start()

```

Realizzazione lato Client

```

try:
    #connetto client e server
    print("Benvenuto in pyVersioning")

    #prendo l'indirizzo del server
    while (True):
        print("Digitare l'ip del server (\"localhost\" per un server
locale):")
        host = input()
        if (host == "localhost"):
            break

        try:
            socket.inet_aton(host)
            break
        except socket.error as er:
            print("IP non valido", end="\n\n")

    #stabilisco la connessione
    print("Connessione al server...", sep="\n")
    connection = rpyc.connect(host, 18812)
    print("Connessione stabilita.", end="\n\n")

    #lancio il client
    Client(connection).runMenu()

    #chiudo la connessione
    connection.close()

except Exception as ex:

```

```
        print("Si è verificato un errore: {}".format(ex), "Il programma  
verrà terminato.", sep="\n", end="\n\n")
```

Metodi esposti dal Server

```
def on_connect(self):  
    print("\nClient connesso.")
```

```
def on_disconnect(self):  
    print("\nClient disconnesso.")
```

```
### metodi di interfaccia Client-Server rpyc ###
```

```
class exposed_File():
```

```
    def exposed_open(self, filePath, mode="r"):  
        self.filePath = filePath  
        self.file = open(filePath, mode, errors="ignore")
```

```
    def exposed_write(self, bytes):  
        return self.file.write(bytes)
```

```
    def exposed_read(self, bytes):  
        return self.file.read()
```

```
    def exposed_close(self):  
        return self.file.close()
```

```
    def exposed_getmtime(self):  
        return path.getmtime(self.filePath)
```

```
def exposed_findFile(self, repoName, branchName, fileRelPath,  
startChangeset=None):  
    return self.findFile(repoName, branchName, fileRelPath,  
startChangeset)
```

```
def exposed_existsRepo(self, repoName):  
    return self.existsRepo(repoName)
```

```

def exposed_existsBranch(self, repoName, branchName):
    return self.getRepo(repoName).existsBranch(branchName)

def exposed_getRepo(self, repoName):
    return self.getRepo(repoName)

def exposed_addRepo(self, repoName):
    """crea un nuovo repository, il trunk e il changeset 0, ritorna la
    directory del changeset 0"""

    self.addRepo(repoName)
    return
self.getRepo(repoName).getBranch(TRUNK).getChangeset(0).changesetDir

def exposed_removeRepo(self, repoName):
    self.removeRepo(repoName)

def exposed_addBranch(self, repoName, branchName):
    """crea un nuovo branch"""

    self.getRepo(repoName).addBranch(branchName)

def exposed_removeBranch(self, repoName, branchName):
    """rimuove il branch"""

    self.getRepo(repoName).removeBranch(branchName)

def exposed_addChangeset(self, repoName, branchName, comment):
    """aggiunge un changeset al branch "branchName" """

    return
self.getRepo(repoName).getBranch(branchName).addChangeset(comment).changesetDir

def exposed_existsChangeset(self, repoName, branchName, changesetNum):
    """ritorna True se esiste il changeset "changesetNum" nel branch
    "branchName" """

    try:

```

```

self.getRepo(repoName).getBranch(branchName).getChangeset(changesetNum)
    return True
except:
    return False

def exposed_listDir(self, dir):
    return uti.listDir(dir)

def exposed_listBranch(self, repoName, branchName):
    """ritorna tutti i file e sottocartelle del branch selezionato"""

    #creo una versione completa in una cartella temporanea
    tmpDir =
self.getRepo(repoName).getBranch(branchName).getLatestVersion()

    #memorizzo una lista dei file nella versione
    list = self.exposed_listDir(tmpDir)

    #rimuovo la cartella temporanea
    shutil.rmtree(tmpDir)

    #formatto i path dei file per la stampa
    return [elem.replace(tmpDir,"") for elem in list]

def exposed_showRepos(self):
    """ritorna la lista di repositories sul server"""

    return self.getRepoList()

def exposed_showBranches(self, repoName):
    """ritorna la lista di branch nel repository "repoName" """

    return self.getRepo(repoName).getBranchList()

def exposed_showChangesets(self, repoName, branchName):
    """ritorna la lista di changeset nel branch "branchName" con i file
    modificati"""

    return
self.getRepo(repoName).getBranch(branchName).getChangesetList()

```

```

def exposed_getLatestVersion(self, repoName, branchName):
    """scarica l'ultima versione del branch "branchName" in una cartella
    temporanea """
    branch = self.getRepo(repoName).getBranch(branchName)

    return branch.getLatestVersion(), branch.getLastChangesetNum()

def exposed_getSpecificVersion(self, repoName, branchName,
changesetNum):
    """scarica la versiona aggiornata al changeset "changesetNum" del
    branch "branchName" in una cartella temporanea """

    branch = self.getRepo(repoName).getBranch(branchName)
    return
self.getRepo(repoName).getBranch(branchName).getSpecificVersion(changesetNum)

def exposed_getLastChangeset(self, repoName, branchName):
    """ritorna l'ultimo changeset del branch"""
    return
self.getRepo(repoName).getBranch(branchName).getLastChangesetNum()

```

Trasferimento file

Il trasferimento file è una parte consistente del progetto che basa la sua utilità sul trasferimento di dati da e verso un server.

Anche in questo caso ci si appoggia sulla libreria [rpyc](#).

L'idea alla base è quella di effettuare di fatto un override di metodi solitamente usati per scrivere e leggere file in locale con delle rpc. La differenza risulta di impatto minimo lato client, dove l'uso delle chiamate è molto simile all'uso ordinario.

Lato server, l'idea è quella di incapsulare chiamate per lettura/scrittura di file locali in metodi resi disponibili all'esterno.

Un incapsulamento su livelli ulteriori permette il trasferimento non solo di file ma di intere cartelle e sottocartelle.

L'uso di percorsi relativi permette di manipolare facilmente i file presenti sulle macchine ai due capi del software.

Realizzazione lato server

```
class exposed_File():

    def exposed_open(self, filePath, mode="r"):
        self.filePath = filePath
        self.file = open(filePath, mode, errors="ignore")

    def exposed_write(self, bytes):
        return self.file.write(bytes)

    def exposed_read(self, bytes):
        return self.file.read()

    def exposed_close(self):
        return self.file.close()

    def exposed_getmtime(self):
        return path.getmtime(self.filePath)

def exposed_findFile(self, repoName, branchName, fileRelPath,
startChangeset=None):
    return self.findFile(repoName, branchName, fileRelPath,
startChangeset)
```


Realizzazione lato Client

```
def copyDirToClient(self, dirFrom, dirTo):
    """copia la cartella "dirFrom" nella cartella "dirTo" """

    #sovrascrivo la cartella
    if (path.isdir(dirTo)):
        shutil.rmtree(dirTo)
    os.makedirs(dirTo)

    #copio tutti i file contenuti
    for file in self.server.listdir(dirFrom):
        self.copyFileToClient(file, file.replace(dirFrom, dirTo))

def copyFileToClient(self, fileFrom, fileTo):
    """copia il file "fileFrom" del server nel path "fileTo" """

    remoteFile = self.server.File()
    remoteFile.open(fileFrom)

    #copio il file del server nella cartella temporanea
    localFile = open(fileTo, "w")

    shutil.copyfileobj(remoteFile, localFile)

    remoteFile.close()
    localFile.close()

    #copio la data di ultima modifica dal file del server
    editTime = remoteFile.getmtime()
    os.utime(fileTo, (int(editTime), int(editTime)))

def copyDirToServer(self, dirFrom, dirTo):
    """copia la cartella "dirFrom" nella cartella "dirTo" """

    #controllo che la cartella dirFrom esista
    if (path.isdir(dirFrom) == False):
        raise Exception

    #copio tutti i file contenuti
```

```
for file in uti.listdir(dirFrom):
    self.copyFileToServer(file, file.replace(dirFrom, dirTo))

def copyFileToServer(self, fileFrom, fileTo):
    """copia il file "fileFrom" del client nel path "fileTo" sul
    server"""

    localFile = open(fileFrom, errors="ignore")

    remoteFile = self.server.File()
    remoteFile.open(fileTo, "w")

    shutil.copyfileobj(localFile, remoteFile)

    remoteFile.close()
    localFile.close()
```

Sviluppo

Si riportano di seguito una serie di informazioni relative allo sviluppo e alla vera e propria realizzazione del progetto. Si astrae il più possibile dall'implementazione in codice, per la quale si rimanda all'[appendice 1](#).

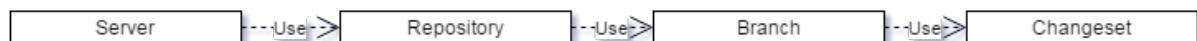
Server

Il server è deputato alla realizzazione vera e propria delle operazioni analizzate sopra e costituisce la logica operativa alla base del progetto.

I compiti del server possono essere divisi in due categorie:

- **Interfaccia con il client:** una serie di metodi esposti dal server al client, invocabili da quest'ultimo tramite rpc fornendo al client un'interfaccia ad operazioni più complesse svolte e implementate dal server stesso. Per maggiori informazioni si rimanda alla sezione "[metodi esposti dal server](#)" e "[trasferimento file](#)".
- **Logica operativa:** una serie di classi, attributi e metodi che svolgono tutte le operazioni fondamentali per la gestione di repositories, branches, changesets e files, oltre ad assolvere alle richieste del client.

La logica operativa è realizzata tramite una struttura di classi gerarchicamente collegate.



La classe principale del server ha il compito di gestire i repositories:

- creazione di repository
- rimozione di repository
- ricerca di repository
- listato di repositories
- verifica dell'esistenza di repository

È compito di questa classe gestire l'interazione con il client, esporre metodi allo stesso e gestire la loro implementazione.

La classe Repository ha il compito di gestire i branches:

- creazione di branch (va differenziata la creazione del branch principale - "trunk" - e degli altri branches)
- rimozione di branch
- ricerca di branch
- listato di branches
- verifica dell'esistenza di branch
- fornire il file di info del repository

La classe Branch ha il compito di gestire i changesets:

- creazione di changesets (va differenziata la creazione del changeset iniziale, changesets di backup, changesets normali)
- ricerca di changeset
- gestione del numero dei changesets
- listato di changeset (con file modificati e tipo di modifica)
- realizzazione ultima versione
- realizzazione versione specifica
- fornire il file di info del branch
- compilare il file di info dei changesets

La classe Changeset fornisce oggetti con cui interagire per gestire i changesets, essi a loro volta sono usati per ottenere informazioni sul changeset e sulle modifiche contenute in esso.

Per il codice sviluppato si rimanda all'[appendice 1](#).

Client

Il client è progettato per essere prevalentemente interattivo con l'utente. Si presenta infatti come un'applicazione a riga di comando e dispone di un determinato insieme di istruzioni. Costituendo di fatto la parte che dialoga con l'utente, la logica operativa sottostante è piuttosto limitata. È compito del client:

- gestire i comandi impartiti dall'utente,
- valutare se i comandi sono compatibili con la situazione corrente,
- interrogare il server,
- gestire la risposta del server ed eventuali eccezioni,
- comunicare i risultati all'utente.

Proprio per questo motivo, il client astrae completamente dalla logica del server e dalle classi con cui esso è implementato. Esso si limita a invocare rpc sul server che poi gestisce queste chiamate per effettuare le operazioni richieste.

Fanno eccezione:

- la gestione del trasferimento file, la cui logica è divisa fra client e server, il server espone metodi utili alla copia di file ma è lato client che vengono impartite le istruzioni e invocate le necessarie rpc, per maggiori informazioni si fa riferimento al capitolo ["trasferimento file"](#).
- la gestione dei file in modifica, il client interroga in un primo momento il server, poi effettua confronti con i file locali e identifica quali file hanno subito modifiche.

Per i comandi disponibili si rimanda alla guida interna al programma stesso, invocabile con il comando:

```
> help
```

Si riportano comunque i comandi e una breve descrizione

- > exit - chiude il programma (scrive su file il repository e branch corrente per essere letti e impostati al prossimo avvio)
- > repolist - stampa la lista dei repositories presenti sul server
- > branchlist - stampa una lista dei branches presenti sul repository corrente sul server
- > createrepo [repoName] - crea il repository "repoName" nel server
- > createbranch [branchName] - crea il branch "branchName" nel server
- > removerepo [repoName] - rimuove il repository "repoName" dal server
- > removebranch [branchName] - rimuove il branch "branchName" dal server
- > maprepo [repoName] - mappa il repository "repoName" nella macchina locale
- > mapbranch [branchName] - mappa il branch "branchName" nella macchina locale
- > droprepo [repoName] - elimina il repository "repoName" dalla macchina locale
- > dropbranch [branchName] - elimina il branch "branchName" dalla macchina locale

- > `setrepo [repoName]` - imposta il repository "repoName" come repository corrente
- > `setbranch [branchName]` - imposta il branch "branchName" come branch corrente
- > `currdir` - stampa il percorso di esecuzione corrente (fa riferimento al repository e branch correnti)
- > `listdir` - stampa una lista di file e sottocartelle per il progetto selezionato lette dall'ultima versione del server
- > `history` - stampa la lista dei changeset del branch corrente letti dal server, include una lista di file modificati (indicando di che tipo di modifica si tratta)
- > `getlatest` - scarica la versione più recente del branch corrente
- > `getspecific [changeset]` - scarica la versione specificata in "changeset" del branch corrente
- > `pending` - stampa una lista dei file modificati in locale (indicando di che tipo di modifica si tratta)
- > `excludeextension [ext]` - esclude tutti i file .[ext] dai file in modifica
- > `excludefile [file]` - esclude il file "file" dai file in modifica
- > `includeextension [ext]` - include l'estensione se precedentemente esclusa
- > `includeallextensions` - include tutte le estensioni precedentemente escluse
- > `includefile [file]` - include il file se precedentemente escluso
- > `includeallfiles` - include tutti i file precedentemente esclusi
- > `includeall` - include tutti i file e estensioni precedentemente esclusi
- > `printexcluded` - stampa la lista di estensioni e file esclusi
- > `commit [file]` - effettua il commit del file "file"
- > `commitall` - effettua il commit di tutti i file in pending
- > `undo [file]` - annulla le modifiche sul file "file"
- > `undoall` - annulla le modifiche su tutti i file in pending e scarica l'ultima versione
- > `compare [file]` - effettua un confronto fra il file "file" e la versione del server
- > `winmerge [file]` - effettua un confronto fra il file "file" e la versione del server con winmerge
- > `opendir` - apre la cartella corrente
- > `clear` - pulisce il terminale

Per il codice sviluppato si rimanda all'[appendice 1](#).

Client e server fanno uso di metodi comuni per la gestione dell'interazione con l'utente e la lettura e scrittura di tag su file oltre ad altre operazioni minori. Per questo motivo si è realizzata una ulteriore classe di utilities (uti).

Particolare menzione merita la scrittura di informazioni su file. Si tratta di file scritti su filesystem allo scopo di memorizzare informazioni che non è possibile mantenere in memoria o non è conveniente ricavare di volta in volta.

- Client
 - Percorso corrente per successivi avvii.
 - Numero di versione locale con riferimento alle versioni del server.

- Esclusioni.
 - File modificati in previsione di un commit.
- Server
 - Date di creazione di repository, branch, changesets
 - Changeset di origine per nuovi branch
 - Ultimo changeset di un branch
 - Informazioni sul changeset e sui file modificati

Commit e versioni

Particolare menzione merita il meccanismo di commit e di gestione delle versioni.

Un sistema di versioning di questo tipo deve permettere agli utilizzatori di generare un elevato numero di commit, scaricare versioni complete e aggiornate oppure versioni specifiche, oltre alla possibilità di effettuare operazioni di roll-back delle versioni.

Come già accennato, per limitare la ridondanza, si è deciso di mantenere, per ogni commit, i soli file modificati e non una versione completa. Questa scelta riduce drasticamente l'occupazione di memoria nel caso che i progetti gestiti siano di dimensioni elevate.

Tuttavia tale scelta pone il problema di ricavare una versione completa su richiesta e in modo rapido. La soluzione adottata prevede di creare incrementalmente una versione completa, partendo dall'ultima versione completa e aggiungendo a questa i file memorizzati nei changeset successivi (sovrascrivendo file uguali con l'ultima versione degli stessi).

Questo procedimento però rischia tuttavia di impattare sulle performance e quindi sui tempi necessari a ricavare una versione. Tale operazione è largamente utilizzata anche per altre funzionalità (soprattutto per ricavare le differenze rispetto ad una versione locale).

Per questo si è deciso di scegliere una soluzione ibrida, tale per cui vengono realizzati "changeset di backup" (ossia changeset particolari e identificati come tali che contengono l'intera versione aggiornata). La realizzazione di una versione aggiornata fa quindi riferimento all'ultimo changeset di backup presente, ignorando le modifiche precedenti, e aggiungendo a questo le modifiche successive.

Fanno eccezioni quei file che sono stati rimossi a seguito di un commit, tali file vengono semplicemente eliminati una volta giunti a considerare il relativo commit.

Pending Changes

Pur non essendo un'operazione complessa, il riconoscimento dei file modificati localmente risulta fondamentale e largamente utilizzata anche in altre operazioni.

Per riconoscere i file modificati localmente è necessario innanzitutto scaricare una versione aggiornata in locale, quindi ogni file viene confrontato con il file della versione più aggiornata.

Il confronto avviene comparando le date di modifica dei file e il loro contenuto.

Il confronto permette anche di identificare la tipologia di modifica subita dal file:

- File aggiunto
- File rimosso
- File modificato
- File vecchio (rispetto alla versione aggiornata)

Le operazioni di *commit*, *undo* e *compare* fanno uso dell'operazione di *pending changes*, per identificare i file modificati da inviare, su cui annullare le modifiche o da confrontare.

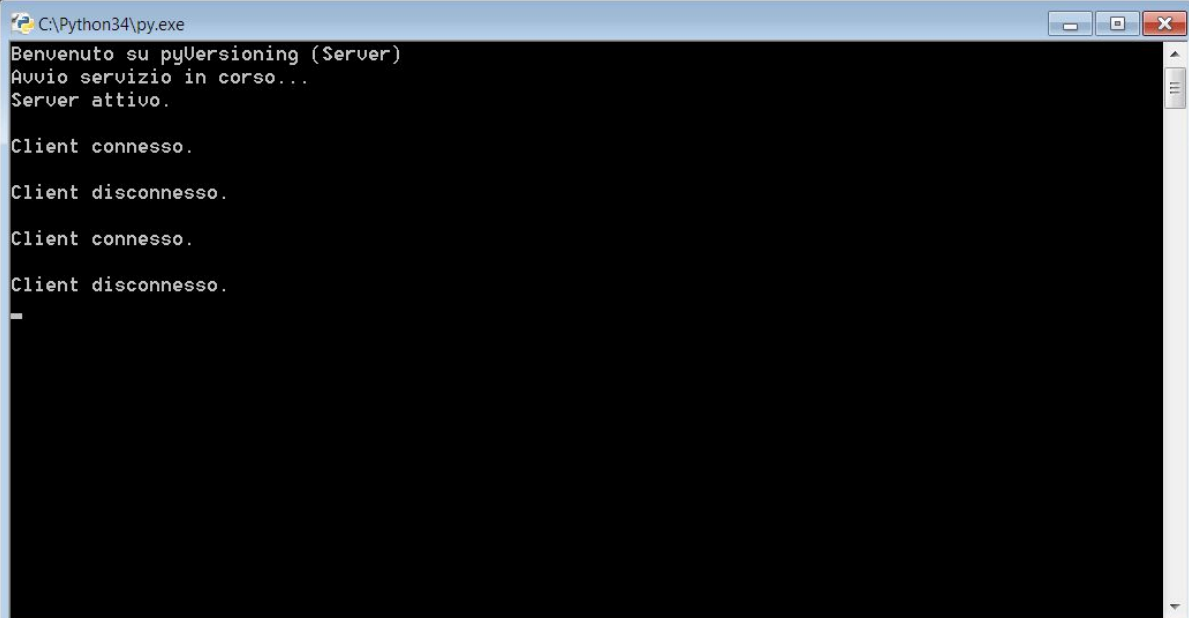
Test

La fase di test finale, prevalentemente di tipo black-box, si è concentrata sul corretto funzionamento ed esecuzione dei comandi impartiti dal client. La verifica è stata effettuata in maniera informale, validando fisicamente i risultati.

Benché auspicato non è stato possibile fornire una beta a terzi per verificare in maniera più efficace il funzionamento complessivo.

Si riportano di seguito pochi test ritenuti più significativi. Per i test è stato utilizzato lo stesso il progetto in oggetto come contenuto di un repository.

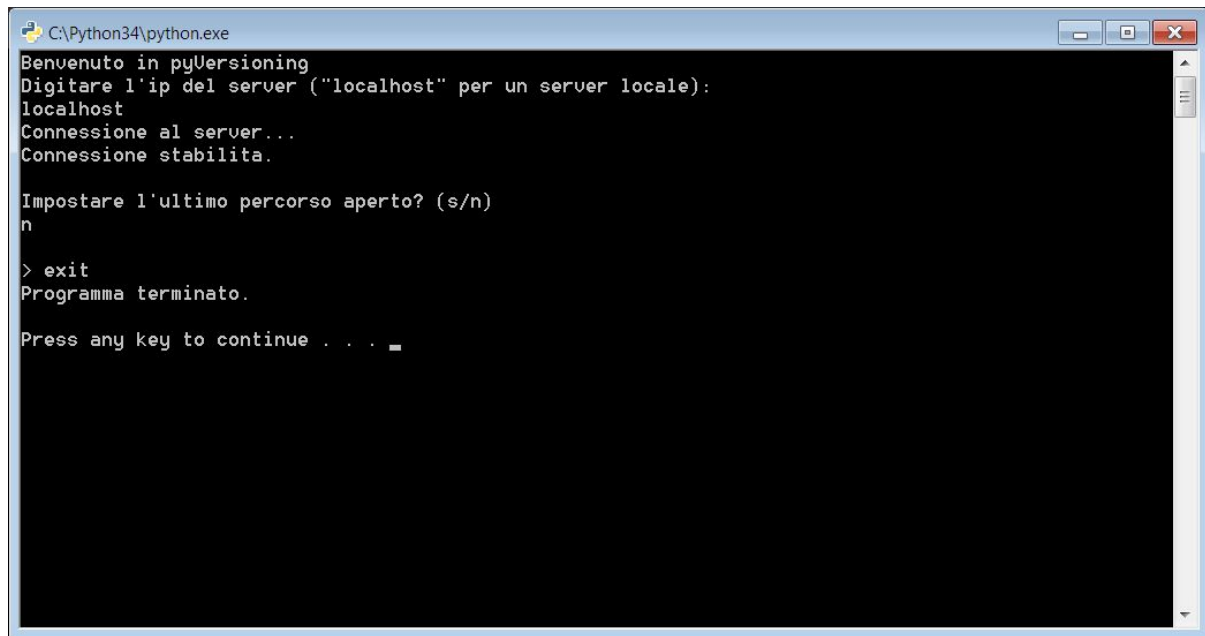
Server



```
C:\Python34\py.exe
Benvenuto su pyVersioning (Server)
Avvio servizio in corso...
Server attivo.

Client connesso.
Client disconnesso.
Client connesso.
Client disconnesso.
-
```

Client

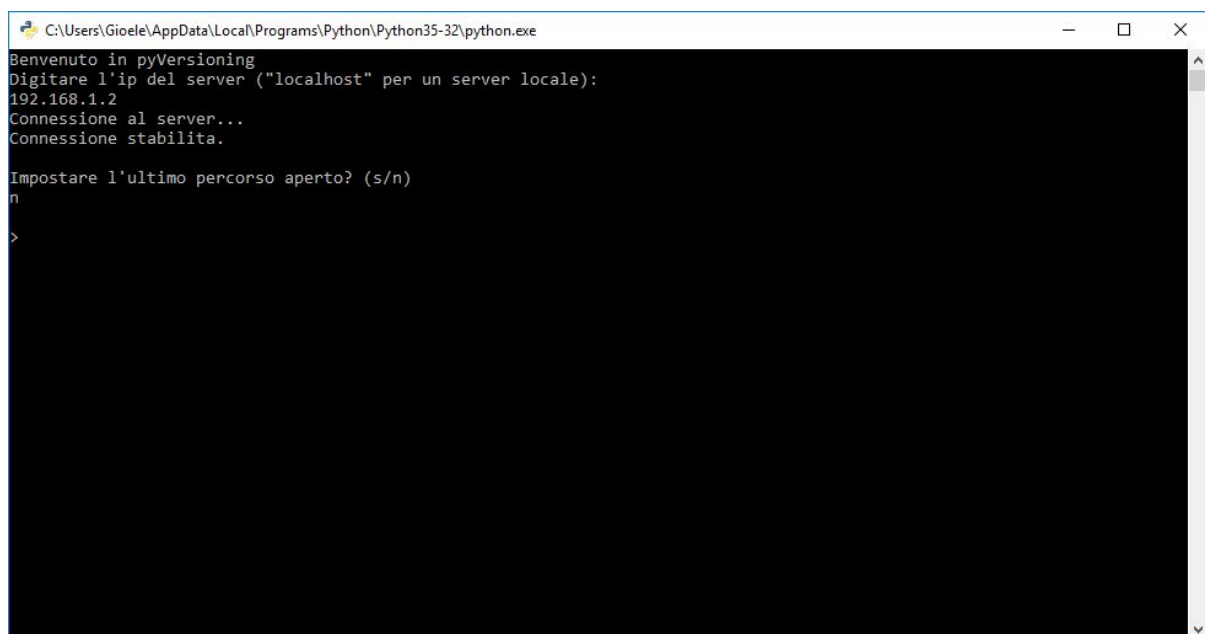


```
C:\Python34\python.exe
Benvenuto in pyVersioning
Digitare l'ip del server ("localhost" per un server locale):
localhost
Connessione al server...
Connessione stabilita.

Impostare l'ultimo percorso aperto? (s/n)
n

> exit
Programma terminato.

Press any key to continue . . .
```

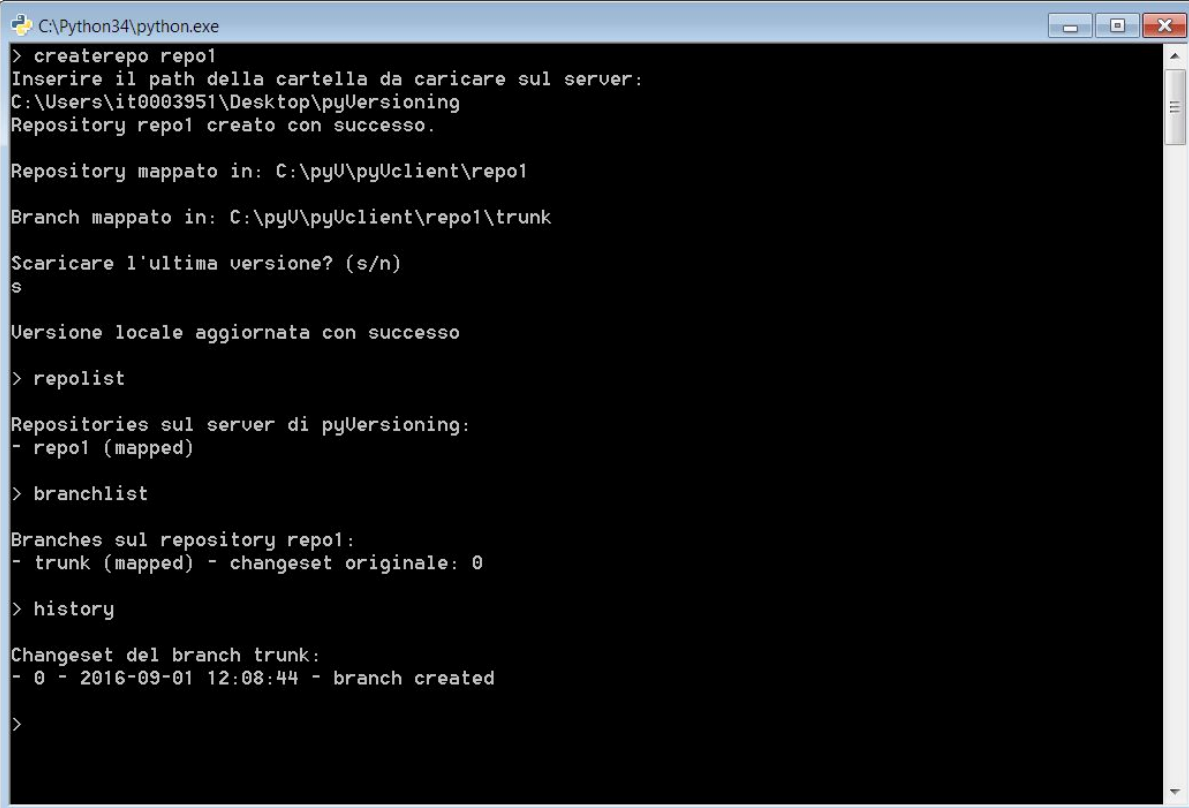


```
C:\Users\Gioele\AppData\Local\Programs\Python\Python35-32\python.exe
Benvenuto in pyVersioning
Digitare l'ip del server ("localhost" per un server locale):
192.168.1.2
Connessione al server...
Connessione stabilita.

Impostare l'ultimo percorso aperto? (s/n)
n

>
```

Creazione di un nuovo repository



```
C:\Python34\python.exe
> createrepo repo1
Inserire il path della cartella da caricare sul server:
C:\Users\it0003951\Desktop\pyVersioning
Repository repo1 creato con successo.

Repository mappato in: C:\pyU\pyUclient\repo1
Branch mappato in: C:\pyU\pyUclient\repo1\trunk
Scaricare l'ultima versione? (s/n)
s
Versione locale aggiornata con successo

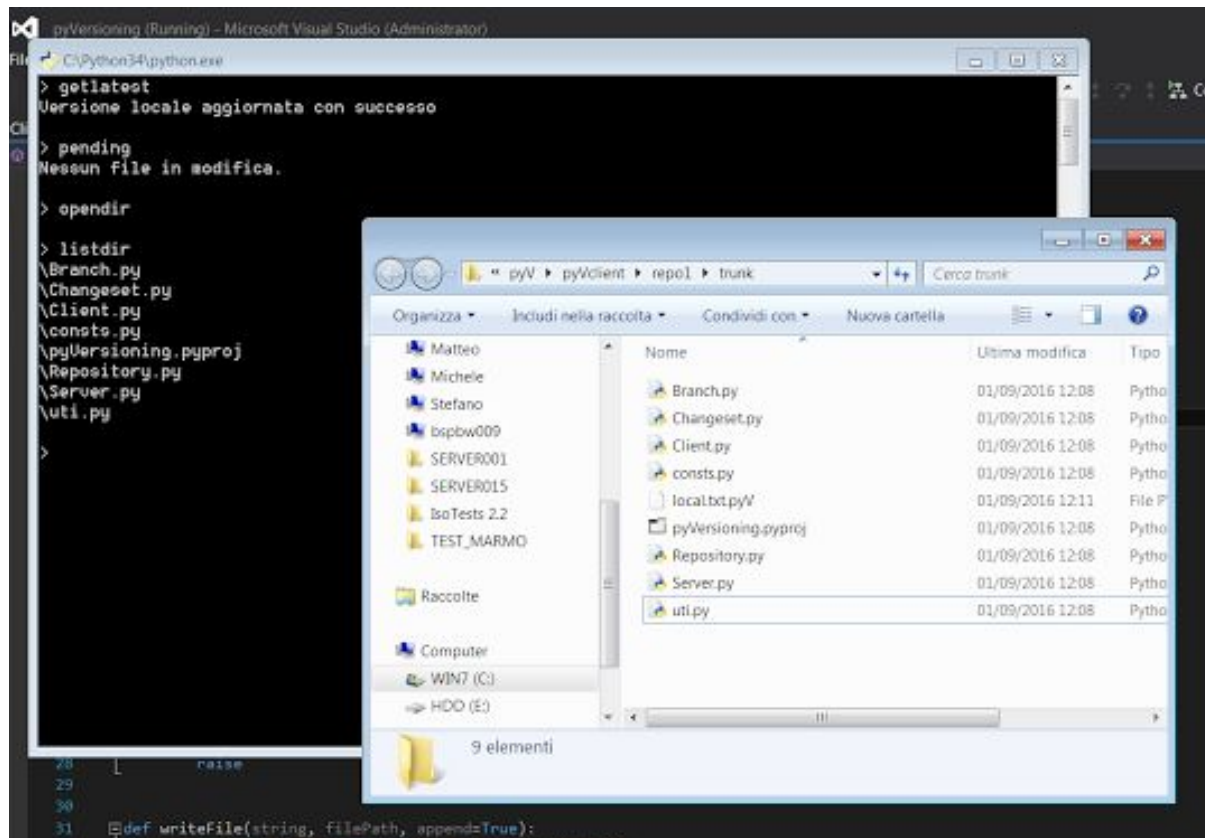
> repolist
Repositories sul server di pyVersioning:
- repo1 (mapped)

> branchlist
Branches sul repository repo1:
- trunk (mapped) - changeset originale: 0

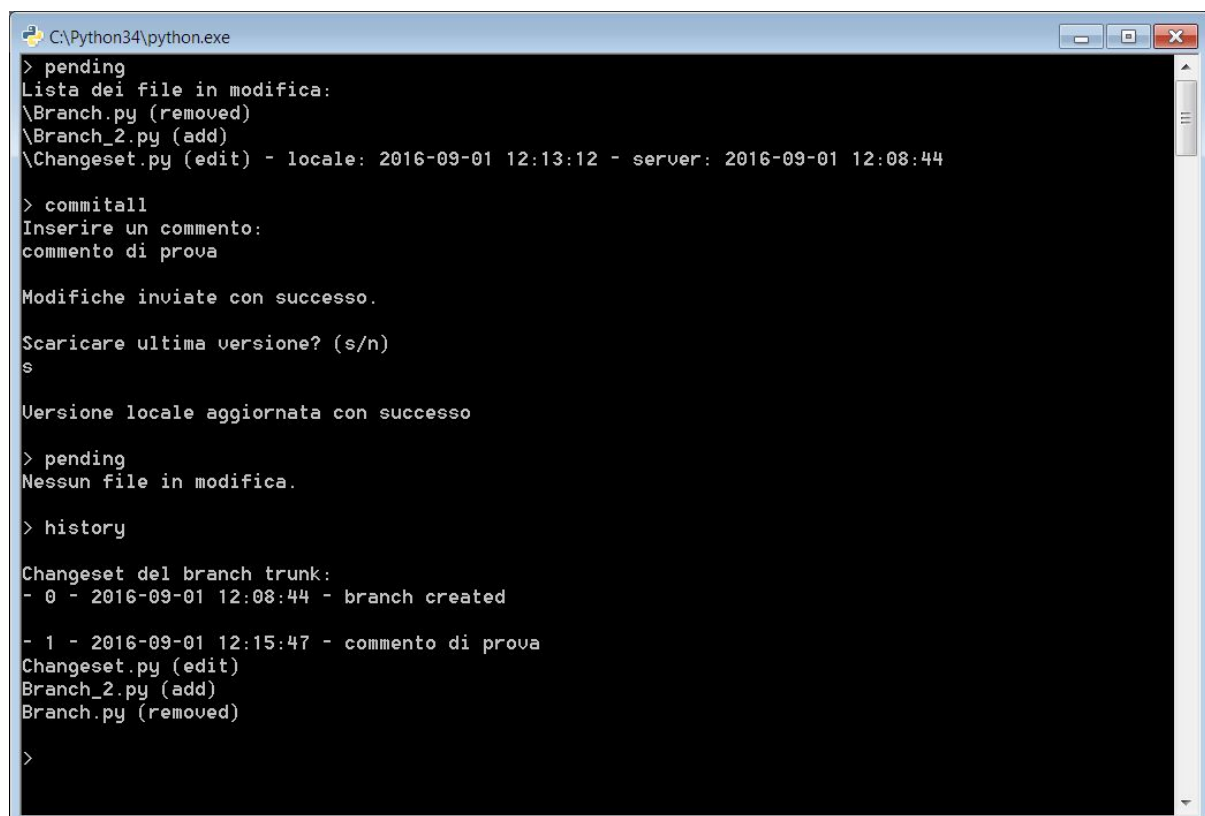
> history
Changeset del branch trunk:
- 0 - 2016-09-01 12:08:44 - branch created

>
```

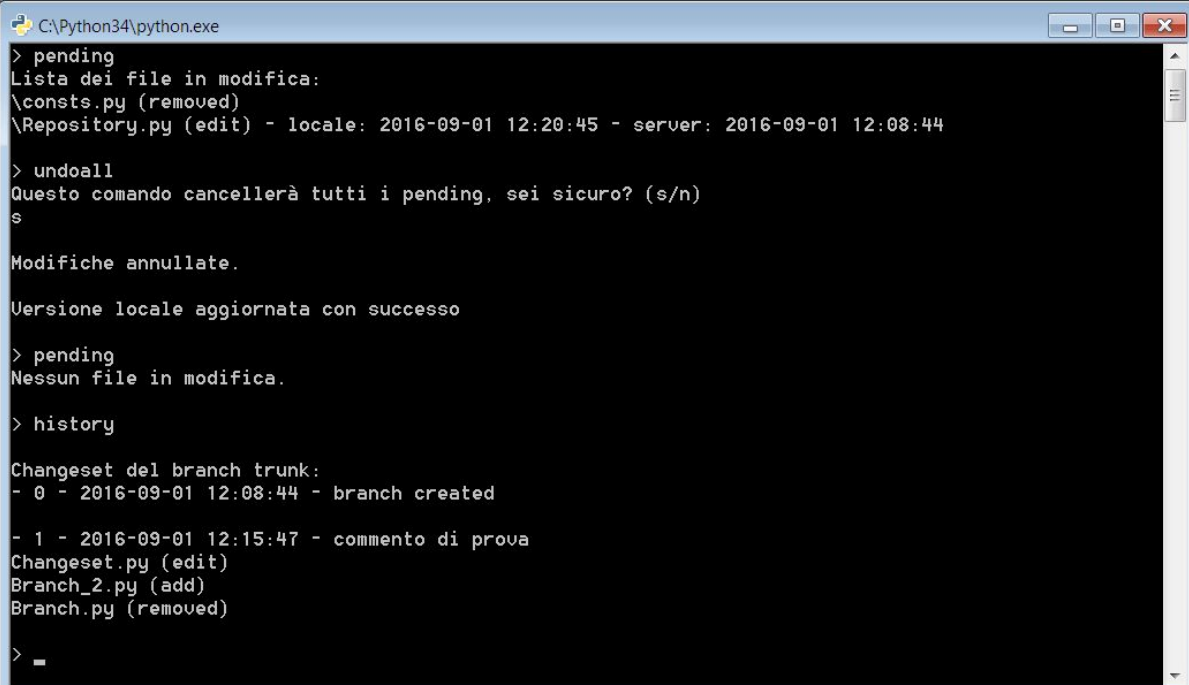
Scarico l'ultima versione



Commit



Undo



```
C:\Python34\python.exe
> pending
Lista dei file in modifica:
\consts.py (removed)
\Repository.py (edit) - locale: 2016-09-01 12:20:45 - server: 2016-09-01 12:08:44

> undoall
Questo comando cancellerà tutti i pending, sei sicuro? (s/n)
s

Modifiche annullate.

Versione locale aggiornata con successo

> pending
Nessun file in modifica.

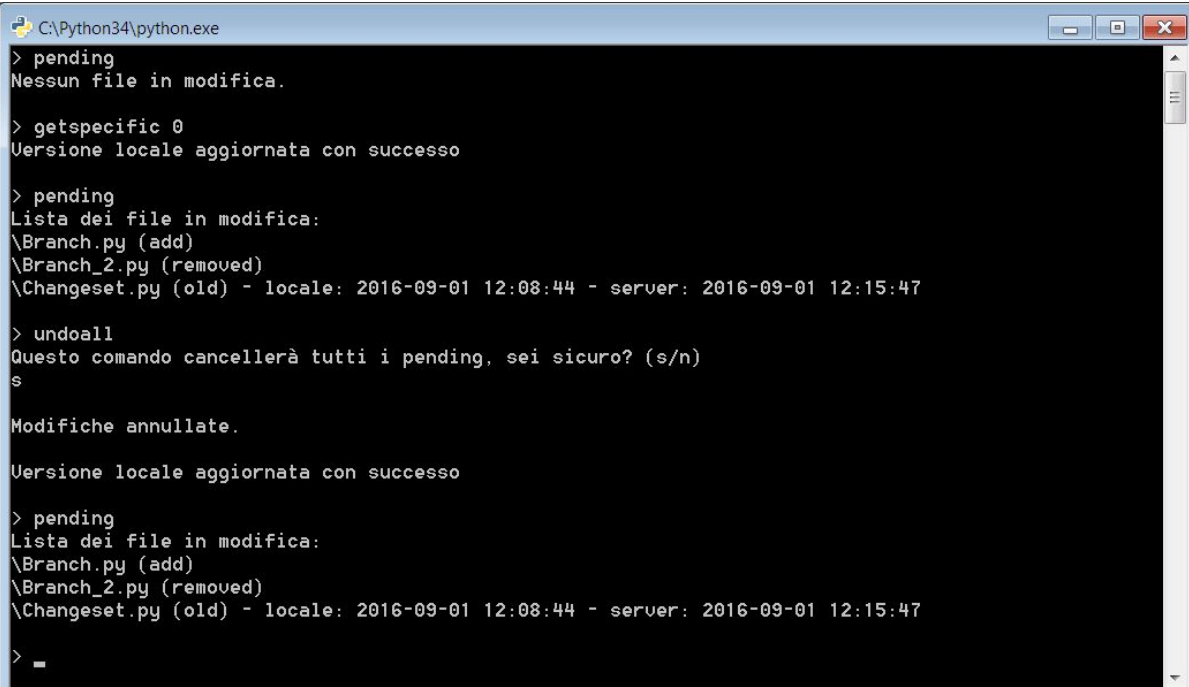
> history

Changeset del branch trunk:
- 0 - 2016-09-01 12:08:44 - branch created

- 1 - 2016-09-01 12:15:47 - commento di prova
Changeset.py (edit)
Branch_2.py (add)
Branch.py (removed)

> _
```

Scarico una versione specifica. Undo alla versione originale.



```
C:\Python34\python.exe
> pending
Nessun file in modifica.

> getspecific 0
Versione locale aggiornata con successo

> pending
Lista dei file in modifica:
\Branch.py (add)
\Branch_2.py (removed)
\Changeset.py (old) - locale: 2016-09-01 12:08:44 - server: 2016-09-01 12:15:47

> undoall
Questo comando cancellerà tutti i pending, sei sicuro? (s/n)
s

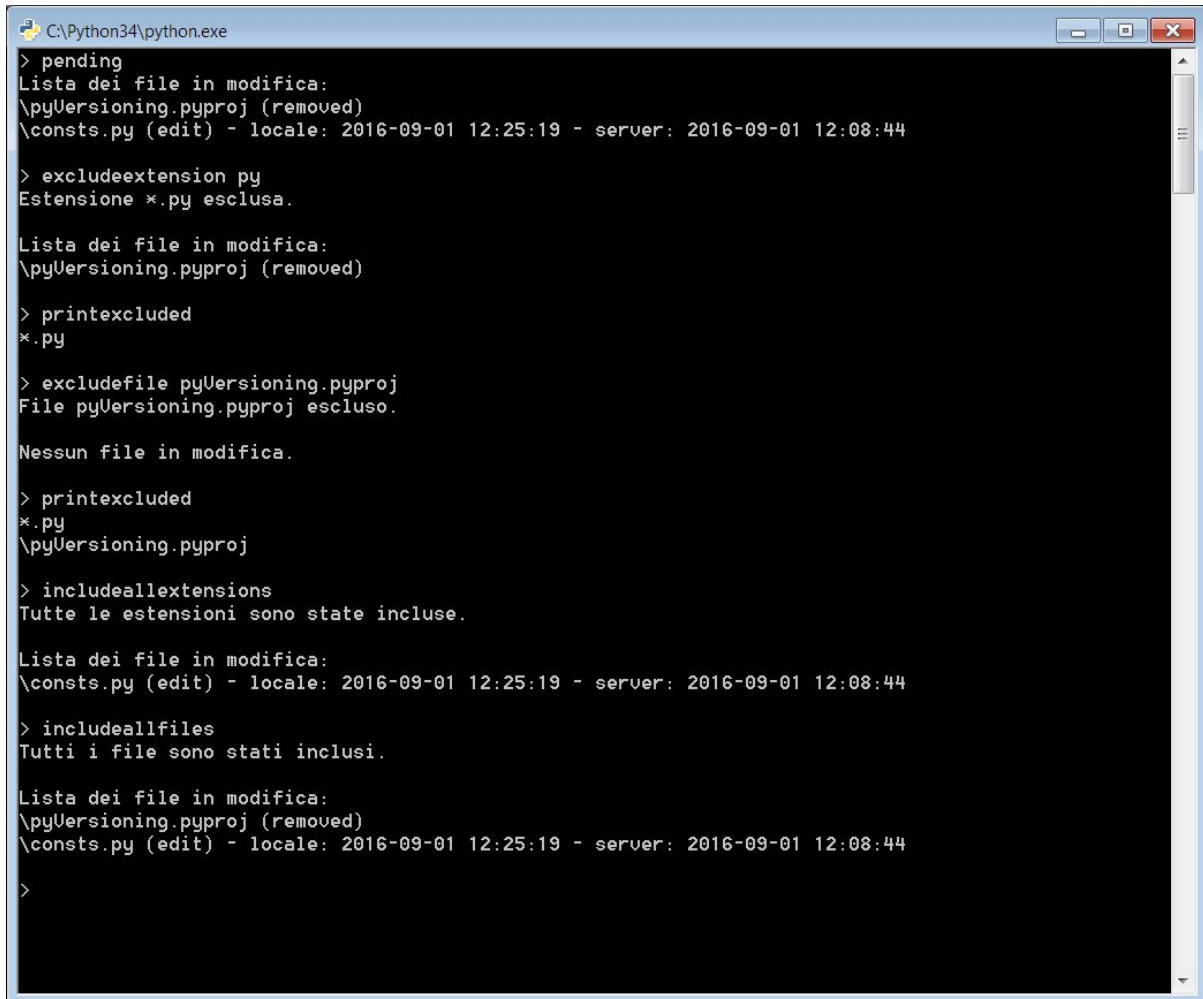
Modifiche annullate.

Versione locale aggiornata con successo

> pending
Lista dei file in modifica:
\Branch.py (add)
\Branch_2.py (removed)
\Changeset.py (old) - locale: 2016-09-01 12:08:44 - server: 2016-09-01 12:15:47

> _
```

Esclusioni e inclusioni

A screenshot of a Windows command prompt window titled "C:\Python34\python.exe". The window has a black background with white text. The text shows a series of commands and their outputs in Italian. The commands include: > pending, > excludeextension py, > printexcluded, > excludefile pyVersioning.pyproj, > includeallextensions, and > includeallfiles. The outputs list files in modification, such as \pyVersioning.pyproj (removed) and \consts.py (edit), along with timestamps for locale and server. The window has standard Windows window controls (minimize, maximize, close) in the top right corner.

```
C:\Python34\python.exe
> pending
Lista dei file in modifica:
\pyVersioning.pyproj (removed)
\consts.py (edit) - locale: 2016-09-01 12:25:19 - server: 2016-09-01 12:08:44

> excludeextension py
Estensione *.py esclusa.

Lista dei file in modifica:
\pyVersioning.pyproj (removed)

> printexcluded
*.py

> excludefile pyVersioning.pyproj
File pyVersioning.pyproj escluso.

Nessun file in modifica.

> printexcluded
*.py
\pyVersioning.pyproj

> includeallextensions
Tutte le estensioni sono state incluse.

Lista dei file in modifica:
\consts.py (edit) - locale: 2016-09-01 12:25:19 - server: 2016-09-01 12:08:44

> includeallfiles
Tutti i file sono stati inclusi.

Lista dei file in modifica:
\pyVersioning.pyproj (removed)
\consts.py (edit) - locale: 2016-09-01 12:25:19 - server: 2016-09-01 12:08:44

>
```

Creazione branch e rimozione branch locale

```
C:\Python34\python.exe
> branchlist

Branches sul repository repo1:
- trunk (mapped) - changeset originale: 0

> history

Changeset del branch trunk:
- 0 - 2016-09-01 12:08:44 - branch created
- 1 - 2016-09-01 12:15:47 - commento di prova
Changeset.py (edit)
Branch_2.py (add)
Branch.py (removed)

> createbranch branch2
Branch branch2 creato con successo.

Branch mappato in: C:\pyU\pyUclient\repo1\branch2

Scaricare l'ultima versione? (s/n)
s

Versione locale aggiornata con successo

> branchlist

Branches sul repository repo1:
- trunk (mapped) - changeset originale: 0
- branch2 (mapped) - changeset originale: 1

> dropbranch branch2
Rimuovere la cartella C:\pyU\pyUclient\repo1\branch2? (s/n)
s

Cartella rimossa: C:\pyU\pyUclient\repo1\branch2

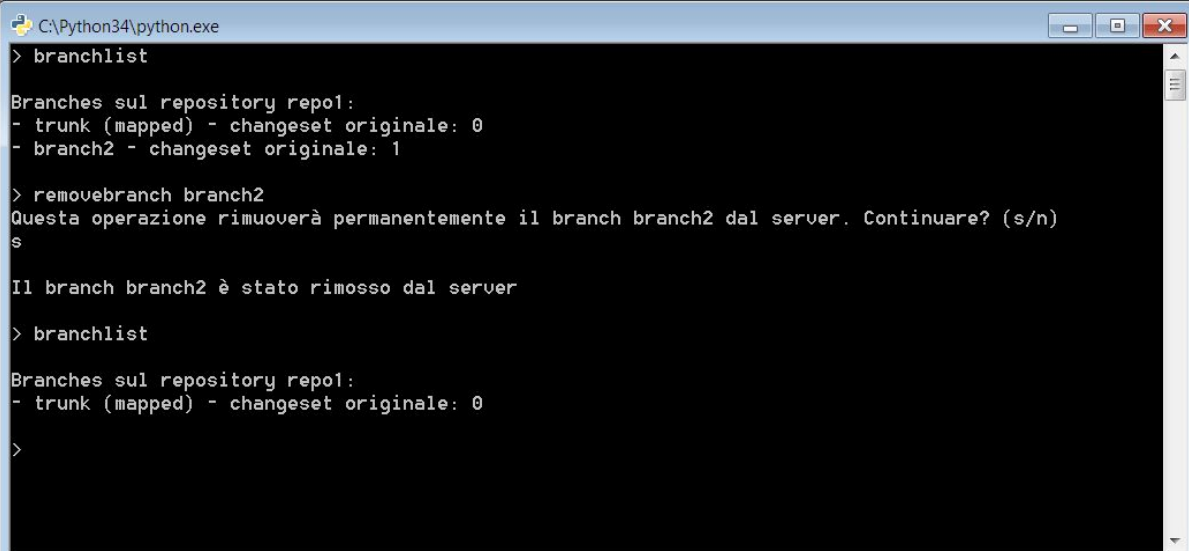
> C:\pyU\pyUclient\repo1\

> branchlist

Branches sul repository repo1:
- trunk (mapped) - changeset originale: 0
- branch2 - changeset originale: 1

>
```


Rimozione di branch dal server



```
C:\Python34\python.exe
> branchlist

Branches sul repository repo1:
- trunk (mapped) - changeset originale: 0
- branch2 - changeset originale: 1

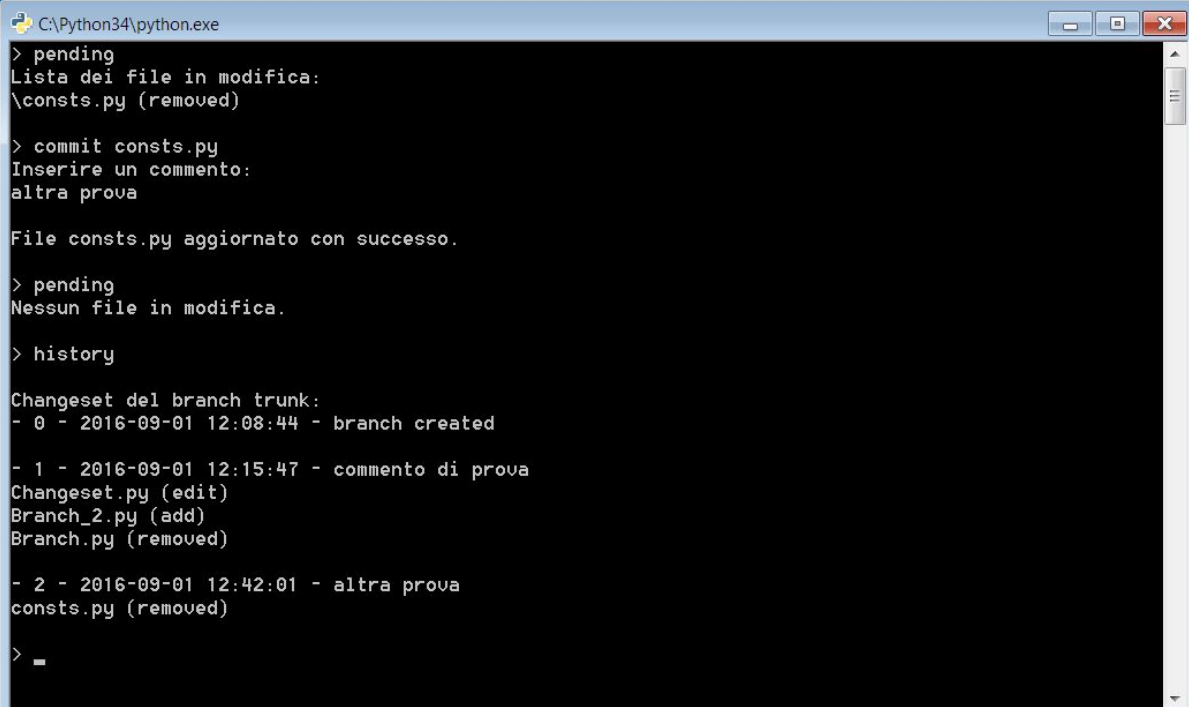
> removebranch branch2
Questa operazione rimuoverà permanentemente il branch branch2 dal server. Continuare? (s/n)
s

Il branch branch2 è stato rimosso dal server

> branchlist

Branches sul repository repo1:
- trunk (mapped) - changeset originale: 0
>
```

Commit di un solo file



```
C:\Python34\python.exe
> pending
Lista dei file in modifica:
\consts.py (removed)

> commit consts.py
Inserire un commento:
altra prova

File consts.py aggiornato con successo.

> pending
Nessun file in modifica.

> history

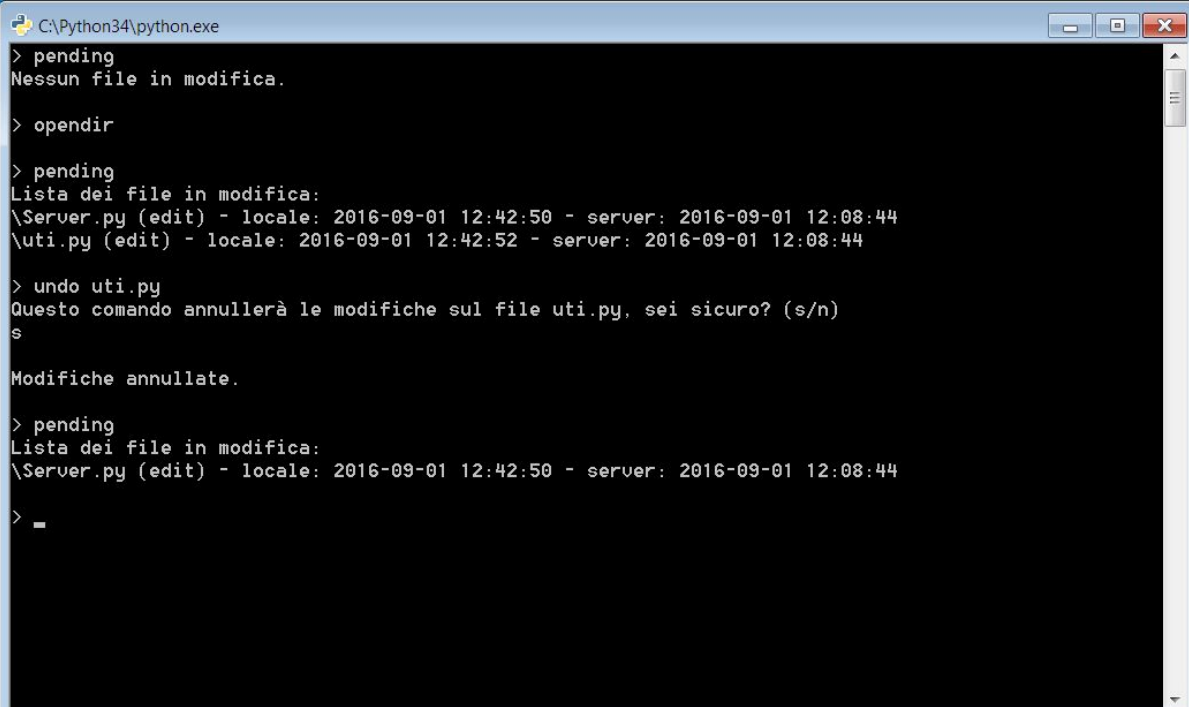
Changeset del branch trunk:
- 0 - 2016-09-01 12:08:44 - branch created

- 1 - 2016-09-01 12:15:47 - commento di prova
Changeset.py (edit)
Branch_2.py (add)
Branch.py (removed)

- 2 - 2016-09-01 12:42:01 - altra prova
consts.py (removed)

> _
```

Undo di un solo file



```
C:\Python34\python.exe
> pending
Nessun file in modifica.

> opendir

> pending
Lista dei file in modifica:
\Server.py (edit) - locale: 2016-09-01 12:42:50 - server: 2016-09-01 12:08:44
\uti.py (edit) - locale: 2016-09-01 12:42:52 - server: 2016-09-01 12:08:44

> undo uti.py
Questo comando annullerà le modifiche sul file uti.py, sei sicuro? (s/n)
s

Modifiche annullate.

> pending
Lista dei file in modifica:
\Server.py (edit) - locale: 2016-09-01 12:42:50 - server: 2016-09-01 12:08:44

> _
```

Rollback

```
C:\Python34\py.exe
> history

Changeset del branch trunk:
- 0 - 2016-09-01 12:08:44 - branch created

- 1 - 2016-09-01 12:15:47 - commento di prova
Changeset.py (edit)
Branch_2.py (add)
Branch.py (removed)

- 2 - 2016-09-01 12:42:01 - altra prova
consts.py (removed)

- 3 - 2016-09-01 13:09:23 - prova
uti.py (edit)

> rollback 2
Versione locale aggiornata con successo

Inserire un commento:
rollback a cs 2

Modifiche inviate con successo.

Scaricare ultima versione? (s/n)
s

Versione locale aggiornata con successo

> history

Changeset del branch trunk:
- 0 - 2016-09-01 12:08:44 - branch created

- 1 - 2016-09-01 12:15:47 - commento di prova
Changeset.py (edit)
Branch_2.py (add)
Branch.py (removed)

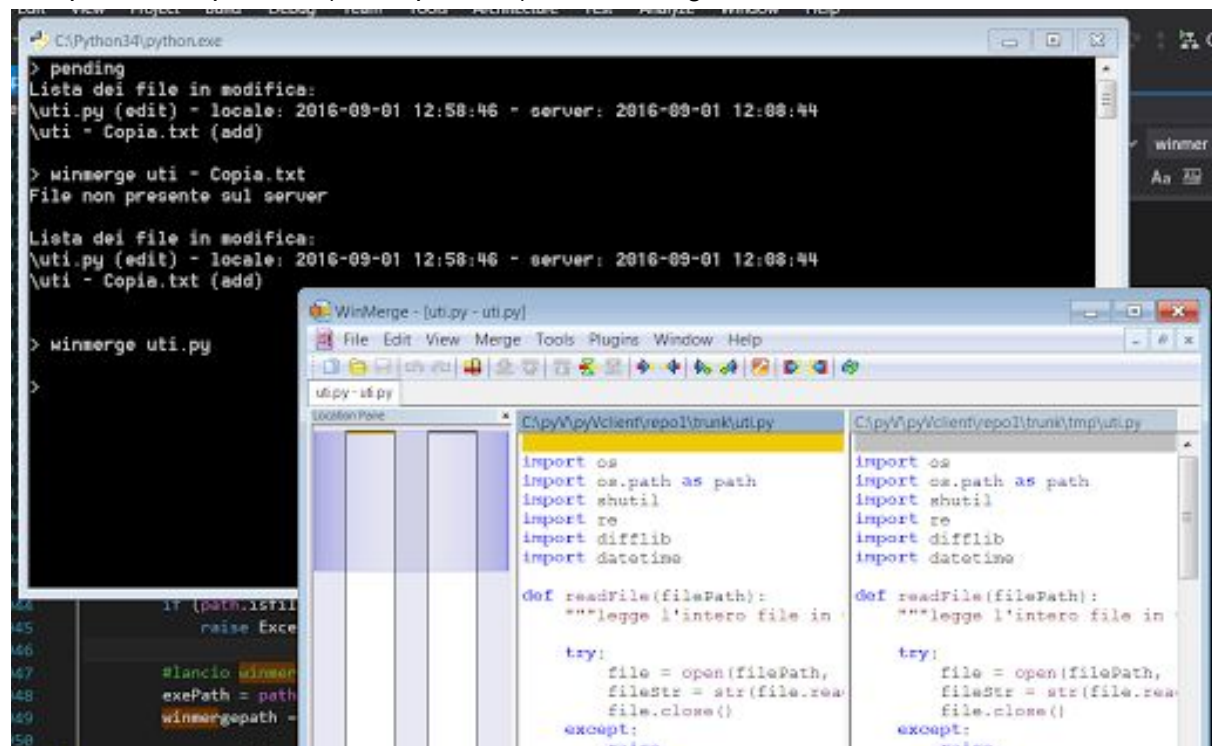
- 2 - 2016-09-01 12:42:01 - altra prova
consts.py (removed)

- 3 - 2016-09-01 13:09:23 - prova
uti.py (edit)

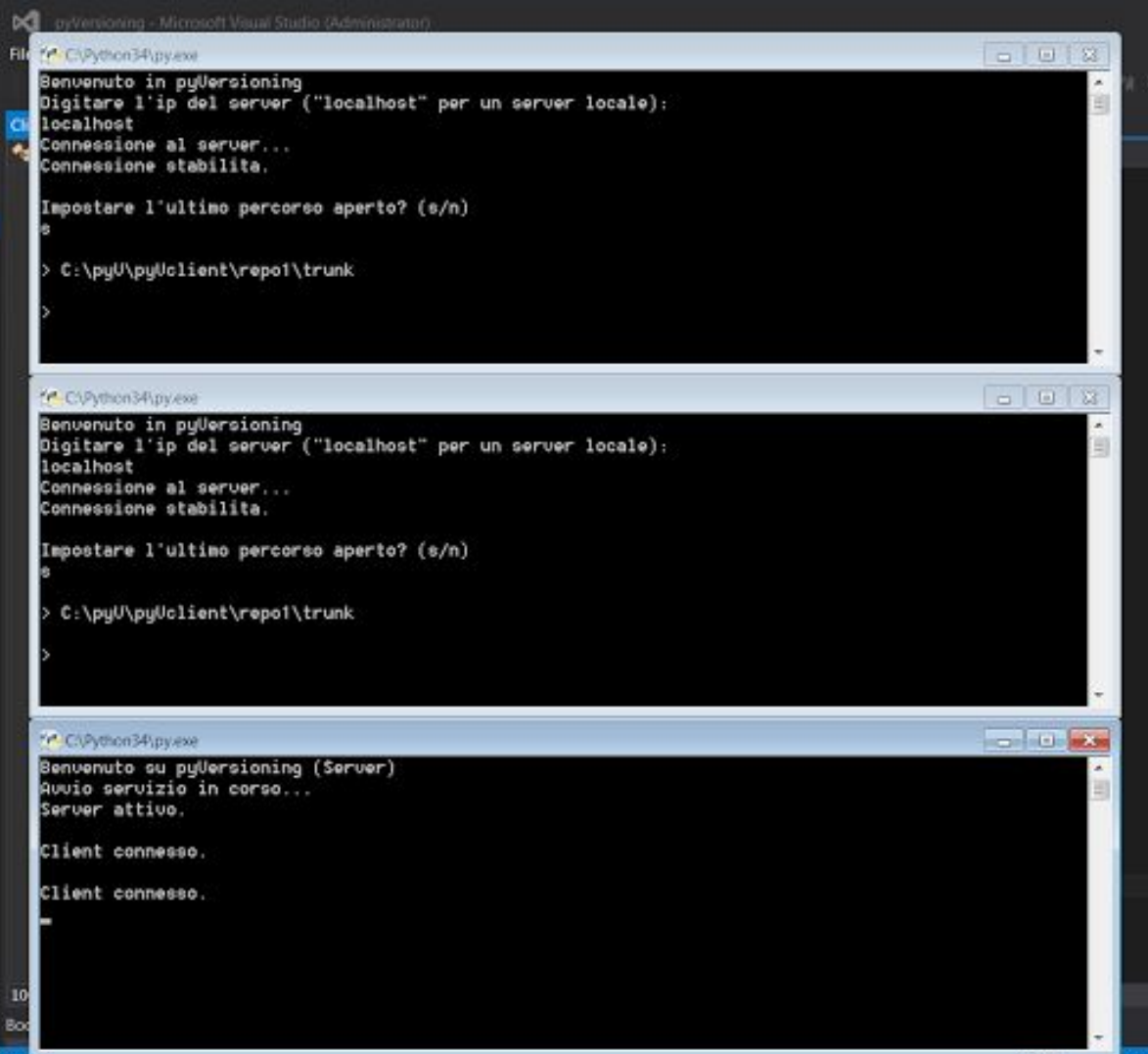
- 4 - 2016-09-01 13:11:00 - rollback a cs 2
uti.py (old)

>
```

Compare di file presenti (e non presenti) con winmerge



Connessione di più client



```
pyVersioning - Microsoft Visual Studio (Administrator)
File Edit View Tools Window Help
C:\Python34\py.exe
Benvenuto in pyVersioning
Digitare l'ip del server ("localhost" per un server locale):
localhost
Connessione al server...
Connessione stabilita.

Impostare l'ultimo percorso aperto? (s/n)
s
> C:\pyU\pyUclient\repo1\trunk
>

C:\Python34\py.exe
Benvenuto in pyVersioning
Digitare l'ip del server ("localhost" per un server locale):
localhost
Connessione al server...
Connessione stabilita.

Impostare l'ultimo percorso aperto? (s/n)
s
> C:\pyU\pyUclient\repo1\trunk
>

C:\Python34\py.exe
Benvenuto su pyVersioning (Server)
Avvio servizio in corso...
Server attivo.

Client connesso.

Client connesso.

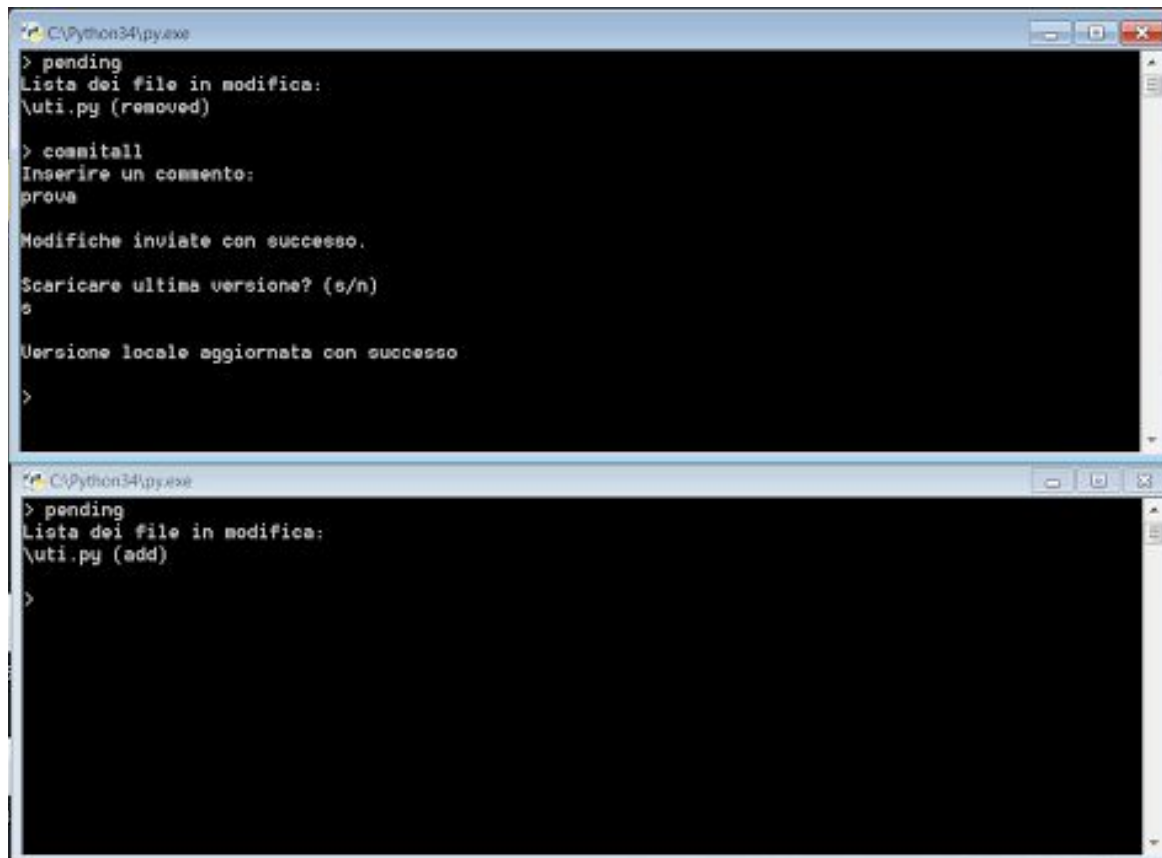
10
Boo
Ready 11:55:00 Col
```

Verifica comportamento con più client connessi.

I due utenti hanno scaricato l'ultima versione nello stesso momento.

Il primo utente rimuove il file `uti.py` ed effettua un commit

Il secondo utente verifica i file in pending e come atteso trova ancora il file `uti.py` nella versione locale.



```
C:\Python34\pyaxe
> pending
Lista dei file in modifica:
\uti.py (removed)

> commitall
Inserire un commento:
prova

Modifiche inviate con successo.

Scaricare ultima versione? (s/n)
s

Versione locale aggiornata con successo

>
```

```
C:\Python34\pyaxe
> pending
Lista dei file in modifica:
\uti.py (add)

>
```

Conclusioni

Tecniche utilizzate

La fase di progettazione del progetto ha seguito le regole della progettazione top-down, con introduzione delle macro-funzionalità richieste per un progetto di questo tipo che sono state via via suddivise e raffinate secondo una tecnica di tipo divide-et-impera.

La realizzazione del progetto è stata guidata fortemente da test di tipo white-box e da uno sviluppo incrementale di tipo bottom-up. Singoli moduli strutturati a livello teorico sono stati sviluppati e testati con porzioni di codice nel ruolo di software-stub (non riportate e non conservate in quanto obsolete per la versione finale). I singoli moduli sono poi stati collegati e testati incrementalmente per garantire la stabilità di sotto-moduli destinati a comporre moduli più ampi.

Si è preferito dare la precedenza alle parti che costituiscono la logica funzionale del programma per implementare poi la parte di comunicazione fra i processi e di interazione con l'utente. Questo approccio ha lo scopo di dividere la logica funzionale dal resto del software, partendo quindi da un kernel ben strutturato da interrogare dall'esterno.

Con l'introduzione di porzioni di software più esterne si è resa necessaria un'opera di refactoring svolta a più riprese per adeguare e migliorare la struttura sottostante.

Sviluppi futuri

Nell'eventualità di dedicare futuri sviluppi a questo progetto, i seguenti punti potrebbero essere di particolare interesse:

- gestione di statistiche e di tracciabilità del codice
- sviluppo di un'interfaccia grafica più user-friendly
- aumento delle funzionalità del server
- gestione e risoluzione di conflitti
- merging di due branches
- supporto di una più vasta gamma di file
- scrittura di file di sistema in formati più idonei (XML)
- sviluppo e integrazione di un software di gestione del processo di sviluppo

Stime e sforzo dello sviluppo

La scelta del linguaggio non ha particolari motivazioni, l'obiettivo prefissato era un acquisizione di competenze e familiarità con un linguaggio non conosciuto.

La scelta degli strumenti di sviluppo è legata alla familiarità con gli stessi, non sono state prese in considerazione alternative.

Le stime iniziali sono state fatte in modo del tutto euristico. Causa la mancanza di esperienza relativa a stimare l'entità del lavoro, la scarsa conoscenza del linguaggio e l'introduzione di nuove funzionalità durante la fase di sviluppo, le stime fatte in fase di progettazione non sono stati rispettate. Con l'acquisizione di familiarità con il linguaggio si sono resi necessari refactoring non inizialmente previsti, mentre con la realizzazione delle funzionalità previste si sono rese utili funzionalità non considerate in precedenza.

Si stima intuitivamente un incremento di circa il 40% del tempo impiegato per lo sviluppo e di un 50% del codice sviluppato. La progettazione e il calcolo dei rischi sono invece rientrati nelle stime. Non richiede particolare menzione la fase di analisi del problema in quanto si trae facilmente ispirazione da piattaforme simili già ben note.

Il risultato finale soddisfa le specifiche prefissate e presenta un discreto numero di funzionalità aggiuntive.

Appendice 1

Codice

Client.py

```
import os
import os.path as path
import datetime
import shutil
import filecmp
import socket
import rpyc
import inspect
import uti
from consts import *

class Client:

    root = ""
    _currPath = ""
    _currRepo = ""
    _currBranch = ""
    server = None

    def __init__(self, conn):

        #setto il path della cartella root
        self.root = "C:\pyV\pyVclient"
        if (path.isdir(self.root) == False):
            os.makedirs(self.root)
        self.currPath = self.root

        #setto il riferimento alla connessione con il server
        self.server = conn.root
        #chiedo all'utente se desidera impostare l'ultimo percorso
        usato
        if (uti.askQuestion("Impostare l'ultimo percorso aperto?")):
            #setto il repository se memorizzato nel file
            try:
                self.setRepo(uti.readFileByTag(LAST_REPO,
self.lastRunFile)[0])

            #setto il branch se memorizzato nel file
```

```

        self.setBranch(uti.readFileByTag(LAST_BRANCH,
self.lastRunFile)[0])
        self.printCurrPath()
    except:
        print("Impossibile effettuare l'operazione",
end="\n\n")

```

```

def copyDirToClient(self, dirFrom, dirTo):
    """copia la cartella "dirFrom" nella cartella "dirTo" """

    #sovrascrivo la cartella
    if (path.isdir(dirTo)):
        shutil.rmtree(dirTo)
    os.makedirs(dirTo)

    #copio tutti i file contenuti
    for file in self.server.listdir(dirFrom):
        self.copyFileToClient(file, file.replace(dirFrom,
dirTo))

```

```

def copyFileToClient(self, fileFrom, fileTo):
    """copia il file "fileFrom" del server nel path "fileTo" """

    remoteFile = self.server.File()
    remoteFile.open(fileFrom)

    #copio il file del server nella cartella temporanea
    localFile = open(fileTo, "w")

    shutil.copyfileobj(remoteFile, localFile)

    remoteFile.close()
    localFile.close()

    #copio la data di ultima modifica dal file del server
    editTime = remoteFile.getmtime()
    os.utime(fileTo, (int(editTime), int(editTime)))

```

```

def copyDirToServer(self, dirFrom, dirTo):
    """copia la cartella "dirFrom" nella cartella "dirTo" """

    #controllo che la cartella dirFrom esista
    if (path.isdir(dirFrom) == False):

```

```

        raise Exception

    #copio tutti i file contenuti
    for file in uti.listdir(dirFrom):
        self.copyFileToServer(file, file.replace(dirFrom,
dirTo))

def copyFileToServer(self, fileFrom, fileTo):
    """copia il file "fileFrom" del client nel path "fileTo" sul
server"""

    localFile = open(fileFrom, errors="ignore")

    remoteFile = self.server.File()
    remoteFile.open(fileTo, "w")

    shutil.copyfileobj(localFile, remoteFile)

    remoteFile.close()
    localFile.close()

def runMenu(self):
    """esegue i comandi dell'utente fino al comando "exit" """

    #eseguo i comandi dell'utente
    while True:
        print("> ", end="")
        userInput = input()
        self.menu(userInput)

        #il programma termina con il comando "exit"
        if (userInput == "exit"):
            break

def menu(self, userInput):
    """esegue il comando "userInput" """

    #costruisco una lista di comando e argomenti
    commandList = userInput.split()
    commandList.reverse()

    #eseguo il comando
    try:

```

```

try:
    command = commandList.pop()
except:
    raise Exception("Valore non ammesso")

if (command == "exit"):
    self.checkCommand(commandList)
    #memorizzo gli ultimi repo/branch settati
    uti.writeFileByTag(LAST_REPO, self._currRepo,
self.lastRunFile)
    uti.writeFileByTag(LAST_BRANCH, self.currBranch,
self.lastRunFile)
    print("Programma terminato.", end="\n\n")

elif (command == "clear"):
    self.checkCommand(commandList)
    os.system("cls")

elif (command == "opendir"):
    self.checkCommand(commandList, paramNum=0)
    os.system("explorer
{}".format(self.getCurrPath()))
    print()

elif (command == "currrdir"):
    self.checkCommand(commandList, paramNum=0)
    self.printCurrPath()

elif (command == "listdir"):
    self.checkCommand(commandList, paramNum=0,
checkRepo=True, checkBranch=True)
    self.listDir()

elif (command == "createrepo"):
    self.checkCommand(commandList, paramNum=1)
    self.createRepo(commandList.pop())

elif (command == "createbranch"):
    self.checkCommand(commandList, paramNum=1,
checkRepo=True)
    self.createBranch(commandList.pop())

elif (command == "removerepo"):
    self.checkCommand(commandList, paramNum=1)
    self.removeRepo(commandList.pop())

```

```

elif (command == "removebranch"):
    self.checkCommand(commandList, paramNum=1,
checkRepo=True)
    self.removeBranch(commandList.pop())

elif (command == "repolist"):
    self.checkCommand(commandList, paramNum=0)
    self.showRepos()

elif (command == "branchlist"):
    self.checkCommand(commandList, paramNum=0,
checkRepo=True)
    self.showBranches()

elif (command == "maprepo"):
    self.checkCommand(commandList, paramNum=1)
    self.mapRepo(commandList.pop())

elif (command == "mapbranch"):
    self.checkCommand(commandList, paramNum=1,
checkRepo=True)
    self.mapBranch(commandList.pop())

elif (command == "droprepo"):
    self.checkCommand(commandList, paramNum=1)
    self.removeRepoMap(commandList.pop())

elif (command == "dropbranch"):
    self.checkCommand(commandList, paramNum=1,
checkRepo=True)
    self.removeBranchMap(commandList.pop())

elif (command == "setrepo"):
    self.checkCommand(commandList, paramNum=1)
    self.setRepo(commandList.pop())
    self.printCurrPath()

elif (command == "setbranch"):
    self.checkCommand(commandList, paramNum=1,
checkRepo=True)
    branchName = commandList.pop()
    self.setBranch(branchName)
    self.printCurrPath()

elif (command == "history"):

```

```

        self.checkCommand(commandList, paramNum=0,
checkRepo=True, checkBranch=True)
        self.showHistory()

        elif (command == "getlatest"):
            self.checkCommand(commandList, paramNum=0,
checkRepo=True, checkBranch=True)
            self.getLatestVersion()

        elif (command == "getspecific"):
            self.checkCommand(commandList, paramNum=1,
checkRepo=True, checkBranch=True)
            self.getSpecificVersion(int(commandList.pop()))

        elif (command == "rollback"):
            self.checkCommand(commandList, paramNum=1,
checkRepo=True, checkBranch=True)
            self.getSpecificVersion(int(commandList.pop()))
            self.commitAll()

        elif (command == "pending"):
            self.checkCommand(commandList, paramNum=0,
checkRepo=True, checkBranch=True)
            self.printPendingChanges()

        elif (command == "excludeextension"):
            self.checkCommand(commandList, paramNum=1,
checkRepo=True, checkBranch=True)
            self.excludeExtension(commandList.pop())
            self.printPendingChanges()

        elif (command == "excludefile"):
            self.checkCommand(commandList, checkRepo=True,
checkBranch=True)
            commandList.reverse()
            self.excludeFile(" ".join(commandList))
            self.printPendingChanges()

        elif (command == "includeextension"):
            self.checkCommand(commandList, paramNum=1,
checkRepo=True, checkBranch=True)
            self.includeExtension(commandList.pop())
            self.printPendingChanges()

        elif (command == "includeallextensions"):

```

```

        self.checkCommand(commandList, paramNum=0,
checkRepo=True, checkBranch=True)
        self.includeAllExtension()
        self.printPendingChanges()

    elif (command == "includefile"):
        self.checkCommand(commandList, checkRepo=True,
checkBranch=True)

        commandList.reverse()
        self.includeFile(" ".join(commandList))
        self.printPendingChanges()

    elif (command == "includeallfiles"):
        self.checkCommand(commandList, paramNum=0,
checkRepo=True, checkBranch=True)
        self.includeAllFile()
        self.printPendingChanges()

    elif (command == "includeall"):
        self.checkCommand(commandList, paramNum=0,
checkRepo=True, checkBranch=True)
        self.includeAllExtension()
        self.includeAllFile()
        self.printPendingChanges()

    elif (command == "printexcluded"):
        self.checkCommand(commandList, paramNum=0,
checkRepo=True, checkBranch=True)
        self.printExcluded()

    elif (command == "commit"):
        self.checkCommand(commandList, checkRepo=True,
checkBranch=True)

        commandList.reverse()
        self.commitOne(" ".join(commandList))

    elif (command == "commitall"):
        self.checkCommand(commandList, paramNum=0,
checkRepo=True, checkBranch=True)
        self.commitAll()

    elif (command == "undo"):
        self.checkCommand(commandList, checkRepo=True,
checkBranch=True)

        try:
            commandList.reverse()

```



```

        self.undoFile(" ".join(commandList))
    except:
        self.printPendingChanges()

    elif (command == "undoall"):
        self.checkCommand(commandList, paramNum=0,
checkRepo=True, checkBranch=True)
        self.undoAll()

    elif (command == "compare"):
        self.checkCommand(commandList, checkRepo=True,
checkBranch=True)
        try:
            commandList.reverse()
            self.compare(" ".join(commandList))
        except Exception as e:
            print(e, end="\n\n")
            self.printPendingChanges()

    elif (command == "winmerge"):
        self.checkCommand(commandList, checkRepo=True,
checkBranch=True)
        try:
            commandList.reverse()
            self.merge(" ".join(commandList))
        except Exception as e:
            print(e, end="\n\n")
            self.printPendingChanges()
        print()

    elif (command == "help"):
        self.checkCommand(commandList)
        self.prinHelp()

    else:
        print("Valore non ammesso", end="\n\n")

except Exception as e:
    print(e, end="\n\n")

def checkCommand(self, commandList, paramNum=None, checkRepo=False,
checkBranch=False):
    """controlla che i parametri e il path corrente siano
compatibili con il comando"""

```

```

        if ((paramNum is not None) & (len(commandList) != paramNum)):
            raise Exception("Parametri errati")
        elif ((checkRepo) & (not self.currRepo)):
            raise Exception("Nessun repository settato")
        elif ((checkBranch) & (not self.currBranch)):
            raise Exception("Nessun branch settato")

    def existsRepo(self, repoName):
        """ritorna True se esiste il repository "repoName" in
locale"""

        #ottengo la cartella del repository
        repoDir = path.join(self.root, repoName)
        #verifico che la cartella esista
        return path.isdir(repoDir)

    def existsBranch(self, branchName):
        """ritorna True se esiste il branch "branchName" in locale"""

        #ottengo la cartella del branch
        branchDir = path.join(self.root, self.currRepo, branchName)
        #verifico che la cartella esista
        return path.isdir(branchDir)

    def printCurrPath(self):
        """stampa il path corrente"""

        print("> {}".format(uti.getPathForPrint(self.currPath)),
end="\n\n")

    def listDir(self):
        """stampa tutti i file e sottocartelle del branch selezionato
sul server"""

        branchList = self.server.listBranch(self.currRepo,
self.currBranch)
        for elem in branchList:
            print(uti.getPathForPrint(elem))

        print()

```

```

def createRepo(self, repoName):
    """crea un nuovo repository sul server"""

    print("Inserire il path della cartella da caricare sul
server:")
    sourceDir = input()
    if (path.isdir(sourceDir) == False):
        raise Exception("Percorso errato.")

    #creo repository e trunk
    destDir = self.server.addRepo(repoName)
    self.copyDirToServer(sourceDir, destDir)
    print("Repository {} creato con successo.".format(repoName),
end="\n\n")
    self.mapRepo(repoName)
    self.mapBranch(TRUNK)

def createBranch(self, branchName):
    """crea un nuovo branch sul server"""

    self.server.addBranch(self.currRepo, branchName)
    print("Branch {} creato con successo.".format(branchName),
end="\n\n")
    self.mapBranch(branchName)

def removeRepo(self, repoName):
    """rimuove il repository dal server"""

    #verifico che la cartella esista
    if (self.server.existsRepo(repoName) == False):
        raise Exception("Repository {} non
presente".format(repoName))
    else:
        try:
            if (uti.askQuestion("Questa operazione rimuoverà
permanentemente il repository {} dal server.
Continuare?".format(repoName))):
                self.server.removeRepo(repoName)
                print("Il repository {} è stato rimosso dal
server".format(repoName), end="\n\n")

                if (self.existsRepo(repoName)):
                    if (uti.askQuestion("Eliminare anche
la copia locale?")):

```

```

                                self.removeRepoMap(repoName)
        except:
            raise Exception("Impossibile effettuare
l'operazione.")

    def removeBranch(self, branchName):
        """rimuove il branch dal server"""

        if (branchName == TRUNK):
            raise Exception("Impossibile rimuovere il ramo
principale.")

        #verifico che la cartella esista
        if (self.server.existsBranch(self.currRepo, branchName) ==
False):
            raise Exception("Branch {} non
presente".format(branchName))
        else:
            try:
                uti.askQuestion("Questa operazione rimuoverà
permanentemente il branch {} dal server.
Continuare?".format(branchName))
                self.server.removeBranch(self.currRepo,
branchName)

                print("Il branch {} è stato rimosso dal
server".format(branchName), end="\n\n")
                if (self.existsBranch(branchName)):
                    if (uti.askQuestion("Eliminare anche la
copia locale?")):
                        self.removeBranchMap(branchName)
            except:
                raise Exception("Impossibile effettuare
l'operazione.")

    def showRepos(self):
        """mostra la lista dei repository presenti sul server"""

        repolist = self.server.showRepos()
        if (len(repolist) == 0):
            print("\nNessun repository presente sul server di
pyVersioning")
        else:
            print("\nRepositories sul server di pyVersioning:")

```

```

    for repoName in repolist:
        #verifico che la cartella esista in locale
        if (self.existsRepo(repoName)):
            print("- {} ({}).format(repoName, "mapped"))
        else:
            print("- {}".format(repoName))
    print()

def showBranches(self):
    """mostra la lista dei branch presenti sul server"""

    branchList = self.server.showBranches(self.currRepo)
    if (len(branchList) == 0):
        print("\nNessun Branch presente sul repository
{}").format(self.currRepo))
    else:
        print("\nBranches sul repository
{}:".format(self.currRepo))

        for branchName in branchList:
            #verifico se la cartella esiste in locale
            if (self.existsBranch(branchName)):
                print("- {} ({} - changeset originale:
{}").format(branchName, "mapped", branchList[branchName]))
            else:
                print("- {} - changeset originale:
{}").format(branchName, branchList[branchName]))
        print()

def showHistory(self):
    """mostra la lista dei changeset presenti nel branch corrente
con la lista dei file modificati"""

    print("\nChangeset del branch {}:".format(self.currBranch))
    for changeSet in self.server.showChangesets(self.currRepo,
self.currBranch):
        print("- {}".format(changeSet))

def mapRepo(self, repoName):
    """mappa il repository nella cartella del client"""

```

```

        #se il repository esiste sul server, creo una cartella sul
client
        #altrimenti viene generata un'eccezione
        if (self.server.existsRepo(repoName) == False):
            raise Exception("Repository {} non
presente".format(repoName))
        else:
            #ottengo il path del repository
            clientDir = path.join(self.root, repoName)

            #chiedo all'utente se sovrascrivere la cartella
            if (uti.askAndRemoveDir(clientDir, askOverride=True)):
                #mappo il repository nella cartella del client

                os.makedirs(clientDir)
                print("Repository mappato in:", clientDir,

end = "\n\n")

                #setto anche il repository mappato come
repository corrente di default
                self.setRepo(repoName)

def mapBranch(self, branchName):
    """mappa il branch nella cartella del client"""

    #se il branch esiste sul server, creo una cartella sul client
    #altrimenti viene generata un'eccezione
    if (self.server.existsBranch(self.currRepo, branchName) ==
False):
        raise Exception("Branch {} non
presente".format(branchName))
    else:
        #ottengo il path del branch
        clientDir = path.join(self.root, self.currRepo,
branchName)

        #chiedo all'utente se sovrascrivere la cartella
        if (uti.askAndRemoveDir(clientDir, askOverride=True)):
            #mappo il branch nella cartella del client
            branchDir = path.join(self.root, self.currRepo,
branchName)

            os.makedirs(branchDir)

```

```

        #setto il branch mappato come branch corrente di
default
        self.setBranch(branchName)

        print("Branch mappato in:", clientDir, end =
"\n\n")

        if (uti.askQuestion("Scaricare l'ultima
versione?")):
            self.getLatestVersion()

def removeRepoMap(self, repoName):
    """rimuove la cartella del repo sul client"""

    if (self.existsRepo(repoName)):
        repodir = path.join(self.root, repoName)
        if (uti.askAndRemoveDir(repodir)):
            if (self.currRepo == repoName):
                self.currBranch = ""
                self.currRepo = ""
                self.printCurrPath()
            else:
                raise Exception("Repository {} non
presente".format(repoName))

def removeBranchMap(self, branchName):
    """rimuove la cartella del branch sul client"""

    if (self.existsBranch(branchName)):
        branchdir = path.join(self.root, self.currRepo,
branchName)
        if (uti.askAndRemoveDir(branchdir)):
            if (self.currBranch == branchName):
                self.currBranch = ""
                self.printCurrPath()
            else:
                raise Exception("Branch {} non
presente".format(branchName))

def setRepo(self, repoName):
    """setta il repository corrente"""

    #verifico che il repository locale esista

```

```

        if (self.existsRepo(repoName)):
            #aggiorno il repository corrente
            self.currRepo = repoName
            self.currBranch = ""
        else:
            #verifico se esiste sul server
            if (self.server.existsRepo(repoName)):
                print("Il repository {} non è stato
mappato".format(repoName))
                if (uti.askQuestion("Effettuare il map del
repository?")):
                    self.mapRepo(repoName)
            else:
                print("Il repository {} non
esiste".format(repoName))

    def setBranch(self, branchName):
        """setta il branch corrente"""

        #verifico che il branch locale esista
        if (self.existsBranch(branchName)):

            #aggiorno il branch corrente
            self.currBranch = branchName
        else:
            if (self.server.existsBranch(self.currRepo,
branchName)):
                print("Il branch {} non è stato
mappato".format(branchName))
                if (uti.askQuestion("Effettuare il map del
branch?")):
                    self.mapBranch(branchName)
            else:
                print("Il branch {} non
esiste".format(branchName))

    def getLatestVersion(self):
        """scarica l'ultima versione e la copia nella cartella del
branch"""

        #prendo la latestVersion da Repository e Branch corrente
        serverDir, lastChangesetNum =
self.server.getLatestVersion(self.currRepo, self.currBranch)
        self.copyDirToClient(serverDir, self.currPath)

```



```

        uti.writeFileByTag(LAST_CHANGESET, lastChangesetNum,
self.localVersionFile)
        print("Versione locale aggiornata con successo", end="\n\n")

    def getSpecificVersion(self, changesetNum):
        """scarica una versione specifica e la copia nella cartella
del branch"""

        if (self.server.existsChangeset(self.currRepo,
self.currBranch, changesetNum) == False):
            raise Exception("Changeset non presente")

        #prendo la versione specifica da Repository e Branch corrente
con il numero di changeset passato
        serverDir = self.server.getSpecificVersion(self.currRepo,
self.currBranch, changesetNum)
        self.copyDirToClient(serverDir, self.currPath)

        uti.writeFileByTag(LAST_CHANGESET, changesetNum,
self.localVersionFile)
        print("Versione locale aggiornata con successo", end="\n\n")

    def printPendingChanges(self):
        """stampa una lista dei file modificati in locale"""

        #scorro tutti i file nella lista dei pending
        pendingList = self.getPendingChanges()
        if (len(pendingList) == 0):
            raise Exception("Nessun file in modifica.")
        else:
            print("Lista dei file in modifica:")
            for file in pendingList:
                if (pendingList[file] == EDIT) |
(pendingList[file] == OLD):
                    #prendo la data di ultima modifica del file
                    localFileDate =
datetime.datetime.fromtimestamp(path.getmtime(file)).strftime("%Y-%m-%d
%H:%M:%S")
                    #prendo la data di ultima modifica del file
sul server
                    serverFile = self.getServerFile(file)

```

```

        serverFileDate =
datetime.datetime.fromtimestamp(path.getmtime(serverFile)).strftime("%Y-
%m-%d %H:%M:%S")

        #stampo file e data ultima modifica
        print("{} ({{}) - locale: {} - server:
{}".format(file.replace(self.currPath, ""), pendingList[file],
localFileDate, serverFileDate))

        elif (pendingList[file] == ADD):
            #stampo file aggiunti in locale
            print("{}
({{}})".format(file.replace(self.currPath, ""), pendingList[file]))

        elif (pendingList[file] == REMOVED):
            #stampo file rimossi in locale
            print("{}
({{}})".format(file.replace(self.currPath, ""), pendingList[file]))

    print()

def getPendingChanges(self):
    """ritorna una lista dei file modificati in locale"""

    #scarico una latestVersion in una cartella temporanea
    tmpDir = path.join(self.currPath, TMP_DIR)

    serverDir, lastChangesetNum =
self.server.getLatestVersion(self.currRepo, self.currBranch)
    self.copyDirToClient(serverDir, tmpDir)

    #scorro tutti i file della cartella locale
    pendings = {}
    for dirPath, dirNames, files in os.walk(self.currPath):
        if (TMP_DIR in dirNames):
            dirNames.remove(TMP_DIR)
        if (COMMIT_DIR in dirNames):
            dirNames.remove(COMMIT_DIR)

        for fileName in files:
            if (fileName == LOCAL_VERSION_FILE):
                continue

            localFile = path.join(dirPath, fileName)
            try:

```

```

        #verifico se il file è fra quelli da
escludere
        if (self.isExcluded(localFile)):
            continue

        #cerco sul server il file corrispondente al
localFile
        #(cerco sempre a partire dall'ultima
versione così da segnalare anche file vecchi)
        serverFile = self.getServerFile(localFile)

        #se il file locale è stato modificato va
aggiunto ai pending
        if (int(path.getmtime(localFile)) >
int(path.getmtime(serverFile))):
            if (filecmp.cmp(localFile, serverFile)
== False):
                pendings[localFile] = EDIT
            elif(int(path.getmtime(localFile)) <
int(path.getmtime(serverFile))):
                if (filecmp.cmp(localFile, serverFile)
== False):
                    pendings[localFile] = OLD

        except:
            #se il file non viene trovato nel server
vuol dire che è stato aggiunto in locale
            pendings[localFile] = ADD

        #scorro tutti i file del server
        for dirPath, dirNames, files in os.walk(tmpDir):
            for fileName in files:
                serverFile = path.join(dirPath, fileName)
                localFile = serverFile.replace(tmpDir,
self.currPath)

                #verifico se il file è fra quelli da escludere
                if (self.isExcluded(localFile)):
                    continue

            #i file presenti nel server e non nella versione
locale sono stati rimossi
            if (path.isfile(localFile) == False):
                pendings[localFile] = REMOVED

        #ritorno la lista dei pendig
        return pendings

```

```

def isExcluded(self, file):
    """ritorna True se il file è da escludere dai pending"""

    fileName, fileExtension = path.splitext(file)

    #leggo le estensioni escluse
    try:
        excludedExt = uti.readFileByTag(EXT_IGNORE,
self.localVersionFile)
    except:
        excludedExt = ()

    #se l'estensione del file è fra quelle da escludere ritorno
True
    for ext in excludedExt:
        if (fileExtension == ext):
            return True

    #leggo la lista dei file da escludere
    try:
        excludedFiles = uti.readFileByTag(FILE_IGNORE,
self.localVersionFile)
    except:
        excludedFiles = ()

    #se il file è fra quellei da escludere ritorno True
    for exludedFile in excludedFiles:
        if (file == exludedFile):
            return True

    return False


def excludeExtension(self, ext):
    """aggiunge l'estensione "ext" alla lista delle estensioni da
escludere"""

    if (not ext.startswith(".")):
        ext = ".{}".format(ext)

    uti.writeFileByTag(EXT_IGNORE, ext, self.localVersionFile,
True)
    print("Estensione *{} esclusa.".format(ext), end="\n\n")

```

```

def includeExtension(self, ext):
    """rimuove l'esclusione sull'estensione"""

    if (not ext.startswith(".")):
        ext = ".{}".format(ext)

    uti.removeByTagAndVal(EXT_IGNORE, ext, self.localVersionFile)
    print("Estensione *{} inclusa.".format(ext), end="\n\n")

def includeAllExtension(self):
    """rimuove tutte le esclusioni su estensioni"""

    uti.removeByTag(EXT_IGNORE, self.localVersionFile)
    print("Tutte le estensioni sono state incluse.", end="\n\n")

def excludeFile(self, fileName):
    """aggiunge il file alla lista dei file da escludere"""

    file = self.findFileInPendings(fileName)
    uti.writeFileByTag(FILE_IGNORE, file, self.localVersionFile,
True)
    print("File {} escluso.".format(fileName), end="\n\n")

def includeFile(self, fileName):
    """rimuove l'esclusione sul file"""

    #file = self.findFileInPendings(fileName)
    uti.removeByTagAndVal(FILE_IGNORE, fileName,
self.localVersionFile)
    print("File {} incluso".format(fileName), end="\n\n")

def includeAllFile(self):
    """rimuove tutte le esclusioni su files"""

    uti.removeByTag(FILE_IGNORE, self.localVersionFile)
    print("Tutti i file sono stati inclusi.", end="\n\n")

def printExcluded(self):
    """stampa a video tutte le estensioni e file esclusi"""

```

```

        excludedExt = uti.readFileByTag(EXT_IGNORE,
self.localVersionFile)
        excludedFiles = uti.readFileByTag(FILE_IGNORE,
self.localVersionFile)

        if ((len(excludedExt) == 0) & (len(excludedFiles) == 0)):
            raise Exception("Nessun file o estensione esclusi")

        for ext in excludedExt:
            print("*{}".format(ext))

        for file in excludedFiles:
            print(file.replace(self.currPath, ""))
        print()

    def commitOne(self, fileName):
        """crea un nuovo changeset con le modifiche del solo file
"fileName" """

        #creo una cartella temporanea
        tmpDir = path.join(self.currPath, COMMIT_DIR)
        if (path.isdir(tmpDir)):
            shutil.rmtree(tmpDir)
        os.makedirs(tmpDir)

        #prendo il file corrente dai pending
        try:
            file = self.findFileInPendings(fileName)
            tag = self.getPendingChanges()[file]
        except:
            raise

        #aggiunto i file da committare nella cartella temporanea
        self.addForCommit(file, tag, tmpDir)

        #effettuo il commit
        print("Inserire un commento: ")
        comment = input()
        self.doCommit(tmpDir, comment)
        print("\nFile {} aggiornato con successo.".format(fileName),
end="\n\n")

    def commitAll(self):

```

```

        """crea un nuovo changeset con le modifiche dei file in
pending"""

        #creo una cartella temporanea
        tmpDir = path.join(self.currPath, COMMIT_DIR)
        if (path.isdir(tmpDir)):
            shutil.rmtree(tmpDir)
        os.makedirs(tmpDir)

        #scorro tutti i file nella lista dei pending
        pendingList = self.getPendingChanges()
        if (len(pendingList) == 0):
            raise Exception("Nessun file in modifica.")

        #aggiungo ai pending tutti i file diversi dalla versione del
server
        for file in pendingList:
            self.addForCommit(file, pendingList[file], tmpDir)

        #effettuo il commit
        print("Inserire un commento: ")
        comment = input()
        self.doCommit(tmpDir, comment)

        print("\nModifiche inviate con successo.", end="\n\n")

        #chiedo all'utente se desidera anche aggiornare la versione
locale
        if (uti.askQuestion("Scaricare ultima versione?")):
            self.getLatestVersion()

    def addForCommit(self, file, tag, tmpDir):
        """aggiunge un file alla cartella temporanea dei file da
committare"""

        if (tag != REMOVED):
            #copio il file nella cartella temporanea (creo le
cartelle se non presenti)
            #prendo il path del file da copiare
            tmpFileDir = path.dirname(file.replace(self.currPath,
tmpDir))

            #se non esiste creo il percorso
            if (path.isdir(tmpFileDir) == False):
                os.makedirs(tmpFileDir)

```

```

        #copio il file
        shutil.copy2(file, tmpFileDir)

        #inserisco il tag nel file changeset.txt.pyV
        uti.writeFileByTag(tag,
file.replace("{}\\".format(self.currPath), ""), path.join(tmpDir,
COMMIT_FILE), True)

    def doCommit(self, sourceDir, comment):
        """effettua il commit dei file contenuti in "sourceDir" su un
nuovo changeset"""

        #creo un nuovo changeset in cui copiare la cartella
temporanea
        changesetDir = self.server.addChangeset(self.currRepo,
self.currBranch, comment)
        self.copyDirToServer(sourceDir, changesetDir)

        #rimuovo la cartella temporanea
        if (path.isdir(sourceDir)):
            shutil.rmtree(sourceDir)

        uti.writeFileByTag(LAST_CHANGESET,
self.server.getLastChangeset(self.currRepo, self.currBranch),
self.localVersionFile)

    def undoFile(self, file):
        """annulla le modifiche sul file e riporta la versione a
quella originale"""

        try:
            #prendo il file corrispondente dal server e lo
sovrascrivo al file locale
            filePath = self.findFileInPendings(file)
            serverFile = self.getServerFile(filePath)

            if (uti.askQuestion("Questo comando annullerà le
modifiche sul file {}, sei sicuro?".format(file))):
                if (path.isfile(serverFile)):
                    shutil.copy2(serverFile,
path.dirname(filePath))
                else:
                    #se il file non è presente sul server era
in add sul client, quindi va semplicemente rimosso

```



```

        os.remove(filePath)
        print("Modifiche annullate.", end="\n\n")
    except:
        raise Exception("File non presente in pending.")

    def undoAll(self):
        """annulla tutte le modifiche e riporta la versione a quella
        originale"""

        if (uti.askQuestion("Questo comando cancellerà tutti i
        pending, sei sicuro?")):
            print("Modifiche annullate.", end="\n\n")

    self.getSpecificVersion(int(uti.readFileByTag(LAST_CHANGESET,
    self.localVersionFile)[0]))

    def compare(self, localFile):
        """effettua il diff del file in pending con il file sul
        server"""

        try:
            pendingFile = self.findFileInPendings(localFile)
        except:
            raise Exception("File non presente in pending")

        if (path.isfile(pendingFile) == False):
            raise Exception("File non presente sul client")

        serverFile = self.getServerFile(pendingFile)
        if (path.isfile(serverFile) == False):
            raise Exception("File non presente sul server")

        print ("".join(uti.diff(serverFile, pendingFile)))

    def merge(self, localFile):
        """effettua il diff del file in pending con il file sul
        server con winmerge"""

        try:
            pendingFile = self.findFileInPendings(localFile)
        except:
            raise Exception("File non presente in pending")

```

```

        if (path.isfile(pendingFile) == False):
            raise Exception("File non presente sul client")

        serverFile = self.getServerFile(pendingFile)
        if (path.isfile(serverFile) == False):
            raise Exception("File non presente sul server")

        #lancio winmerge
        winMergePath = ""
        try:
            winMergePath = uti.readFileByTag(WINMERGE_PATH,
self.lastRunFile)[0]
        except:
            pass

        while (not path.isfile(winMergePath)):
            print("\nDigitare il path dell'eseguibile di WinMerge:
")
            winMergePath = input()

        uti.writeFileByTag(WINMERGE_PATH, winMergePath,
self.lastRunFile)
        os.system("start {} {} {}".format(winMergePath, pendingFile,
serverFile))

    @staticmethod
    def printHelp():
        """stampa una lista di comandi con descrizione"""

        print("\n> exit - chiude il programma",
            "> repolist - stampa la lista dei repositories
presenti sul server",
            "> branchlist - stampa una lista dei branches presenti
sul repository corrente sul server",
            "> createrepo [repoName] - crea il repository
\"repoName\" nel server",
            "> createbranch [branchName] - crea il branch
\"branchName\" nel server",
            "> removerepo [repoName] - rimuove il repository
\"repoName\" dal server",
            "> removebranch [branchName] - rimuove il branch
\"branchName\" dal server",
            "> maprepo [repoName] - mappa il repository
\"repoName\" nella macchina locale",

```

```

        "> mapbranch [branchName] - mappa il branch
\"branchName\" nella macchina locale",
        "> droprepo [repoName] - elimina il repository
\"repoName\" dalla macchina locale",
        "> dropbranch [branchName] - elimina il branch
\"branchName\" dalla macchina locale",
        "> setrepo [repoName] - imposta il repository
\"repoName\" come repository corrente",
        "> setbranch [branchName] - imposta il branch
\"branchName\" come branch corrente",
        "> currdir - stampa il percorso di esecuzione
corrente",
        "> listdir - stampa una lista di file e sottocartelle
per il progetto selezionato",
        "> history - stampa la lista dei changeset del branch
corrente",
        "> getlatest - scarica la versione più recente del
branch corrente",
        "> getspecific [changeset] - scarica la versione
specificata in \"changeset\" del branch corrente",
        "> pending - stampa una lista dei file modificati in
locale",
        "> excludeextension [ext] - esclude tutti i file
.[ext] dai file in modifica",
        "> excludefile [file] - esclude il file \"file\" dai
file in modifica",
        "> includeextension [ext] - include l'estensione se
precedentemente esclusa",
        "> includeallexensions - include tutte le estensioni
precedentemente escluse",
        "> includefile [file] - include il file se
precedentemente escluso",
        "> includeallfiles - include tutti i file
precedentemente esclusi",
        "> includeall - include tutti i file e estensioni
precedentemente esclusi",
        "> printexcluded - stampa la lista di estensioni e
file esclusi",
        "> commit [file] - effettua il commit del file
\"file\"",
        "> commitall - effettua il commit di tutti i file in
pending",
        "> undo [file] - annulla le modifiche sul file
\"file\"",
        "> undoall - annulla le modifiche su tutti i file in
pending e scarica l'ultima versione",

```

```

        "> compare [file] - effettua un confronto fra il file
        \"file\" e la versione del server",
        "> winmerge [file] - effettua un confronto fra il
        file \"file\" e la versione del server con winmerge",
        "> opendir - apre la cartella corrente",
        "> clear - pulisce il terminale",
        "> help - stampa la guida", sep="\n", end="\n\n")

```

```

def getCurrPath(self):
    """ritorna il path di esecuzione"""

```

```

    return self._currPath

```

```

def setCurrPath(self, currPath):
    """setta il path di esecuzione"""

```

```

    self._currPath = currPath

```

```

currPath = property(getCurrPath, setCurrPath)

```

```

def getCurrRepo(self):
    """ritorna il repository selezionato"""

```

```

    return self._currRepo

```

```

def setCurrRepo(self, repoName):
    """setta il repository selezionato"""

```

```

    self._currRepo = repoName
    self.currPath = path.join(self.root, repoName)

```

```

currRepo = property(getCurrRepo, setCurrRepo)

```

```

def getCurrBranch(self):
    """ritorna il branch selezionato"""

```

```

    return self._currBranch

```

```

def setCurrBranch(self, branchName):
    """setta il branch selezionato"""

```

```

        self._currBranch = branchName
        self.currPath = path.join(self.root, self.currRepo,
branchName)

currBranch = property(getCurrBranch, setCurrBranch)

def getLocalVersionFile(self):
    """ritorna il file dei pending nel branch corrente"""

    return path.join(self.getCurrPath(), LOCAL_VERSION_FILE)

localVersionFile = property(getLocalVersionFile)

def getLastRunFile(self):
    """ritorna il file dell'ultimo run"""

    return path.join(self.root, LAST_RUN_FILE)

lastRunFile = property(getLastRunFile)

"""NOTA: non ammette 2 file con lo stesso nome"""
def findFileInPendings(self, fileName):
    """cerca il file "fileName" frai pending"""

    for pendingFile in self.getPendingChanges():
        if (path.basename(pendingFile) == fileName):
            return pendingFile

    raise Exception("File non trovato in pending")

def getServerFile(self, localFile):
    """prende il file della versione locale e restituisce il file
del server contenuto nella cartella temporanea"""

    tmpDir = path.join(self.getCurrPath(), TMP_DIR)
    return localFile.replace(self.getCurrPath(), tmpDir)

### Main ###

```

```

try:
    #connetto client e server
    print("Benvenuto in pyVersioning")

    #prendo l'indirizzo del server
    while (True):
        print("Digitare l'ip del server (\"localhost\" per un server
locale):")
        host = input()
        if (host == "localhost"):
            break

        try:
            socket.inet_aton(host)
            break
        except socket.error as er:
            print("IP non valido", end="\n\n")

    #stabilisco la connessione
    print("Connessione al server...", sep="\n")
    # noinspection PyUnboundLocalVariable
    connection = rpyc.connect(host, 18812)
    print("Connessione stabilita.", end="\n\n")

    #lancio il client
    Client(connection).runMenu()

    #chiudo la connessione
    connection.close()

except Exception as ex:
    print("Si è verificato un errore: {}".format(ex), "Il programma
verrà terminato.", sep="\n", end="\n\n")

```

Server.py

```
import os
import os.path as path
import shutil
import rpyc
import uti
from consts import *
from rpyc.utils.server import ThreadedServer
from Repository import Repository

class Server(rpyc.Service):

    #attributi
    root = "C:\pyV\pyVServer"

    ### Client-Server ###

    def on_connect(self):
        print("\nClient connesso.")

    def on_disconnect(self):
        print("\nClient disconnesso.")

    ### metodi di interfaccia Client-Server rpyc ###
    class exposed_File:
        filePath = ""
        file = ""

        def exposed_open(self, filePath, mode="r"):
            self.filePath = filePath
            self.file = open(filePath, mode, errors="ignore")

        def exposed_write(self, stream):
            return self.file.write(stream)

        # noinspection PyUnusedLocal
        def exposed_read(self, stream):
            return self.file.read()

        def exposed_close(self):
            return self.file.close()

        def exposed_getmtime(self):
```

```

        return path.getmtime(self.filePath)

    def exposed_findFile(self, repoName, branchName, fileRelPath,
startChangeset=None):
        return self.findFile(repoName, branchName, fileRelPath,
startChangeset)

    def exposed_existsRepo(self, repoName):
        return self.existsRepo(repoName)

    def exposed_existsBranch(self, repoName, branchName):
        return self.getRepo(repoName).existsBranch(branchName)

    def exposed_getRepo(self, repoName):
        return self.getRepo(repoName)

    def exposed_addRepo(self, repoName):
        """crea un nuovo repository, il trunk e il changeset 0,
ritorna la directory del changeset 0"""

        self.addRepo(repoName)
        return
self.getRepo(repoName).getBranch(TRUNK).getChangeset(0).changesetDir

    def exposed_removeRepo(self, repoName):
        self.removeRepo(repoName)

    def exposed_addBranch(self, repoName, branchName):
        """crea un nuovo branch"""

        self.getRepo(repoName).addBranch(branchName)

    def exposed_removeBranch(self, repoName, branchName):
        """rimuove il branch"""

        self.getRepo(repoName).removeBranch(branchName)

```



```

def exposed_addChangeset(self, repoName, branchName, comment):
    """aggiunge un changeset al branch "branchName" """

    return
self.getRepo(repoName).getBranch(branchName).addChangeset(comment).changesetDir

def exposed_existsChangeset(self, repoName, branchName,
changesetNum):
    """ritorna True se esiste il changeset "changesetNum" nel
branch "branchName" """

    try:

self.getRepo(repoName).getBranch(branchName).getChangeset(changesetNum)
        return True
    except:
        return False

# noinspection PyMethodMayBeStatic
def exposed_listDir(self, dirPath):
    return uti.listDir(dirPath)

def exposed_listBranch(self, repoName, branchName):
    """ritorna tutti i file e sottocartelle del branch
selezionato"""

    #creo una versione completa in una cartella temporanea
    tmpDir =
self.getRepo(repoName).getBranch(branchName).getLatestVersion()

    #memorizzo una lista dei file nella versione
    filesList = self.exposed_listDir(tmpDir)

    #rimuovo la cartella temporanea
    shutil.rmtree(tmpDir)

    #formatto i path dei file per la stampa
    return [elem.replace(tmpDir,"") for elem in filesList]

def exposed_showRepos(self):
    """ritorna la lista di repositories sul server"""

```

```

        return self.getRepoList()

    def exposed_showBranches(self, repoName):
        """ritorna la lista di branch nel repository "repoName" """

        return self.getRepo(repoName).getBranchList()

    def exposed_showChangesets(self, repoName, branchName):
        """ritorna la lista di changeset nel branch "branchName" con
i file modificati"""

        return
self.getRepo(repoName).getBranch(branchName).getChangesetList()

    def exposed_getLatestVersion(self, repoName, branchName):
        """scarica l'ultima versione del branch "branchName" in una
cartella temporanea """
        branch = self.getRepo(repoName).getBranch(branchName)

        return branch.getLatestVersion(),
branch.getLastChangesetNum()

    def exposed_getSpecificVersion(self, repoName, branchName,
changesetNum):
        """scarica la versione aggiornata al changeset "changesetNum"
del branch "branchName" in una cartella temporanea """

        return
self.getRepo(repoName).getBranch(branchName).getSpecificVersion(changesetNum)

    def exposed_getLastChangeset(self, repoName, branchName):
        """ritorna l'ultimo changeset del branch"""
        return
self.getRepo(repoName).getBranch(branchName).getLastChangesetNum()

    def getRepoList(self):
        """ritorna la lista di repository presenti"""

        #prendo il contenuto della root

```

```

        #seleziono solo le cartelle (i repository)
        return [name for name in os.listdir(self.root)
                if path.isdir(path.join(self.root,
name))]]

    def existsRepo(self, repoName):
        """ritorna True se il repository è presente sul disco, False
altrimenti"""

        if (path.isdir(path.join(self.root, repoName))):
            return True

        return False

    def getRepo(self, repoName):
        """ritorna il repository "repoName", se non esiste solleva
un'eccezione"""

        if (self.existsRepo(repoName)):
            return Repository(path.join(self.root, repoName))

        raise Exception("Il repository non esiste.")

    def addRepo(self, repoName):
        """viene creato un nuovo repository, se il repository esiste
già viene sollevata un'eccezione"""

        #creo un oggetto repository
        repo = Repository(path.join(self.root, repoName))
        if (path.isdir(repo.repoDir)):
            raise Exception

        os.makedirs(repo.repoDir)

        #creo il trunk - se già presente sollevo un'eccezione
        repo.addBranch(TRUNK, isTrunk=True)

    def removeRepo(self, repoName):
        """rimuove un repository"""

        if (self.existsRepo(repoName)):
            shutil.rmtree(path.join(self.root, repoName))

```

```

    def findFile(self, repoName, branchName, fileRelPath,
startChangeset=None):
        """prende in input il path relativo di un file e restituisce
il path del file corrispondente sul server"""

        #prendo il branch corrente
        branch = self.getRepo(repoName).getBranch(branchName)

        #se non diversamente specificato, la ricerca parte
dall'ultimo changeset
        if (startChangeset is None):
            startChangeset = branch.getLastChangesetNum()

        #scorro tutti i changeset presenti nella cartella del branch
a partire da quello specificato
        for changesetID in range(startChangeset, -1, -1):
            changeset = branch.getChangeset(changesetID)

            try:
                #prendo il changeset precedente (più recente) e
verifico se è un changeset di backup
                prevChangeset = branch.getChangeset(changesetID +
1)

                #se il changeset precedente era un changeset di
backup e non ho ancora trovato il file
                #vuol dire che il file non è presente sul server
                if (prevChangeset.isBackup()):
                    raise Exception("File non trovato")
            except:
                pass

            #ricavo il path del file sul server
            serverFile = path.join(changeset.changesetDir,
fileRelPath)

            #se il file esiste in questo branch interrompo la
ricerca
            if (path.isfile(serverFile)):
                return serverFile

### Main ###
if __name__ == "__main__":
    print("Benvenuto su pyVersioning (Server)")

```

```
#creo la cartella di installazione se non presente
root = "C:\pyV\pyVServer"
if (path.isdir(root) == False):
    os.makedirs(root)

#avvio il servizio rpyc
print("Avvio servizio in corso...")
server = ThreadedServer(Server, port = 18812)
print("Server attivo.")
server.start()
```

Repository.py

```
import os
import os.path as path
import shutil
import distutils.dir_util as dir_util
import uti
from consts import *
from Branch import Branch

class Repository:

    repoName = ""      #nome del repository
    repoDir = ""       #path del repository

    def __init__(self, repoDir):
        self.repoName = path.basename(repoDir)
        self.repoDir = repoDir

    def existsBranch(self, branchName):
        """ritorna True se il branch è presente sul disco, False
        altrimenti"""

        if (path.isdir(path.join(self.repoDir, branchName))):
            return True

        return False

    def getBranch(self, branchName):
        """ritorna il Branch "branchName", se non esiste solleva
        un'eccezione"""

        if (self.existsBranch(branchName)):
            return Branch(path.join(self.repoDir, branchName))

        raise Exception("Il branch non esiste.")

    def getBranchList(self):
        """ritorna la lista di branch presenti"""

        branchList = {}
        #prendo il contenuto della root
        #seleziono solo le cartelle (i branch)
        for branchName in os.listdir(self.repoDir):
```

```

        if (path.isdir(path.join(self.repoDir, branchName))):
            branchList[branchName] =
uti.readFileByTag(CHANGESET0, self.getBranch(branchName).branchTxt)[0]

        return branchList

def addBranch(self, branchName, isTrunk=False):
    """crea un nuovo branch
    se il branch non è il trunk crea un branch copiando il
contenuto del trunk
    se il branch esiste già viene sollevata un'eccezione"""

    #creo il percorso del nuovo branch
    #creo il branch
    branch = Branch(path.join(self.repoDir, branchName))
    if (path.isdir(branch.branchDir)):
        raise Exception

    os.makedirs(branch.branchDir)

    #se sto creando il trunk, creo il primo changeset vuoto, il
chiamante deve riempirlo
    if (isTrunk):
        branch.addChangeset("branch created", isBackup=True)
        #se sto inserendo il primo changeset nel branch scrivo
anche il tag CHANGESET0
        uti.writeFileByTag(CHANGESET0,
str(branch.getLastChangesetNum()), branch.branchTxt)

    #se sto creando un branch, copio l'ultima versione del trunk
    else:
        try:
            #prendo il trunk
            trunk = Branch(path.join(self.repoDir, TRUNK))
            #ottengo la LatestVersion e la copio nella
cartella temporanea
            tmpDir = trunk.getLatestVersion()
            #creo il branch sul disco e ci copio il contenuto
della cartella temporanea
            changeset = branch.addChangeset("branch created",
isBackup=True)
            uti.writeFileByTag(CHANGESET0,
str(trunk.getLastChangesetNum()), branch.branchTxt)
            dir_uti.copy_tree(tmpDir, changeset.changesetDir)
        except:

```

```
raise
```

```
def removeBranch(self, branchName):  
    """rimuove il branch "branchName" """  
  
    if (self.existsBranch(branchName)):  
        if (branchName == TRUNK):  
            raise Exception("Impossibile eliminare il ramo  
\"trunk\"")  
        shutil.rmtree(path.join(self.repoDir, branchName))
```


Branch.py

```
import os
import os.path as path
import shutil
import distutils.dir_util as dir_util
import datetime
import time
import natsort
import uti
from consts import *
from Changeset import Changeset

class Branch:
    branchDir = ""          #path del branch
    branchTxt = ""          #path del file branch.txt.pyV

    def __init__(self, branchDir):

        self.branchDir = branchDir
        self.branchTxt = path.join(self.branchDir, BRANCH_FILE)

    def addChangeset(self, comment, isBackup=False):
        """crea il prossimo changeset"""

        #controllo se è necessario creare un changeset di backup (ne
viene fatto uno ogni giorno)
        #cerco l'ultimo changeset di backup
        if (isBackup == False):
            try:
                lastBackupChangeset =
self.getLastBackupChangeset(self.getLastChangesetNum())

                #prendo la data dell'ultimo changeset di backup
                dateStr = uti.readFileByTag(
DATE,
lastBackupChangeset.changesetTxt)[0]
                date = uti.getDate(dateStr)

                #prendo la data odierna
                today = datetime.date.today()
                #se è passato un giorno dall'ultimo backup ne
creo uno

                diff = abs(today - date)
                if (diff.days > 0):
```

```

                                #creo una cartella temporanea e ci copio
una ultima versione completa
                                tmpDir = self.getLatestVersion()
                                #creo un changeset di backup con l'ultima
versione
                                changeset = self.addChangeset(comment =
"backup {}".format(dateStr), isBackup=True)
                                #copio l'ultima versione nella cartella del
changeset
                                dir_util.copy_tree(tmpDir,
changeset.changesetDir)
                                except:
                                    pass

                                #creo il changeset
                                changeset = Changeset(path.join(self.branchDir,
str(self.getNextChangesetNum() )))

                                if (path.isdir(changeset.changesetDir)):
                                    raise Exception

                                os.makedirs(changeset.changesetDir)

                                #aggiorno l'ultimo changeset
                                uti.writeFileByTag(LAST_CHANGESET,
str(self.getNextChangesetNum()), self.branchTxt)

                                #scrivo il commento per il nuovo changeset
                                uti.writeFileByTag(COMMENT, comment, changeset.changesetTxt)

                                #scrivo se il changeset è un backup
                                if (isBackup):
                                    uti.writeFileByTag(IS_BACKUP, 1,
changeset.changesetTxt)
                                else:
                                    uti.writeFileByTag(IS_BACKUP, 0,
changeset.changesetTxt)

                                #scrivo data e ora di creazione
                                uti.writeFileByTag(DATE, "{}
{}".format(time.strftime("%d/%m/%Y"), time.strftime("%H:%M:%S")),
changeset.changesetTxt)

                                return changeset

```

```

def getChangeset(self, changesetNum):
    """ritorna il changeset associato al "changesetNum" """

    if (path.isdir(path.join(self.branchDir, str(changesetNum)))
== False):
        raise Exception("Changeset non presente")

    return Changeset(path.join(self.branchDir,
str(changesetNum)))

def getLastChangeset(self):
    """ritorna l'ultimo changeset del branch"""

    return self.getChangeset(self.getLastChangesetNum())

def getLastChangesetNum(self):
    """ritorna il numero del last_changeset se il brach esiste,
-1 per un nuovo branch"""

    try:
        return int(uti.readFileByTag(LAST_CHANGESET,
self.branchTxt)[0])
    except:
        return -1

def getNextChangesetNum(self):
    """ritorna il numero del last_changeset + 1"""

    return self.getLastChangesetNum() + 1

def getChangesetList(self):
    """ritorna la lista dei changesets con modifiche"""

    #prendo tutte le cartelle all'interno del branch (i
changeset)
    dirs = [ dirName for dirName in os.listdir(self.branchDir) if
((path.isdir(path.join(self.branchDir, dirName))) & (dirName.isdigit()))
]

    #ritorno una tupla di "chageset - data creazione - commento"
    results = []

```

```

        #for dir in dirs: ""
        for currDir in natsort.natsorted(dirs):
            changeset = self.getChangeset(int(currDir))
            #prendo il path della cartella del changeset
            #prendo la data di creazione del changeset
            date =
datetime.datetime.fromtimestamp(path.getctime(changeset.changesetDir)).s
trftime("%Y-%m-%d %H:%M:%S")

            try:
                #prendo il commento dal file del changeset
                comment = uti.readFileByTag(COMMENT,
changeset.changesetTxt)[0]
            except:
                comment = ""

            #prendo una lista di tutti i file modificati in questo
changeset
            changes = ""
            try:
                for file in uti.readFileByTag(EDIT,
changeset.commitTxt):
                    changes += "{} ({}).\n".format(file, EDIT)
            except:
                pass

            try:
                for file in uti.readFileByTag(OLD,
changeset.commitTxt):
                    changes += "{} ({}).\n".format(file, OLD)
            except:
                pass

            try:
                for file in uti.readFileByTag(ADD,
changeset.commitTxt):
                    changes += "{} ({}).\n".format(file, ADD)
            except:
                pass

            try:
                for file in uti.readFileByTag(REMOVED,
changeset.commitTxt):
                    changes += "{} ({}).\n".format(file,
REMOVED)
            except:

```

```

        pass

        #aggiungo il changeset e le sue statistiche
        results.append("{} - {} - {} \n{}".format(currDir,
str(date), comment, changes))

    return results

def getLatestVersion(self):
    """copia l'ultima versione completa in una cartella
temporanea"""

    #prendo l'ultimo changeset
    lastChangeset = self.getLastChangesetNum()

    #prendo la versione associata all'ultimo changeset
    destdir = self.getSpecificVersion(lastChangeset)

    return destdir

def getSpecificVersion(self, changesetNum):
    """copia la versione completa fino al changeset
"changesetNum" nella cartella branchDir/tmp"""

    #creo la cartella temporanea
    destDir = path.join(self.branchDir, TMP_DIR)
    if (path.isdir(destDir)):
        shutil.rmtree(destDir)

    #scorro tutti i changeset all'indietro fino al primo
changeset di backup
    currChangeset = self.getLastBackupChangeset(changesetNum)

    #copio tutto il changeset di backup nella cartella
provvisoria
    shutil.copytree(currChangeset.changesetDir, destDir)

    #scorro tutti i changeset successivi e copio i file presenti
nella cartella temporanea
    for changesetID in
range(int(path.basename(currChangeset.changesetDir)) + 1, changesetNum +
1):
        currChangeset = Changeset(path.join(self.branchDir,
str(changesetID)))

```

```

        #copia di tutti i file e sottocartelle contenute nella
        cartella sourceDir dentro la cartella destDir
        dir_util.copy_tree(currChangeset.changesetDir, destDir)
        #rimuove tutti i file cancellati con questo changeset
        for fileToRemove in uti.readFileByTag(REMOVED,
currChangeset.commitTxt):
            os.remove(path.join(destDir, fileToRemove))
        try:
            os.remove(path.join(destDir, CHANGESET_FILE))
            os.remove(path.join(destDir, COMMIT_FILE))
        except:
            pass

        return destDir

def getLastBackupChangeset(self, startChangeset):
    """ritorna l'ultimo changeset di backup"""

    #scorro tutti i changeset all'indietro fino al primo
changeset di backup
    for changesetID in range(startChangeset, -1, -1):
        currChangeset = Changeset(path.join(self.branchDir,
str(changesetID)))
        if (currChangeset.isBackup()):
            return currChangeset

    #se non viene trovato nessun changeset di backup alzo
un'eccezione
    raise Exception

```

Changeset.py

```
import os.path as path
import uti
from consts import *

class Changeset:

    changesetDir = "" #path del changeset
    changesetTxt = "" #path del file changeset.txt.pyV
    commitTxt = "" #path del file commit.txt.pyV

    def __init__(self, changesetDir):
        self.changesetDir = changesetDir
        self.changesetTxt = path.join(self.changesetDir,
CHANGESET_FILE)
        self.commitTxt = path.join(self.changesetDir, COMMIT_FILE)

    def isBackup(self):
        """ritorna True se il changeset è un changeset di backup"""

        try:
            #leggo il tag is_backup del file
            if (int(uti.readFileByTag(IS_BACKUP,
self.changesetTxt)[0]) == 1):
                return True
            return False
        except:
            return False
```

uti.py

```
import os
import os.path as path
import shutil
import re
import difflib
import datetime

def readFile(filePath):
    """legge l'intero file in una stringa"""

    try:
        file = open(filePath, "r", errors="ignore")
        fileStr = str(file.read())
        file.close()
    except:
        raise

    return fileStr

def readFileByTag(tag, filePath):
    """ritorna i valori del tag passato, letto da file"""

    #se ho trovato il tag lo restituisco
    try:
        return re.findall("{}=(.*)".format(tag), readFile(filePath))
    except:
        raise

def writeFile(string, filePath, append=True):
    """scrive sul file in append o in sovrascrittura"""

    #apro il file
    if (append):
        file = open(filePath, "a", errors="ignore")
    else:
        file = open(filePath, "w", errors="ignore")

    #scrivo la stringa nel file
    print(string, file=file)

    #chiudo il file
    file.close()
```



```

#trimFile(filePath) #rimossa - troppo lenta

def writeFileByTag(tag, value, filePath, add=False):
    """scrive/cambia il valore al tag passato, scritto su file"""

    #rimuovo il tag già presente
    if (add==False):
        removeByTag(tag, filePath)

    #aggiungo il tag
    writeFile("{}={}".format(tag, str(value)), filePath)

def removeByTag(tag, filePath):
    """cancella le righe contenenti il tag dal file filePath"""

    try:
        newFileStr = re.sub("{}=(.*)".format(tag), "",
readFile(filePath))
        writeFile(newFileStr, filePath, False)
    except:
        pass

def removeByTagAndVal(tag, value, filePath):
    """cancella la riga dal file "filePath" """

    try:
        newFileStr = re.sub("{}=(.*){}(.*)".format(tag, value), "",
readFile(filePath))
        writeFile(newFileStr, filePath, False)
    except:
        pass

def trimFile(filePath):
    """elimina righe vuote dal file"""

    writeFile(re.sub("\nZ", "", re.sub("\A\n", "", re.sub("(\n)+",
"\n", readFile(filePath)))), filePath)

def askAndRemoveDir(dirPath, askOverride=False):

```

```
"""chiede all'utente se rimuovere/sovrascrivere la cartella "dir"
ed eventualmente la rimuove"""
```

```
#verifico se la cartella esiste
if (path.isdir(dirPath)):

    #chiedo all'utente se procede e sovrascrivere la cartella
    while True:
        if (askOverride):
            msg = "La cartella {} è già presente,
sovrascrivere?".format(getPathForPrint(dirPath))
        else:
            msg = "Rimuovere la cartella
{}?".format(getPathForPrint(dirPath))

        if (askQuestion(msg)):
            #l'utente ha scelto di sovrascrivere
            #rimuovo la cartella
            shutil.rmtree(dirPath)
            print("Cartella rimossa:",
getPathForPrint(dirPath), end = "\n\n")
            return True
        else:
            #l'utente ha scelto di non sovrascrivere
            print("Operazione annullata", end = "\n\n")
            return False

#se la cartella non esiste non serve rimuoverla
return True
```

```
def askQuestion(question):
    """rivolge la domanda "question" all'utente, attende una risposta
    "s": ritorna True - "n": ritorna False - altrimenti ripete la
    domanda"""
```

```
while True:
    try:
        print(question, "(s/n)")
        userInput = input()
        if ((userInput == "s") | (userInput == "S")):
            return True
        elif ((userInput == "n") | (userInput == "N")):
            return False
    finally:
```

```

        print()

def listDir(dirPath):
    """ritorna tutti i file e sottocartelle della dir selezionata"""

    #creo la lista di tutti i file e sottocartelle
    elemsList = []
    for root, dirs, files in os.walk(dirPath):
        for fileName in files:
            elemsList.append(path.join(root, fileName))
        for dirName in dirs:
            elemsList.append(path.join(root, dirName))

    return elemsList

def getPathForPrint(pathToPrint):
    """formatta un path per la stampa a video"""

    return pathToPrint.replace("/", "\\")

def getDate(dateStr):
    """ritorna un oggetto data da una stringa "dd/mm/YY HH:MM:SS" """

    [day,month,year] = map(int, dateStr.split()[0].split('/'))
    return datetime.date(year, month, day)

def diff(file1, file2):
    """effettua un diff di due file"""

    return difflib.ndiff(open(file1, errors="ignore").readlines(),
open(file2, errors="ignore").readlines())

```

consts.py

```
ADD = "add"
BRANCH_FILE = "branch.txt.pyV"
CHANGESET0 = "changeset_0"
CHANGESET_FILE = "changeset.txt.pyV"
COMMENT = "comment"
COMMIT_DIR = "commit"
COMMIT_FILE = "commit.txt.pyV"
DATE = "date"
EDIT = "edit"
EXT_IGNORE = "ext_ignore"
FILE_IGNORE = "file_ignore"
IS_BACKUP = "is_backup"
LAST_BRANCH = "last_branch"
LAST_CHANGESET = "last_changeset"
LAST_REPO = "last_repo"
LAST_RUN_FILE = "lastrun.txt.pyV"
LOCAL_VERSION_FILE = "local.txt.pyV"
OLD = "old"
REMOVED = "removed"
TMP_DIR = "tmp"
TRUNK = "trunk"
WINMERGE_PATH = "winmerge"
```