

Programare orientată pe obiecte

Tema - MyCloud

Deadline: 4.01.2016

Ora 23:55

Responsabili tema: *Alexandru Rotaru, Adrian Tănase, Mihai Nan*

Profesor titular: *Carmen Odubășteanu*



Facultatea de Automatică și Calculatoare
Universitatea Politehnica din București
Anul universitar 2015 - 2016
Seria CC

1 Obiective

În urma realizării acestei teme, studentul va fi capabil:

- să aplice corect principiile programării orientate pe obiecte studiate în cadrul cursului;
- să construiască o ierarhie de clase, pornind de la un scenariu propus;
- să utilizeze un design orientat-obiect;
- să realizeze o serie de teste pentru a verifica funcționalitatea aplicației;
- să realizeze o interfață grafică, folosind componente de tip Swing, pentru o aplicație;
- să înțeleagă principiul minimal de funcționare al unui sistem de backup în cloud;
- să dezvolte concepte de parsare lexicală și semantică;
- să trateze excepțiile ce pot interveni în timpul rularii unei aplicații;
- să transpună o problemă din viața reală într-o aplicație ce se poate realiza folosind noțiunile dobândite în cadrul cursului.

2 Descriere

Scopul acestei teme este de a simula operațiile pe care le face un sistem de backup în cloud, asemănător cu serviciul oferit de [Dropbox](#), [Microsoft OneDrive](#) sau [Google Drive](#). Aceasta simulare nu va dispune de resurse fizice (nu o să faceți o aplicație distribuită pe mai multe calculatoare fizice), fiecare entitate necesară (aici într-un server sau stațiile de stocare) în program va fi reprezentată de un obiect.

Serviciul vostru va avea în spate trei stații pentru stocare, un server central care deține informațiile curente despre fiecare stație, și, bineînțeles, stația curentă, cea de pe care operează clientul. Clientul nu este decât o persoană care deschide programul vostru, verifică anumite fișiere, adaugă sau șterge fișiere și, când consideră că e cazul, își va uploada fișierele în cloud. În momentul în care utilizatorul decide să-și uploadeze fișierele, sistemul vostru de cloud va încerca să le stocheze, dacă găsește loc, dacă nu, vă va anunța că nu are suficient spațiu.

3 Arhitectura aplicației

3.1 Sistem de fișiere, utilizatori și comenzi

Pentru a putea să vă jucați și să simulați un serviciu de cloud ce oferă suport pentru stocare de fișiere, va trebui mai întâi să reprezentați acest sistem de fișiere în memorie. Pentru început, trebuie să oferiți suport pentru crearea și modificarea acestui sistem de fișiere, comenzile obligatorii pe care va trebui să le implementați fiind: `ls`, `mkdir`, `cd`, `rm`, `touch`, `cat`, `echo`, `pwd`, `upload` și `sync`.

Observație

Fiecare fișier și director va avea un set clar de permisiuni care în cazul directorilor se va propaga și în ierarhia copil, fiind din ce în ce mai restrictivă (dacă un director nu-i va permite unui utilizator anumite operații atunci nici fișierele copil sau subdirectoarele nu vor permite acest lucru).

De asemenea, trebuie să faceți gestiunea utilizatorilor, aceștia putându-se loga sau deloga în sistem. Comenzile de gestiune a utilizatorilor sunt **newuser** → creează un nou utilizator, **login** → autentifică un utilizator, **logout** → ieșire din sistem, **userinfo** → informațiile despre anumiți utilizatori. Pentru fiecare user va trebui obligatoriu să salvați **username**, **parola**, **numele** și **prenumele**, **data** și **ora la care a fost creat** și **data și ora ultimei logări**.

Observatie

1. Vor exista doi utilizatori speciali: **root** care va avea toate drepturile si **guest** care nu va avea niciun drept. In momentul in care un user iese din sistem, utilizatorul de la tastatura va fi automat logat ca **guest** pana cand se autentifica din nou.

2. La un moment dat, **NU** poate fi decat un singur user logat. Pentru a va asigura faptul ca nu puteti avea mai multi utilizatori logati in sistem, veti crea o clasa de management (o clasa care va sti userul curent logat) care sa nu poate fi instantiata decat o singura data.

NU vor exista directoare/fisiere cu acelasi nume.



Singleton Pattern

Cod sursa Java

```
1 public class MySingleton {
2     private static final MySingleton INSTANCE = new MySingleton();
3
4     private MySingleton() {
5         //Constructor ce se poate apela doar din clasa MySingleton
6     }
7
8     public static final MySingleton getInstance() {
9         return INSTANCE;
10    }
11 }
```

Sistemul de fisiere va contine fisiere si directoare, organizate ierarhic. Sarcina voastra este sa va ganditi la modalitate de a crea o ierarhie arborescenta in care fisierele sunt frunze iar directoarele pot fi atat noduri cat si frunze. Un director poate avea mai multi fii, atat alte directoare cat si fisiere, in schimb, un fisier nu poate avea noduri copii.



Composite Pattern

Veti porni de la urmatorul snippet, avand posibilitatea de a adauga si alte metode in interfata dar respectati "interface segregation principle". Fiecare element din ierarhie va implementa interfata data.

Cod sursa Java

```
1 Interface Repository {
2     public void accept(Command c);
3 }
```

Fiecare **fisier** va trebui sa contina obligatoriu **numele**, **dimensiunea**, **tip (binar/text)** si un **array de octeti** ce reprezinta datele continute, **ora la care a fost creat** si **permisiunile**. Fiecare **director** va trebui sa detina un **container cu subdirectoarele si fisierele copil**, impreuna cu permisiunile, **data la care a fost creat** si **dimensiunea totala a fisierelor si subdirectoarelor**.

Permisiunile vor avea umatoarea forma: un membru de tip boolean pentru **dreptul de read**, alt membru, tot de tip boolean pentru **dreptul de write** si **utilizatorul** care detine aceste drepturi.

Comenzile (sau operatiile din sistem) vor trebui sa aiba o interfata uniforma gandita astfel incat sa permita si adaugarea, ulterioara, a altor comenzi si vor trebui sa tina cont de permisiunile utilizatorului curent. Acestea vor avea urmatoarele tipuri: comenzi de scriere in sistem (**mkdir**, **touch**, **rm**), de citire din sistem (**ls**, **cd**, **cat**, **pwd**, **upload**) si independente de sistem (**echo**, **newuser**, **userinfo**, **logout**, **login**).

Observatie

Singurele comezi pe care le va putea scrie un user **guest** sunt cele independente de sistem.

Observatie

In momentul in care apelati execute veti verifica daca utilizatorul are dreptul de a scrie comanda respectiva in contextul dat ca parametru. (contextul poate fi director sau fisier)

Comanda **ls** va lista elementele din interior, daca primeste parametru un director, si va afisa informatiile aferente in cazul unui fisier. La parametrul **-r** se va reapela recursiv pe toata ierarhia de fisier din directorul dat. Parametrii pot fi si expresii regulate, iar in acest caz se vor afisa doar informatiile directoarelor sau fisierelor al caror nume face match pe expresia regulata. La parametrul **-a** se vor afisa detaliile(data crearii, dimensiunea) fisierului sau directorului.

Observatie

Expresiile regulate vor fi Java-Based.

Comanda **cd** va schimba directorul curent, in cazul in care path-ul primit ca parametru este invalid se va arunca o exceptie de tip **MyInvalidPathException** pe care va trebui s-o creati voi si care va detine informatiile despre directorul in care s-a dat comanda, parametrul, utilizatorul si data si ora la care s-a intamplat.

Comanda **pwd** va afisa path-ul absolut al directorului curent. Daca calea absoluta in directorul curent va depasi 255 de caractere, se va arunca o exceptie e tipul **MyPathTooLongException** care va fi derivata din **MyInvalidPathException**.

Comanda **mkdir** va crea un director, cu numele dat ca parametru, in directorul curent.

Comanda **touch** va crea, in directorul curent, un fisier gol cu numele dat ca parametru.

Observatie

Managementul si operatiile pe continutul fisierelor(octetii de date) in sine nu sunt un aspect pe care se pune accent. Optional puteti sa va faceti un generator random de continut care va genera ori text ori octeti cu care sa umpleti fisierele.

Comanda **rm** va sterge fisierul sau un director gol dat ca parametru. In cazul in care se doreste stergerea unui director cu tot cu subdirectoare sale, se va adauga parametrul **-r**. Aceasta comanda va putea primi ca parametru si expresii regulate, in acest caz, se va aplica doar pe entitatile a caror nume va face match pe expresia regulata. Daca se primeste un parametru invalid (ex: nu exista fisierul), se va arunca o exceptie de tipul **MyInvalidPathException**.

Comanda **upload** va incarca, in serviciul vostru de cloud, directorul sau fisierul, al carui nume a fost dat ca parametru. Pentru directoare se va uploada intreaga ierarhie din acel director. In cazul in care operatia nu poate avea loc (ex: nu mai este suficient spatiu in cloud) se va arunca exceptia **MyNotEnoughSpaceException** care va cuprinde urmatoarele informatii: dimensiunea ierarhiei ce s-a vrut uploadata, userul care a data comanda, ora si data la care s-a dat comanda.

Comanda **sync** va *sterge* directorul curent si il va descarca din cloud pentru a reveni la sistemul de fisiere initial.

Toate comenzile vor fi definite ca obiecte (pentru fiecare comanda faceti o clasa separata e.g: "CommandLs"). Acestea obligatoriu vor implementata metoda **execute**, ce va primi ca parametru directorul curent sau, dupa caz, fisierul curent.



Pentru a crea comenzile folositi Factory Pattern

Toate exceptiile care se arunca vor fi inregistrate intr-un obiect de tip **Logger**, iar, la oprirea programului, acestea se vor salva intr-un fisier. Loggerul va mai tine si evidenta logarilor si delogarilor din sistem. Practic va

fi un **observator** pentru eventimente legate de useri si exceptii.



Observer Pattern

Observatie

Unele comezi depind de tipul instantei (**ls** are un comportament diferit pe un director fata de un fisier). Cum interfața **Command** dispune de metoda **execute(Repository r)**, orice apel pe un element al ierarhiei va apela codul metodei cu semnatura precizata anterior. De aceea, pentru a evita hardcode folosind operatorul **instanceof** trebuie sa valorificati urmatorul workaround.

Cod sursa Java

```
1 //In clasa ce implementeaza interfata Command (urmatoarele 3 metode)
2 void execute(Repository rep) {
3     rep.accept(this);
4 }
5
6 void execute(Directory dir) {
7     //whatever
8 }
9
10 void execute(File file) {
11     //whatever
12 }
13
14 //In clasa ce implementeaza interfata Repository (urmatoarea metoda)
15 void accept(Command com) {
16     com.execute(this);
17 }
```

Observatie

In acest mod cand apelati metoda execute pe un obiect cu tipul static **Repository** se va executa codul aferent tipului dinamic al obiectului primit ca parametru.

3.2 Serviciul de cloud, statii si distribuire

Serviciul de cloud va fi abstractizat in modul urmator: veti avea o clasa **CloudService** cu urmatoarele metode: **upload(Directory)** si **sync(Directory)**, prima metoda va incarca in *cloud* directorul dat ca parametru, cea de-a doua metoda va avea rolul de a sterge modificarile facute in folderul dat ca parametru reconstruind ierarhia din informatiile stocate in *cloud* (aceasta pentru cazul in care utilizatorul isi sterge din greseala continutul unui director si vrea sa-si recapete datele).

In momentul in care se incearca uploadarea unui director, serverul verifica daca exista loc suficient pe cele 3 statii, in caz negativ, metoda **upload** va arunca exceptia precizata mai sus. Practic, obiectele de tip **CloudService** vor oferi implementarea, in cazul obiectelor de tip comanda de **upload** sau **sync**.

In momentul in care, pe o statie, nu exista suficient loc pentru a stoca intreaga ierarhie se va proceda in felul urmator: se pun intr-o structura de date liniara (vector, lista) nodurile rezultate in urma parcurgerii in adancime a arborelui. Se parcurg iterativ nodurile si se verifica daca pe statia de stocare curenta mai este suficient spatiu. Daca nu s-a depasit limita de stocare, *nodul* este stocat, iar daca s-a ajuns la limita elementele ramase vor fi date spre stocare unei alte statii.

Observatie

In momentul in care copiat pe o statie va trebui sa simulati faptul ca statiile nu au memorie partajata. Va trebui sa clonati elementele inainte de a le stoca.

În momentul în care scindati ierarhia, va trebui să țineți într-un fel evidența stațiilor/statiei pe care se află restul nodurilor. Puteți face acest lucru înlocuind referințele părinte/copil ale unui nod cu id-ul statiei pe care se află restul ierarhiei. Trebuie să vă asigurați că obiectul de tip **MachineId** este interschimbabil cu un obiect din ierarhia de fișiere.



O să faceți un trade-off și o să încalcați "interface segregation principle". **MachineId** să deriveze **Repository** fără a pune logică în metodele impuse de aceasta (**MachineId** este un adaptor pentru **Repository**).

Fiecare unitate de stocare va moșteni o clasă abstractă de bază numită **StoreStation** ce va conține un obiect de tipul **MachineId**.

Observație

Stațiile de stocare vor putea stoca maximum 10 KB.

StoreStation va acționa ca un container și va oferi o metodă pentru **store**, și o metodă pentru căutarea unui anumit entry stocat. Mașinile își vor stoca datele primite într-un **HashSet**.



În sistem poate exista un singur serviciu de cloud!

3.3 Interfața grafică

Va trebui să creați o interfață, folosind componente Swing, care să semene cât mai mult cu modul în care arată un terminal din lumea unix. Acesta va trebui să poată permite utilizatorului să introducă comenzi, dar și să deruleze prin outputul comenzilor anterioare.

3.3.1 Output formatat

Spre deosebire de terminalele unix, terminalul vostru va afișa, în cazul în care se tastează o comandă cu parametrul **-POO**, output-ul într-un mod formatat.

Exemplu: Pentru comanda **ls -l -POO**, output-ul o să apară poziționat în fereastră ca într-un terminal normal dar nu va fi plain text ci încadrat într-un **JTable**, restul outputului, împreună cu obiectul de tip **JTable**, vor trebui să fie în continuare scrollable.

Comenziile care vor fi sensibile la parametrul **-POO** vor avea output-ul afișat pe ecran folosind o componentă de tipul: **ls** → **JTable**, **userinfo** → **JList**, **echo** → **JDialog**.



Userul poate modifica doar comanda curentă când scrie, rezultatele comenzilor anterioare vor fi read-only.

3.4 Bonusuri

3.4.1 Autocomplete

Dacă utilizatorul introduce comanda **cd** sau **rm**, în momentul în care specifică numele sau calea fișierului, terminalul va trebui să permită posibilitatea autocompletării (asemănător cu autocomplet-ul din IDE-urile în care scrieți cod Java). Cum utilizatorul scrie mai multe litere din parametru sau cale, autocompletarea va trebui să reducă dinamic (în timp real) numărul de posibilități. Utilizatorul va putea să selecteze un entry din lista afișată de autocomplete cu mouse-ul sau folosind combinația săgeți + Enter.

3.4.2 Test Driven Development (TDD)

Realizați o suită de teste (pentru fiecare comandă, pentru a testa încărcarea corectă a fișierului cu ierarhia de fișiere, s.a.m.d.) pe măsura ce scrieți cod, dar înainte de a implementa o nouă funcționalitate, pentru a vă

asigura faptul ca noile functionalitati sunt corecte si pentru a depista din timp posibile erori care propagandu-se sa va dea mari batai de cap.



JUnit Testing

3.4.3 Look and feel

In arhiva auxiliara aveti un **.jar** ce contine un look and feel. Sarcina voastra este de a introduce look and feel-ul acesta in aplicatia voastra pentru ca aceasta sa aiba un aspect placut si modern.

4 Punctaj

Cerinta	Punctaj
Cerinta 1	80 puncte
Cerinta 2	70 puncte
Cerinta 3	50 puncte
Bonus	50 puncte

Atentie!

Solutii de genul folosirii de variabile interne pentru a stoca informatii ce tin de tipul obiectului vor fi depunctate.

Tipizati orice colectie folosita in cadrul implementarii.

Nu este obligatoriu sa folositi exact aceleasi nume pentru metode, interfete si clase ca in cerinta dar trebuie sa aveti grija ca acestea sa fie sugestive si inteligibile. Insa, daca alegeti sa schimbati aceste nume, va trebui sa modificati si clasele puse la dispozitie pentru testare.

Respectati specificatiile detaliate in enuntul temei si folositi hinturile mentionate.

Tema este individuala! Toate solutiile trimise vor fi verificate, folosind o unealta pentru detectarea plagiatului.

Tema se va prezenta la ultimul laborator din semestru.

Tema se va incarca pe site-ul de cursuri pana la termenul specificat in pagina de titlu. Se va trimite o arhiva **.zip** ce va avea un nume de forma **grupa_Nume_Prenume.zip** (ex. **326CC_Popescu_Andreea.zip** si care va contine urmatoarele:

- un folder **SURSE** ce contine doar sursele Java si fisierele de test;
- un folder **PROIECT** ce contine proiectul *NetBeans* sau *Eclipse*;
- un fisier **README** in care veti specifica numele, grupa, gradul de dificultate al temei, timpul alocat rezolvarii si veti detalia modul de implementare, specificand si alte observatii, daca este cazul. Lipsa acestui fisier duce la o depunctare de **10 puncte**.