

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 attributele: `taskId(String)`, `descriere(String)` si metodele: un constructor cu parametri, `set/get`, `execute()` (metoda abstracta), `toString()` si metodele `equals` - `hashCode`; De ce trebuie sa fie clasa Task abstracta?

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

Ce se intampla cand avem o relatie de mostenire intre doua clase si suprascriem equals? ($a=b \Rightarrow b=a$?)

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

Clasa *MessageTask* ar putea fi refactorizata, astfel inca sa incapsuleze un obiect de tipul *Message* avand attributele: *id*, *subject*, *body*, *from*, *to*, *date*

3. Derivați clasa **SortingTask** din *Task* care sortează un vector de numere întregi si afiseaza vectorul sortat, via metoda *execute()*. **Cerință laborator 2p**
Observatie: Se vor acorda doua puncte doar daca *SortingTask* permite sortarea unui vector conform unei strategii, altfel se acorda 1p. Se cer doua strategii de sortare – *BubbleSort* si (*QuickSort* sau *MergeSort*). Sugestie: *SortingTask* incapsuleaza un *AbstractSorter* ce are metoda *sort*.

4. Scrieti un program de test care creeaza un vector (array) de 5 task-uri de tipul **MessageTask** si le afiseaza pe ecran in urmatorul format:

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

Observatie: Se va respecta formatul de afisare al datei.

5. Consideram că interfata **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();  
}
```

Creați două tipuri de containere concrete:

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. Refactorizati clasele **StackContainer** si **QueueContainer** astfel incat sa evitati codul duplicat (bad smell). Vezi refactorizarea „*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

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

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

7. Implementați clasa *TaskContainerFactory* care implementează interfața *Factory*, astfel încât să nu poată exista decât o singură 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-uri de executat  
}
```

care specifică interfața comună pentru o colecție de task-uri 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-uri de tipul *MessageTask* și le execută, 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*, referința primită ca parametrul 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-uri de tipul *MessageTask* și le execută, inițial via un obiect de tipul *StrategyTaskRunner* apoi via un obiect de tipul *PrinterTaskRunner* (decorator), folosind strategia specificată ca parametru în linia de comandă.

14. Scrieți un program de test care creează un vector de task-uri de tipul *MessageTask* și le execută, inițial via un obiect de tipul *StrategyTaskRunner* apoi via un obiect de tipul *DelayTaskRunner* (decorator) apoi via un obiect de tipul *PrinterTaskRunner* (decorator), folosind strategia 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**)

Alte referinte:

A se vedea si cursul 1.

Martin Fowler - Refactoring, improving the design of existing code.

[Factory Method Pattern:] https://www.tutorialspoint.com/design_pattern/factory_pattern.htm

[Decorator Pattern:] https://www.tutorialspoint.com/design_pattern/decorator_pattern.htm

[Singleton Pattern] https://www.tutorialspoint.com/design_pattern/singleton_pattern.htm