

Metode Avansate de Programare, Sem 1+2

1. Definiți clasa **abstractă** `Task` având atributele: `taskId(String)`, `descriere(String)` și metodele: un constructor cu parametri, `set/get`, `execute()` (metoda abstractă), `toString()` și metodele `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 când avem o relație de mostenire între două clase și suprascriem equals? ($a=b \Rightarrow b=a$?)

2. Derivați clasa **`MessageTask`** din clasa `Task`, având 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 încât să încapsuleze un obiect de tipul `Message` având atributele: `id`, `subject`, `body`, `from`, `to`, `date`

3. Suplimentar (pt cine nu are frica de clase): Derivați clasa **`SortingTask`** din `Task` care sortează un vector de numere întregi și afișează vectorul sortat, via metoda `execute`. `SortingTask` permite sortarea unui vector conform unei strategii. Se cer două strategii de sortare – `BubbleSort` și `QuickSort` sau `MergeSort`). Sugestie: `SortingTask` încapsulează 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 afișează pe ecran în următorul 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();
}
```

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](#); Suplimentar
3. Refactorizați clasele **`StackContainer`** și **`QueueContainer`** astfel încât să evitați codul duplicat (bad smell). Vezi refactorizarea „*Extract Superclass*” (Soluția: Create an abstract superclass; make the original classes subclasses of this superclass, vezi cartea: ***Refactoring: Improving the Design of Existing Code by Martin Fowler***). Suplimentar

6. Considerăm că 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](#)]

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 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-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 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; (Suplimentar)

```
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.
15. Creați diagrama de clase. Ce relații între clase există în diagrama creată?

Alte referințe:

A se vedea și 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