Programmazione Ad Oggetti

Progetto “Cloud 2.0”

Autore: Michele Antonazzi

Matricola: 1122118

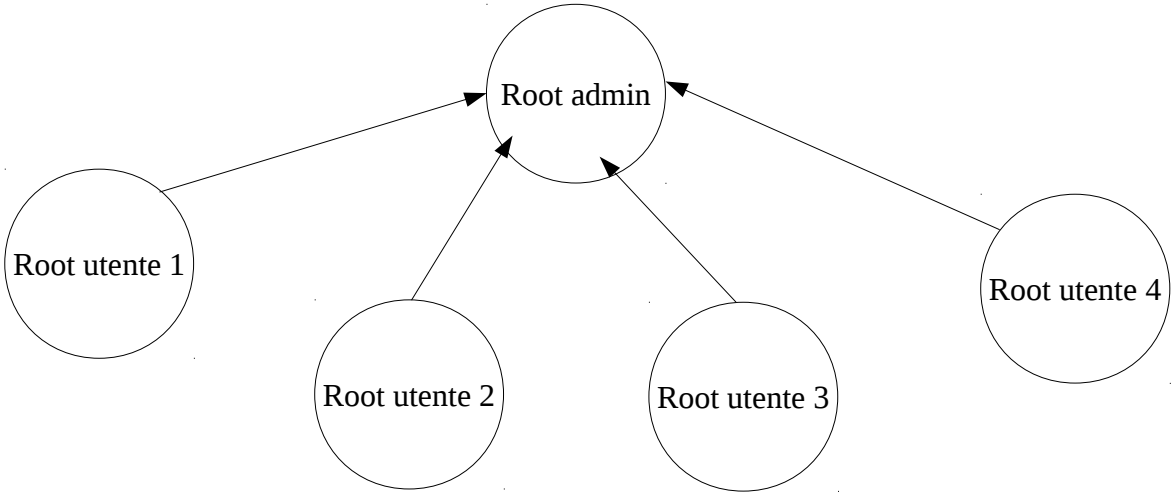
Indice generale

Scopo del progetto

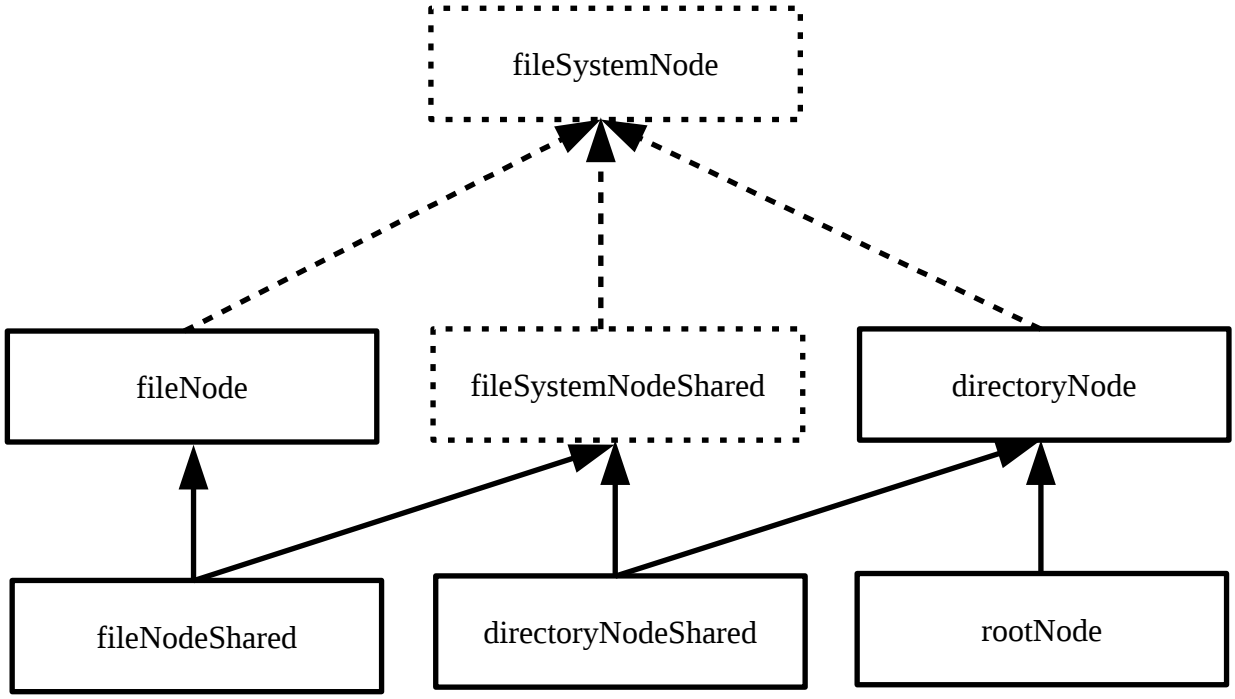
Il progetto da me realizzato vuole emulare il funzionamento di un cluod storage. Questo elaborato cerca di implementare tutte le funzioni basilari che caratterizzano un servizio cloud, tralasciandone però molte altre che avrebbero allungato il tempo di sviluppo complessivo. Questa applicazione si preoccupa di gestire gli utenti e ne prevede 3 tipi diversi: administrator, utente premium e utente standard. L'admin è unico e può accedere a tutti i file di tutti gli utenti, sui quali può compiere qualsiasi azione prevista dal programma. Un altro privilegio dell'admin è quello di poter eliminare gli utenti. I clienti che vogliono usufruire del servizio possono invece registrarsi come utenti standard o premium. A entrambe le tipologie di utenti vengono affidate delle aree protette in cui solo il proprietario e l'amministratore vi possono accedere. L'utente standard tuttavia ha meno privilegi rispetto al premium, infatti il suo spazio massimo di archiviazione è minore, può caricare file più piccoli e le funzionalità di ricerca sono ridotte. Questi vincoli possono essere annullati grazie alla funzionalità di upgrade, grazie alla quale gli utenti standard possono diventare premium. Tutti gli utenti registrati possono successivamente modificare le informazioni relative al loro account, come l'email, il nome, il cognome e la password. Se un utente non vuole più usufruire del servizio può eliminare il suo account, con la relativa perdita di tutti i suoi file. Nel momento in cui un utente entra nella sua area riservata può caricare file e organizzarli in cartelle. Su ogni elemento presente nella suo spazio di archiviazione l'utente può compiere svariate azioni, come rinominare un elemento, copiarlo e incollarlo, condividerlo con un altro utente oppure semplicemente eliminarlo. Una funzionalità aggiuntiva che l'applicazione offre è la ricerca, in cui si può inserire il nome dell'elemento da cercare (oppure un'espressione regolare), il tipo di elemento (file o cartella) e la dimensione minima e massima che l'elemento deve avere (solo per gli utenti premium).

Modello

Lo scheletro dell'intero progetto è un albero n-ario da me sviluppato con l'utilizzo di una gerarchia di classi che verrà presentata in seguito. Ogni nodo interno di questo albero è una directory, che può avere da 0 a n figli, i quali possono essere o file o altre cartelle. Le foglie invece sono i file, che non possono avere figli. Ogni nodo dell'albero ha un puntatore al padre (che deve essere necessariamente una cartella) in modo tale da poter scendere e risalire l'albero a piacimento. L'admin ha accesso alla root dell'albero, in modo tale da poter accedere e controllare i file di tutti gli utenti registrati. Al secondo livello sono presenti le cartelle di root affidate agli utenti che usufruiscono del servizio. Ad ogni utente è assegnata una root e ogni utente può accedere solo alla porzione di albero a lui assegnata (non potrà quindi risalire alla root globale e da li scendere verso sotto-alberi proprietà di altri utenti).



Gerarchia dell’albero



Il diagramma sopra riportato descrive la gerarchia di classi sulla quale si basa l’intero progetto. Attraverso esse è possibile creare un albero n-ario robusto e articolato capace di descrivere fedelmente l’organizzazione di file e cartelle all’interno di un cluod. La classe base astratta fileSystemNode racchiude in se tutte le caratteristiche e proprietà che un qualsiasi elemento deve possedere: il nome e il puntatore al padre. Ogni elemento che compone l’albero è un fileSystemNode e reimplementa i metodi

virtuali puri dichiarati nella base. Una prima estensione della classe base è fileSystemNodeShared, che introduce un campo dati contente il nome dell'utente che ha condiviso un determinato elemento. La gerarchia ora si divide in due rami principali nei quali troviamo due classi concrete, fileNode e directoryNode, che rispettivamente rappresentano file e cartelle. La principale differenza tra queste due classi è la presenza del campo privato subNodes, una std::list<fileSystemNode*> all'interno di directoryNode. Tale lista servirà a contenere i puntatori di tutti i nodi figli di una determinata cartella. Una directory, grazie a questa proprietà, potrà quindi assumere il ruolo di nodo interno e dovrà essere in grado di gestire efficacemente la porzione di albero a lei sottostante. Una proprietà aggiuntiva comune a file e cartelle è la possibilità di essere condivise. Ecco perché fileNode e directoryNode vengono estese a fileNodeShared e directoryNodeShared, entrambe ottenuta attraverso una derivazione multipla che include fileSystemNodeShared. Un'ultima classe, derivata solo da directoryNode è rootNode, che assumerà il ruolo di root per i vari utenti iscritti al sistema. Nell'albero di gestione dei file la radice è di tipo rootNode, con associato l'admin, al primo lilvello, subito sotto, sono presenti solo root ognuna della quali è affidata ad un utente.

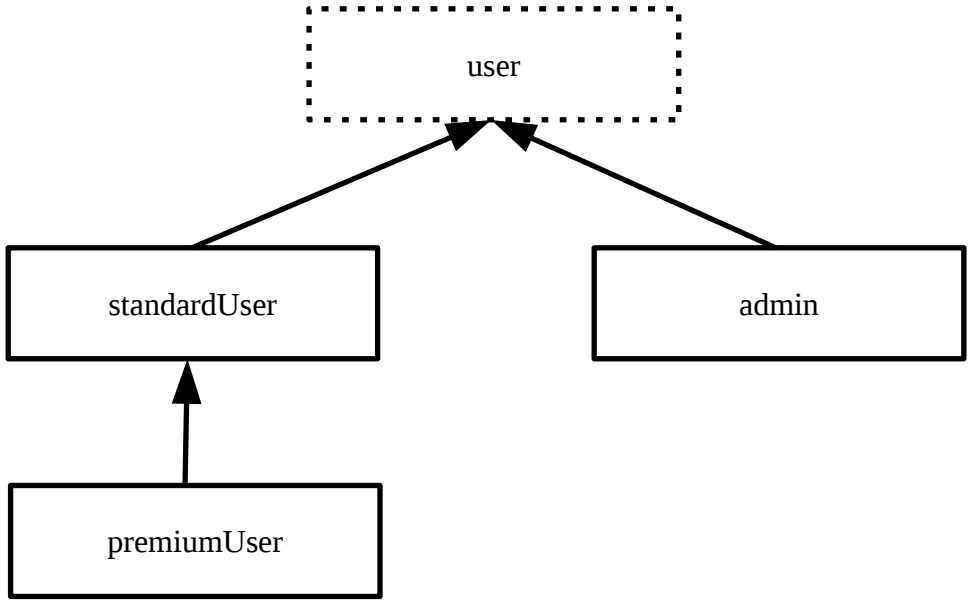
Codice polimorfo

In fileSystemNode sono presenti metodi i seguenti metodi virtuali puri:

- **fileSystemNode* clone(fileSystemNode* = 0) const** → questo metodo provvede a clonare un qualsiasi elemento che compone l'albero, sarà quindi ridefinito per ogni classe della gerarchia cambiando il tipo di ritorno con un puntatore alla classe che si sta clonando. La funzione permette anche di settare un genitore subito dopo la creazione della copia. Tuttavia se nella directory padre è già presente un nodo con lo stesso nome al nuovo elemento non verrà associato nessun genitore e il campo parent verrà lasciato a 0.
- **byteSize getSize() const** → restituisce la dimensione (espressa dalla classe byteSize) di un particolare nodo dell'albero. Questo metodo è virtuale perché la sua implementazione sarà diversa a seconda del tipo di nodo interessato. Nel caso di un file verrà restituita solamente la sua dimensione. Se invece di invoca il metodo su una directory verrà calcolata, attraverso una chiamata ricorsiva, la dimensione dell'intero sotto-albero che ha come radice la cartella stessa.
- **string getInformation() const** → questo metodo restituisce informazioni di carattere generale riguardanti un particolare elemento della struttura dati.

- **bool contains(fileSystemNode*) const** → segnala, attraverso una variabile booleana, se il nodo in questione ha nel suo sotto-albero un particolare elemento. Questo metodo, se invocato su un file, restituisce sempre false perché un file non può avere un sotto-albero sotto di lui. Nel caso di una direcotory invece controlla se nel sotto-albero radicato in essa è presente il nodo passato come parametro.
- **String getType() const** → quest'ultimo metodo virtuale puro serve a restituire il tipo dinamico di ogni puntatore, in modo tale da facilitare la scrittura su xml dell'intero albero.

Gerarchia degli utenti



Questo diagramma riflette la gerarchia di classi che compone il sistema di utenti che possono accedere al sistema. Alla base c'è la classe astratta user che contiene i campi dati che caratterizzano ciascun utente: email, password, name e lastName che servono princilmente ad effettuare il login. Altri campi dati di utente sono: un puntatore a rootNode che identifica la parte di sotto-albero affidata ad un particolare utente, un puntatore a directoryNode che indica la posizione corrente dell'utente mentre naviga tra i suoi file e infine un campo di tipo byteSize, indicante lo spazio totale di archiviazione occupato da un particolare user. Un altro importante elemento definito nella parte protetta di user è la classe searchFunctor, un funtore a cui è affidata la

ricerca di file e cartelle all'interno della struttura dati. Questa classe è particolarmente utile in quanto la ricerca è più o meno precisa a seconda dell'utente che la effettua. Un utente standard può inserire un'espressione regolare e cercare solo file, cartelle o entrambi, invece un utente premium ha la possibilità aggiuntiva di impostare un intervallo di dimensione e quindi visualizzare solo quei file o cartelle che rispettino questo vincolo. La classe user si concretizza in standardUser, la prima tipologia di utente che può iscriversi la sistema. Questo account ha però dei limiti: non può caricare file più grandi di una certa dimensione e ha spazio totale di archiviazione limitato. Questi vincoli sono espressi da due variabili statiche: maxUpload che indica la dimensione massima che il suo spazio di archiviazione può raggiungere e maxUploadFile, che limita invece la dimensione massima dei file che l'utente può caricare.

La classe derivata da standardUser è premiumUser, che estende i privilegi dell'utente. Il vincolo sulla dimensione massima dei file da caricare è eliminato e lo spazio totale di archiviazione è aumentato. L'ultimo tipo di utente è l'amministratore, descritto dalla classe admin che deriva direttamente da user. Questo utente accede ai file di tutti gli utenti e può gestirli a suo piacimento. Questo perché l'amministratore accede alla radice dell'albero e può navigarlo in tutta la sua interezza. Un altro potere dell'admin è quello di poter eliminare gli utenti.

Codice polimorfo

Nella classe base astratta user sono presenti i seguenti metodi virtuali puri:

- **user* clone() const** → questo metodo provvede a clonare un qualsiasi utente e sarà quindi ridefinito per ogni classe della gerarchia cambiando il tipo di ritorno con un puntatore consono alla classe che si sta clonando.
- **int upload(fileSystemNode*) const** → il metodo provvede a caricare un file nella posizione in cui l'utente si trova. Nella classe standardUser tale funzione deve controllare che il file che si vuole caricare non superi una determinata dimensione e lo spazio di archiviazione non oltrepassi una determinata soglia. In premiumUser invece il metodo controlla solamente che lo spazio di archiviazione abbia abbastanza spazio per il file caricato. In fine nella classe admin non sono presenti vincoli e il caricamento avviene in qualsiasi caso.
- **vector<fileSystemNode*> userSearch(const searchNode&) const** → questa funzione è adibita alla ricerca e inizializza in modo diverso la classe

searchFunctor in modo da eseguire diversamente la ricerca. Nell'utente standard esegue una ricerca parziale, escludendo la dimensione, mentre in premiumUser e admin implementa la ricerca completa.

- **string getType () const** → restituisce una stringa con il nome della classe, così da facilitare la stesura su xml.

Manuale utente

Per entrare nel sistema è possibile caricare il file backup.xml, che imposta 3 utenti:

- ranzato@gmail.com → password 12345 (admin)
- utente1@gmail.com → password 12345 (utente standard)
- utente1@gmail.com → password 12345 (utente premium)

Restore & backup

La prima schermata che compare all'apertura del progetto è un QFileDialog in cui si può scegliere il file .xml dal quale caricare i dati di utenti e file precentemente salvati. Se non viene scelto alcun file o esso è corrotto vengono impostati dei valori di default, che consistono di un utente amministratore e di un albero completamente vuoto. Alla chiusura del programma verrà aperta una schermata simile alla prima nella quale sarà possibile indicare il documento nel quale di desidera salvare i dati dell'intero sistema. Nel caso non venga selezionato alcun file di backup tutte le modifiche apportate durante la sessione andranno perse.

Schermata di login e registrazione

Una volta ripristinati i dati apparirà una schermata di login, dalla quale è possibile autenticarsi con la propria email e password oppure creare un proprio account cliccando nella label "Registrati" e inserendo i propri dati. Il programma controllerà automaticamente la validità dei valori inseriti e se la nuova email è già associata ad un utente all'interno del sistema.

Schermata amministratore e utente

La schermata amministratore e utente sono molto simili, se non identiche. A sinistra è presente un QTreeWidget contenente le cartelle e file dell'intero albero. A destra in alto è presente il widget ricerca, dove è possibile settare i parametri ed eseguire la ricerca. A destra in basse è presente un QTableWidget che contiene tutti gli elementi

presenti in una deterinata directory. All’estrema destra, in verticale, c’è una QToolBar in cui sono presenti le azioni disponibili. Descriviamo ora in breve i vari elementi della view.

QTreeWidget

Questo componente grafico contiene, in una vista ad albero, tutti gli elementi proprietari di un utente. Eseguendo un double click su una directory ci si posiziona all’interno di essa e il suo contenuto viene visualizzato nella tabella più a destra.

Widget di ricerca

Questo componente gestisce la ricerca. É possibile inserire un’espressione regolare, filtrare file, cartelle o entrambi e, solo per gli utenti premium, settare un intervallo di ricerca dimensionale. Una volta premuto il pulsante corrispondente la ricerca è avviata e i risultati di essa vengono visualizzati a video all’interno di un QDialog, che contiene una tabella simile a quella presente nella schermata principale. Con un doppio click del muose su un elemento della ricerca il QDialog si chiude e si viene portati all’interno della cartella contenente l’elemento desiderato. Anche il TreeWidget viene modificato, visualizzando il percorso di tale directory.

TableWidget

La parte centrale dell’intera vista è occupata da una tabella che indica gli elementi contenuti dalla cartella in cui ci si trova. Le informazioni visualizzate sono il tipo (espresso da un’icona), il nome, la dimensione e l’utente che li ha condivisi (se sono stati condivisi). Tutte le azioni presenti nella barra laterale sono riferite al file selezionato in tale tabella. Per questo motivo se nella tabella non è presente una riga selezionata alcune delle azione della toolBar vengono disabilitate. Una funzionalità aggiuntiva offerta da questa tabella è la possibilità di aprire, con il tasto destro del mouse, un piccolo menu che ci elencherà le stesse azioni presenti nella toolBar, sempre applicabili all’elemento selezionato. In questo menù è aggiunta una voce chiamata “informazioni” che, se premuta, farà comparire un popup contente le informazione relative alla riga selezionata. Nel caso l’admin sia loggato vedrà, al livello superiore, le root di tutti gli utenti iscritti al sistema. Selezionando l’azione “elimina” su una di queste cartelle (caratterizzate da un’icona diversa) il programma eliminerà oltre alla root anche l’utente che a cui è affidata.

ToolBar

L’ultimo elemento grafico ad essere descritto è la toolBar, contenuta nella parte destra della GUI. Tale componente è suddiviso in due parti, nella sezione più in alto sono contenute tutte le azioni atte a gestire i file all’interno del proprio spazio di archiviazione; nella parte in basso compaiono invece le azioni per amministrare il proprio account. Un compontente degno degno di essere trattato è il QDialog per modificare i dati del proprio account. Alla sua apertura i campi che caratterizzano il proprio profilo vengono riempiti con i valori attuali, per modificarli e sufficiente cambiare il loro valore e inserire la password per confermare la propria identità. Se non si intende modificare la password basta lasciare vuote le ultime due textLine.

Ore impiegate

- Progettazione: 4h
- Scrittura modello: 16h
- Progettazione GUI: 5h
- Scrittura GUI: 38h
- Test e dedug: 10h (distribuite durante la stesura del codice)

Le ore complessive utilizzate sforano le 60 consigliate dal professore. La prima motivazione è stata la difficoltà nel separare modello e interfaccia grafica. Il motivo deriva dal fatto che la struttura ad albero è gestita utilizzando puntatori e non sono riuscito a sviluppare un controller efficente che mantenesse una netta separazione tra modello dati e GUI. Per risolvere il problema ho adottato una via di mezzo, ho creato la classe cloudModel che raccogliesse al suo interno l’albero e la lista di utenti. Questa classe funge anche da controller, infatti ho impedito alla GUI di adottare qualsiasi decisione o compiere modifica al contenitore o agli utenti registrati, compiti che spettano invece a cloudModel che li assolve attuando gli opportuni controlli. Dico che non sono riuscito a mantenere una netta separazione tra GUI e modello in quanto ho inserito all’interno dell’interfaccia grafica i puntatori a cartelle e file, in modo tale da semplificare una GUI che altrimenti sarebbe diventata molto complessa. Sottolineo il fatto che qualsiasi azione di modifica al contenitore o agli utenti è demandata alla classe cluodModel, che assolve il compito e restituisce l’output oppure solleva una eccezione che verrà visualizzata a video.

Un secondo problema che ha rallentato lo sviluppo è stato riscontrato dopo la stesura del modello. Testando il codice da me scritto ho notato un metodo “pericoloso” di directoryNode: openDirectory. Questo metodo restituisce in output la lista dei puntatori ai nodi figli di una cartella. Questo però infrange l’incapsulamento che la parte privata di ogni classe dovrebbe avere, infatti ritornando la lista in output è come se quel campo privato diventasse pubblico. A fronte di ciò ho valutato diverse soluzioni, come quella di usare puntatori smart o effettuare una copia parziale del sotto-albero radicato in una cartella. Alla fine, trovando troppo complicati i puntatori smart e troppo onerosa la copia parziale, ho adottato una soluzione differente: rendere un’azione potenzialmente dannosa innocua. L’azione pericolosa che può invalidare il modello dati e la delete su un sotto-nodo, perché il puntatore non verrebbe rimosso dalla lista di figli del padre. Per evitare questo problema ho modificato il distruttore di ogni classe della gerarchia, così quando un nodo viene distrutto si accede al padre (se ne ha uno) e si rimuove il puntatore al quel nodo che da lì a poco verrà deallocato. Questo approccio è usato anche da Qt.