

# Metode Avansate de Programare – SEMINAR 1+2 + LABORATOR 2

Acest document contine cerintele pt:

## A. Seminarul 1+2. (Eventual completari cu alte exercitii)

- Obiective: Familiarizarea cu limbajul Java, recapitulare concepte OOP, mecanisme OOP, principii OOP, sabloane de proiectare (Factory Method, Decorator, Singleton, Template Method), relatii intre clase (UML)

## B. Laborator 2. (cele care la final au Cerință laborator)

Rezolvați următoarele cerințe:

1. Definiți clasa **abstractă Task** avand atributele: taskID(String), descriere(String) și metodele: un constructor cu parametri, set/get, execute() (metoda abstractă), toString() și ~~equals hashCode~~; De ce trebuie să fie clasa Task abstractă? ✓

*Contractul equals - hashCode: dacă obj1.equals(obj2) atunci obj1.hashCode() == obj2.hashCode().*

*Ce se întâmplă cand avem o relație de moștenire între două clase și suprascriem equals? (a=b => b=a ?)*

2. Derivați clasa **MessageTask** din clasa Task, avand atributele *mesaj* (String), *from*(String), *to*(String) și *date* (*LocalDateTime*) și afișează pe ecran, via metoda *execute*, textul mesajului (valoarea atributului mesaj) și data la care a fost creat; (Vezi și *DateTimeFormatter*) ✓

*Clasa MessageTask ar putea fi refactorizată, astfel încă să incapsuleze un obiect de tipul Message având atributele: id, subject, body, from, to, date*

3. Derivați clasa **SortingTask** din Task care sortează un vector de numere întregi și afisează vectorul sortat, via metoda *execute()*. **Cerință laborator 2p**

**Observație:** Se vor acorda două puncte doar dacă SortingTask permite sortarea unui vector conform unei strategii, altfel se acorda 1p. Se cer două strategii de sortare – BubbleSort și (QuickSort sau MergeSort). Sugestie: SortingTask incapsulează un AbstractSorter ce are metoda sort.

4. Scrieți un program de test care creează un vector (array) de 5 task-uri de tipul **MessageTask** și le afisează pe ecran în urmatorul format:

**Exemplu:** `id=1|description=Feedback lab1|message=Ai obtinut 9.60|from=Gigi|to=Ana|date=2018-09-27 09:29`

Observație: Se va respecta formatul de afișare al datei. ✓

5. Considerăm că interfața **Container** specifică interfața comună pentru colecții de obiecte Task, în care se pot adăuga și din care se pot elimina elemente.

```
public interface Container {  
    Task remove();  
    void add(Task task);  
    int size();  
    boolean isEmpty();  
}
```

} practic fizierul .hr de la OOP

*Creați două tipuri de containere concrete:*

*last in, first out ↪*

1. **StackContainer** - care implementează, folosind o reprezentare pe un array, o strategie de tip LIFO;
2. **QueueContainer** - care implementează, folosind o reprezentare pe un array, o strategie de tip FIFO;  
**Cerință laborator 2p**

3. Refactorizați clasele **StackContainer** și **QueueContainer** astfel încât să evitați codul duplicat (bad smell). Vezi refactoringarea „Extract Superclass” (Solutia: Create an abstract superclass; make the original classes subclasses of this superclass, vezi cartea: **Refactoring: Improving the Design of Existing Code by Martin Fowler**). **Cerință laborator 1p**

6. Considerăm interfața **Factory** care conține o metodă *createContainer*, ce primește ca parametru o strategie (FIFO sau LIFO) și care întoarce un container asociat acelei strategii [Factory Method Pattern]. Creați clasa

- class → domain.Task
- în clasa: String
  - z
- alt + insert ⇒ constructor + generație gettere și settere
- Obiect (clasa inclusă), în toString returnez atributele
- în test (se pun teste)
- Task task = new Task("id", "desc");
- delete -- ca la C++ nu e necesar
- sout - pt afișare
- pe clasa Task, alt+insert ⇒ test
  - acmele clase care să le scrie, acmele obiecte care să le scrie
  - moșeul deosește clasei task cu celelalte
  - /\*\* - de ex: deosește la metoda set pt. specificații + enter
- la metodele care să le scrie
- clasa abstractă → interfață?
  - ↳ un comportament comun care alte clase pot să adere
- extends Task → extinde clasa Task (pt MessageTask)
- La Message Task - putem atribui prim setter
  - \* de făcut refacționare cu o clasă de message

- metoda execute()

ex: suntem "Task executat cu succes"

- metodele statice sunt scrise italic

- devenire se reprezinta dupa

- formator pt date:

```
public static final DateTimeFormatter format  
= DateTimeFormatter.ofPattern("dd/MM/yyyy  
HH:mm");
```

format.format(date);

- List - interfata

- Strategy.java



in clasa: LIFO, FIFO + interfata Container

+ o clasa StackContainer → asta e container de taskuri  
\* implementa

↳ pt a implementa o interfata

this.list = new ArrayList<>();

Tema

implementare metode din StackContainer

- la remove - de pe ultimul.

*TaskContainerFactory* care implementează interfața Factory. Creați containere de tipul Stack sau Queue doar prin apeluri ale metodei *createContainer*.

```
public interface Factory {  
    Container createContainer(Strategy startegy);  
}
```

7. Implementați clasa *TaskContainerFactory* care implementează interfața Factory, astfel încât să nu poată exista decat o singura instanță de acest tip. ([Singleton Pattern](#)) (**Cerință laborator 1p**) + discuție în timpul seminarului.

8. Considerăm interfața

```
public interface TaskRunner {  
    void executeOneTask(); //execută un task din colecția de task-uri de executat  
    void executeAll(); //execută toate task-urile din colecția de task-uri.  
    void addTask(Task t); //adaugă un task în colecția de task-uri de executat  
    boolean hasTask(); //verifică dacă mai sunt task-ri de executat  
}
```

care specifică interfața comună pentru o colecție de taskuri de executat.

9. Creați clasa **StrategyTaskRunner** care implementează interfața **TaskRunner** și care conține:

- Un atribut privat de tipul Container;
- Un constructor ce primește ca parametru o strategie prin care se specifică în ce ordine se vor executa task-urile (*LIFO* sau *FIFO*);

10. Scrieți un program de test care creează un vector de task-ri de tipul *MessageTask* și le executa, via un obiect de tipul *StrategyTaskRunner*, folosind strategia specificată ca parametru în linia de comandă. (`main(String[] args)`).

11. Definiți clasa abstractă **AbstractTaskRunner** ([Decorator Pattern](#)) care implementează interfața **TaskRunner** și care conține ca și atribut privat o referință la un obiect de tipul Task Runner, referință primită ca parametru prin intermediul constructorului.

12. Extindeți clasa *AbstractTaskRunner* astfel:

1. *PrinterTaskRunner* - care afișează un mesaj după execuția unui task în care se specifică ora la care s-a executat task-ul.
2. *DelayTaskRunner* – care execută taskurile cu întârziere; (**Cerință laborator 1p**)

```
try {  
    Thread.sleep(3000);  
} catch (InterruptedException e) {  
    e.printStackTrace();  
}
```

13. Scrieți un program de test care creează un vector de task-ri de tipul *MessageTask* și le executa, initial via un obiect de tipul *StrategyTaskRunner* apoi via un obiect de tipul *PrinterTaskRunner* (decorator), folosind startegia specificată ca parametru în linia de comandă.

14. Scrieți un program de test care creează un vector de task-ri de tipul *MessageTask* și le executa, initial via un obiect de tipul *StrategyTaskRunner* apoi via un obiect de tipul *DelayTaskRunner* (decorator) apoi via un obiect de tipul *PrinterTaskRunner* (decorator), folosind startegia specificată ca parametru în linia de comandă. (**Cerință de laborator 1p**)

15. Creați diagrama de clase. Ce relații între clase există în diagrama creată? (**Cerință de laborator 1p**)