# Programare orientată pe obiecte

## Tema - MyCloud

**Deadline: 4.01.2016**
**Ora 23:55**

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

**Profesor titular:** *Carmen Odubăşteanu*

**Facultatea de Automatică şi Calculatoare**
**Universitatea Politehnica din Bucureşti**
**Anul universitar 2015 - 2016**
**Seria CC**

# 1 Edit

- clarificare la ls fara parametru
- explicatii cat
- schimbare la sync si upload, daca deja ati facut o implementare diferita, va fi acceptata

# 2 Obiective

In urma realizarii acestei teme, studentul va fi capabil:

- sa aplice corect principiile programarii orientate pe obiecte studiate in cadrul cursului;
- sa construiasca o ierarhie de clase, pornind de la un scenariu propus;
- sa utilizeze un design orientat-obiect;
- sa realizeze o serie de teste pentru a verifica functionalitatea aplicatiei;
- sa realizeze o interfata grafica, folosind componente de tip Swing, pentru o aplicatie;
- sa inteleaga principiul minimal de functionare al unui sistem de backup in cloud;
- sa dezvolte concepte de parsare lexicala si semantica;
- sa trateze exceptiile ce pot interveni in timpul rularii unei aplicatii;
- sa transpuna o problema din viata reala intr-o aplicatie ce se poate realiza folosind notiunile dobandite in cadrul cursului.

# 3 Descriere

Scopul acestei teme este de a simula operatiile pe care le face un sistem de backup in cloud, asemanator cu serviciul oferit de Dropbox, Microsoft OneDrive sau Google Drive. Aceasta simulare nu va dispune de resurse fizice (nu o sa faceti o aplicatie distribuita pe mai multe calculatoare fizice), fiecare entititate necesara (aici intra serverul sau statiile de stocare) in program va fi reprezentata de un obiect.

Seriviciul vostru va avea în spate trei staţii pentru stocare, un server central care deţine informaţiile curente despre fiecare statie, şi, bineînteles, staţia curentă, cea de pe care operează clientul. Clientul nu este decât o persoană care deschide programul vostru, verifică anumite fişiere, adaugă sau şterge fişiere şi, când consideră că e cazul, îşi va uploada fişierele în cloud. In momentul în care utilizatorul decide să-şi uploadeze fişierele, sistemul vostru de cloud va încerca să le stocheze, dacă găseşte loc, dacă nu, vă va anunţa că nu are suficient spaţiu.

# 4 Arhitectura aplicatiei

## 4.1 Sistem de fişiere, utilizatori şi comenzi

Pentru a putea sa va jucati si sa simulati un serviciu de cloud ce ofera suport pentru stocare de fisiere, va trebui mai intai sa reprezentati acest sistem de fisiere in memorie. Pentru inceput, trebuie sa oferiti suport pentru crearea si modificarea acestui sistem de fisiere, comenzile obligatorii pe care va trebui sa le implementati fiind: **ls**, **mkdir**, **cd**, **rm**, **touch**, **cat**, **echo**, **pwd**, **upload** si **sync**.

**Observatie**

Fiecare fisier si director va avea un set clar de permisiuni care in cazul directoarelor se va propaga si in ierarhia copil, fiind din ce in ce mai restrictiva (daca un director nu-i va permite unui utilizator anumite operatii atunci nici fisierele copil sau subdirectoarele nu vor permite acest lucru).

De asemenea, trebuie sa faceti gestiunea utilizatorilor, acestia putandu-se loga sau deloga in sistem. Comenzile de gestiune a utilizatorilor sunt **newuser** → creeaza un nou utilizator, **login** → autentifica un utilizator, **logout** → iesire din sistem, **userinfo** → informatiile despre anumiti utilizatori. Pentru fiecare user va trebui obligatoriu sa salvati **username**, **parola**, **numele** si **prenumele**, **data si ora la care a fost creat** si **data si ora ultimei logari**.

**Observatie**

1. Vor exista doi useri 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**

```java
public class MySingleton {
    private static final MySingleton INSTANCE = new MySingleton();

    private MySingleton() {
        //Constructor ce se poate apela doar din clasa MySingleton
    }

    public static final MySingleton getInstance() {
        return INSTANCE;
    }
}
```

Sistemul de fisiere va contine fisiere si directoare, organizate ierarhic. Sarcina voastra este sa va ganditi la modalitate de a creea 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**

```java
Interface Repository {
    public void accept(Command c);
}
```

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 numele 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 **cat** va afisa continutul fisierului dat ca parametru.

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 ca parametru. 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 dat ca parametru si il va descarca din cloud pentru a reveni la sistemul de fisiere initial.

Atentie: Comanda sync nu va functiona daca directorul din cloud a fost sters local, deoarece nu o sa aveti ce parametru sa-i dati comenzii.

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, aceastea 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 interfata ***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**

```
//In interfata Command (urmatoarele 3 metode)
void execute(Repository rep) {
    rep.accept(this);
}

void execute(Directory dir) {
    //whatever
}

void execute(File file) {
    //whatever
}

//In clasa ce implementeaza interfata Repository (urmatoarea metoda)
void accept(Command com) {
    com.execute(this);
}
```

**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.

## 4.2 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).

Inainte de orice alta verificare se vor clona elementele din ierarhie data. In procesul de clonare, la fiecare pas se verifica daca fiul curent din directorul ce se cloneaza exista undeva pe cloud. Daca acesta exista, atunci inlocuiti fiul respectiv cu MachineID-ul statiei pe care nodul fiu se afla deja stocat. Modul in care stocati acest nod astfel incat sa puteti reconstrui arborele este la latitudinea voastra(o solutie ar fi sa stocati in MachineID numele fiului).

In momentul in care se incearca uploadarea unui director, serverul verifica daca exista loc suficient pe cele 3 statii dar si daca exista acel director sau unul din subdirectoare pe cloud. Daca nu mai exista loc 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 copiati pe o statie va trebui sa simulati faptul ca statiile nu au memorie partajata. Va trebui sa clonati elementele inainte de a le stoca.

In momentul in care scindati ierarhia, va trebui sa tineti intr-un fel evidenta statiilor/statiei pe care se afla restul nodurilor. Puteti face acest lucru inlocuind referintele parinte/copil ale unui nod cu id-ul statiei pe care se afla restul ierarhiei. Trebuie sa va asigurati ca obiectul de tip ***MachineId*** este interschimbabil cu un obiect din ierarhia de fisiere.

O sa faceti un trade-off si o sa incalcati "interface segregation principle". MachineId sa deriveze Repository fara a pune logica in metodele impuse de aceasta (MachineId este un adaptor pentru Repository).

Fiecare unitate de stocare va mosteni o clasa abstracta de baza numita ***StoreStation*** ce va contine un obiect de tipul ***MachineId***.

**Observatie**

**Statiile de stocare** vor putea stoca maximum 10 KB.

***StoreStation*** va actiona ca un container si va oferi o metoda pentru **store**, si o metoda pentru cautarea unui anumit entry stocat. Masinile isi vor stoca datele primite intr-un ***HashSet***.

In sistem poate exista un singur serviciu de cloud!

## 4.3 Interfata grafica

Va trebui sa creati o interfata, folosind componente Swing, care sa semene cat mai mult cu modul in care arata un terminal din lumea unix. Acesta va trebui sa poata permite utilizatorului sa introduca comenzi, dar si sa deruleze prin outputul comenzilor anterioare.

### 4.3.1 Output formatat

Spre deosebire de terminalele unix , terminalul vostru va afisa, in cazul in care se tasteaza o comanda cu paramtrul **-POO**, output-ul intr-un mod formatat.

**Exemplu:** Pentru comanda **ls -a -POO**, outputul o sa apara pozitionat in fereastra ca intr-un terminal normal dar nu va fi plain text ci incadrat intr-un ***JTable***, restul outputului, impreuna cu obiectul de tip ***JTable***, vor trebui sa fie in continuare scrollable.

Comenziile care vor fi sensibile la parametrul **-POO** vor avea outputul afisat pe ecran folosind o componenta de tipul: **ls** → ***JTable***, **userinfo** → ***JList***, **echo** → ***JDialog***.

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

## 4.4 Bonusuri

### 4.4.1 Autocomplete

Daca utilizatorul introduce comanda **cd** sau **rm**, in momentul in care specifica numele sau calea fisierului, terminalul va trebui sa permita posibilitatea autocomplet-ului (asemanator cu autocomplet-ul din IDE-urile in care scrieti cod Java). Cum utilizatorul scrie mai multe litere din parametru sau cale, autocompletarea va trebui sa reduca dinamic (in timp real) numarul de posibilitati. Utilizatorul va putea sa selecteze un entry din lista afisata de autocomplete cu mouse-ul sau folosind combinatia sageti + Enter.

### 4.4.2 Test Driven Development (TDD)

Realizati o suita de teste (pentru fiecare comanda, pentru a testa incarcarea corecta a fisierului cu ierahia de fisiere, s.a.m.d.) pe masura ce scrieti cod, dar inainte de a implementa o noua functionalite, pentru a va 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`

### 4.4.3 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.

# 5 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 obervatii, daca este cazul. Lipsa acestui fisier duce la o depunctare de ***10 puncte***.