



We Gonna Need This [Updated]

Calitate si testare Software (Academia de Studii Economice din București)



Scan to open on Studocu



Idea from Jens Wolfgang

CTS FOR DUMMIES

[A nu se lasa la indemana copiilor]
[Poate provoca traume]

Acest document este facut cu trup si suflet.

Daca nici asta nu ne salveaza, nu ne mai poate
salva nimic si nimeni.

CUPRINS

CLEAN CODE	4
PRINCIPII	4
SOLID.....	6
CONVENTII DE NUME.....	7
REGULI DE CLEAN CODE PENTRU METODE	7
REGULI DE CLEAN CODE IN CLASE.....	8
REGULI DE CLEAN CODE IN COMENTARII	8
DESIGN PATTERNS.....	9
DESIGN PATTERNS CREATIONALE.....	10
SINGLETON	10
DESCRIERE.....	10
TIPURI DE SINGLETON	10
UTILIZARI.....	11
SIMPLE FACTORY	12
DESCRIERE	12
UTILIZARI.....	12
FACTORY METHOD.....	12
DESCRIERE.....	12
ABSTRACT FACTORY	13
DESCRIERE.....	13
PROTOTYPE	13
DESCRIERE.....	13
UTILIZARI.....	13
BUILDER	13
DESCRIERE.....	13
IMPLEMENTARE	14
UTILIZARI.....	14
DESIGN PATTERNS STRUCTURALE	15
ADAPTER	15
DESCRIERE.....	15
STRUCTURA.....	15
IMPLEMENTARE	16
UTILIZARI.....	16
FAÇADE	16
DESCRIERE.....	16
IMPLEMENTARE	17

UTILIZARI.....	17
DECORATOR	17
DESCRIERE.....	17
IMPLEMENTARE	17
UTILIZARI.....	18
COMPOSITE.....	18
DESCRIERE.....	18
IMPLEMENTARE	19
UTILIZARI.....	19
FLYWEIGHT	19
DESCRIERE.....	19
IMPLEMENTARE	19
UTILIZARI.....	20
PROXY	20
DESCRIERE.....	20
UTILIZARI.....	20
DESIGN PATTERNS COMPORTAMENTALE	21
STRATEGY.....	21
DESCRIERE.....	21
IMPLEMENTARE	21
UTILIZARE.....	21
OBSERVER.....	22
DESCRIERE.....	22
IMPLEMENTARE	22
UTILIZARI.....	22
STATE.....	22
DESCRIERE.....	22
IMPLEMENTARE	23
UTILIZARI.....	23
COMMAND	23
DESCRIERE.....	23
IMPLEMENTARE	24
UTILIZARI.....	24
TEMPLATE METHOD.....	24
DESCRIERE.....	24
IMPLEMENTARE	25
UTILIZARI.....	25

CHAIN OF RESPONSIBILITY	26
DESCRIERE	26
IMPLEMENTARE	26
UTILIZARI	26
MEMENTO	27
DESCRIERE	27
IMPLEMENTARE	27
UTILIZARI	27
DESIGN PATTERNS RECAP	28
EXTRA: RELATII INTRE DESIGN PATTERNS	29
TESTARE UNITARA	33
TESTARE BLACKBOX	34
TESTARE WHITEBOX	35
JUnit	36
CONCEPTE	36
JUnit ASSERTIONS	36
Right-BICEP	37
RIGHT	37
BOUNDARY	38
INVERSE RELATIONSHIP	38
CROSS-CHECK	38
ERROR CONDITIONS	39
PERFORMANCE	39
CORRECT	39
F.I.R.S.T.	42
CODE COVERAGE	42
COMPLEXITATEA CICLOMATICA	42
DUBLURI DE TESTARE	43
DUMMY OBJECT	43
STUB	43
FAKE	44
SPY	44
MOCK	44
MOCK TESTING	44
JUnit5	45
GRILE	46

CLEAN CODE

Ce este? Un set de principii pentru a scrie cod **usor de inteles** si **usor de modificat**.

Ce presupune?

***usor de inteles:**

- ordinea de executie a intregii aplicatii este logica si bine structurata.
- legaturile dintre elementele codului sunt evidente.
- rolul fiecărei clase, functii, metode si variabile este imediat inteles.

***usor de modificat:**

- clasele si metodele sunt de mici dimensiuni si au un singur task.
- clasele si metodele sunt predictibile, functioneaza conform asteptarilor si sunt disponibile public prin interfete bine documentate.
- codul foloseste **unit tests**.

Alte beneficii?

- bugurile pot fi identificate si rezolvate mai usor.
- codul poate fi folosit si inteles de orice programator.

PRINCIPII

1. DRY – Don't Repeat Yourself
2. KISS – Keep It Simple and Stupid
3. YAGNI – You Ain't Gonna Need It
4. SOLID

KISS – codul trebuie scris cat mai simplu cu putinta.

- este unul dintre cele mai vechi principii de clean code.

DRY – o versiune a principiului KISS, conform careia fiecare functie trebuie sa faca un *singur* lucru.

WET(We Enjoy Typing) – opusul lui DRY; presupune repetitiile inutile din cod.

Exemplu WET:

```
//Version A
let username = getUsername();
let password= getPassword();
let user = { username, password};
client.post(user).then(/*Version A*/);

//Version B
let username = getUsername();
let password= getPassword();
let user = { username, password};
client.get(user).then(/*Version B*/);
```

Exemplu DRY:

```
function getUser(){
  return {
    user:getUsername();
    password:getPassword();
  }
}

//Version A
client.post(getUser()).then(/*Version A*/ );

//Version B
client.get(getUser()).then(/*Version B*/);
```

YAGNI – un programator trebuie sa introduca functionalitati noi doar atunci cand sunt necesare.

- strans legat de metodele de dezvoltare software agile.
- arhitectura software-ului trebuie dezvoltata pas cu pas pentru a putea adresa fiecare problema dinamic si individual => rezolvarea problemei in cel mai eficient mod.

SOLID

S – Single Responsibility principle (SRP)

O – Open-Closed principle (OCP)

L – Liskov Substitution principle (LSP)

I – Interface Segregation principle (ISP)

D – Dependency Inversion principle (DIP)

Single Responsibility principle (SRP)

"There should be no more than one reason to modify a class or a module."

- O clasă trebuie să aibă întotdeauna **o singură responsabilitate și numai una**
- În caz contrar orice schimbare de specificații va duce la inutilitatea ei și rescrierea întregului cod
- Soluția este împărțirea clasei în trei clase, astfel încât fiecare clasă să răspundă unui actor (se poate face împărțirea cu dp-ul FAÇADE!!)

Open-Closed principle (OCP)

"A software artifact should be open for extension, but closed for modification."

- Clasele trebuie să fie **deschise (open) pentru extensii**
- Dar totuși **închise (closed) pentru modificări**

Liskov Substitution principle (LSP)

"Derived classes must be substitutable for their base classes."

- **Obiectele pot fi înlocuite oricând cu instanțe ale claselor derivate** fără ca acest lucru să afecteze funcționalitatea
- Întâlnită și sub denumirea de „Design by Contract”

Interface Segregation principle (ISP)

"Many client-specific interfaces are better than one general purpose interface."

- **Mai multe interfețe specializabile** sunt oricând de preferat unei singure interfețe generale
- Nu riscăm astfel ca prin modificarea „contractului” unui client să modificăm și contractele altor clienți
- Obiectele nu trebuie obligate să implementeze metode care nu sunt utile

Dependency Inversion principle (DIP)

"Depend upon abstractions. Do not depend upon concretions."

- Este introdusă o clasă abstractă sau interfața de care să depindă modulele, pentru a facilita **reutilizarea** și a nu afecta modulele high-level prin modificări ale utilitatilor din module low-level.

CONVENTII DE NUME

- UpperCamelCase
- lowerCamelCase
- System Hungarian Notation
- Apps Hungarian Notation

REGULI DE CLEAN CODE PENTRU METODE

- Single responsibility - SRD
- Keep It Simple & Stupid - KISS
- Deleagă prin pointeri/interfețe
- Folosește interfețe

REGULI DE CLEAN CODE IN CLASE

- Toate metodele dintr-o clasă trebuie să aibă legătură cu acea clasă
- Evitați folosirea claselor generale și mutați prelucrările respective ca metode statice în clasele aferente
- Evitați primitivele ca parametri și folosiți obiecte (clase Wrapper în Java) ori de câte ori acest lucru este posibil
- Atenție la primitive și în cazul prelucrărilor în mai multe fire de execuție
- Folosiți fișiere de resurse pentru șirurile de caractere din GUI
- Clasele ce conlucrează vor fi așezate una lângă alta pe cât posibil
- Folosiți-vă de design patterns acolo unde situația o cere

REGULI DE CLEAN CODE IN COMENTARII

- De cele mai multe ori acestea nu își au deloc locul
- Codul bine scris este auto-explicativ
- Nu folosiți comentarii pentru a vă cere scuze
- Nu comentați codul nefolosit – devine zombie
- Există soluții de versionare pentru recuperarea codului modificat
- Atunci când simțiți nevoia de a folosi comentarii pentru a face o metodă lizibilă, cel mai probabil acea funcție trebuie separată în două funcții
- Evitați blocurile de comentarii introductive
- Toate detaliile de acolo se vor găsi în soluția de versionare
- Sunt indicate doar pentru:
 - biblioteci ce vor fi refolosite de alți programatori (doc comments)
 - <http://www.oracle.com/technetwork/articles/java/index137868.html>
 - TODO comments

DESIGN PATTERNS

- **CREATIONALE:** Singleton, Builder, Factory, Factory Method, Abstract Factory, Prototype.
- **STRUCTURALE:** Adapter, Composite, Decorator, Façade, Flyweight, Proxy.
- **COMPORTAMENTALE:** Chain of Responsibility, Command, Memento, Observer, State, Strategy, Template

		<i>Purpose</i>		
		Creational	Structural	Behavioral
<i>Scope</i>	Class	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Façade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

DESIGN PATTERNS CREATIONALE

SINGLETON

DESCRIERE

- un singur obiect de tipul clasei
- utilizatorii/apelatorii nu au acces la obiect (constructor privat, fara setteri)
- keywords: instanta unica, o singura instanta

DEZAVANTAJE

- **incalca SRP (Single Responsibility Principle)**, deoarece controleaza singur crearea instantei, dar si ciclul de viata => face doua lucruri odata
- **incalca OCP (Open-Closed Principle)**, deoarece, daca vrem sa adaugam sau sa modificam ceva, trebuie sa umblam in codul existent
- **incalca DIP (Dependency Inversion Principle)**, deoarece componentele low-level sunt in directa legatura cu cele high-level (cele doua trebuie sa fie clar separate)
- codul este tightly coupled, deci greu de testat
- poate duce la nevoia de a ordona testele unitare, ceea ce contrazice faptul ca unit test-urile trebuie sa fie independente
- daca dp-ul seamana cu Singleton dar are minim un parametru, atunci probabil e **Factory**

TIPURI DE SINGLETON

1. **Eager Initialization** = presupune initializarea instantei chiar daca nu e folosita
2. **Lazy Initialization** = cea mai implementata varianta de Singleton; problema apare cand e folosita multithreading, deoarece metoda poate fi apelata in acelasi timp pe doua fire de executie si astfel se creaza doua obiecte diferite
3. **Static Block Initialization** = seamana cu eager init. (instanta e initializata chiar daca nu e folosita!) doar ca varianta aceasta furnizeaza posibilitatea de captare a posibilelor exceptii generate de initializarea instantei statice
4. **Thread Safe Singleton** = asigura faptul ca metoda nu o sa fie apelata de alt fir de executie pana nu se termina metoda deja apelata pe un thread

5. **Inner Static Helper Class** = contine o clasa imbricata Helper in clasa Singleton, care va fi incarcata doar cand e apelata functia de creare a instantei; imbina eager init. cu lazy init.
6. **Enum Singleton** = presupune utilizarea unei enumerari pentru crearea unica a instantei, valorile enum-ului fiind accesibile la nivel global; nu permite lazy init.
7. **Serializarea singletonului** = daca avem serializare si apoi deserializam o instanta singleton, se obtin doua instante: cea creata si cea deserializata => pentru evitarea acestui aspect, se implementeaza metoda **readResolve()**, care returneaza instanta creata in cadrul singletonului
8. **Singleton vs Clasa Statica** = singleton respecta principiile OOP => un obiect Singleton poate fi trimis ca parametru unei functii, dar o clasa statica nu poate; o clasa Singleton poate implementa o interfata sau extinde alta clasa
9. **Singleton Collection** = **Singleton registry**, presupune gestiunea obiectelor unice intr-o colectie

UTILIZARI

- deschiderea unei singure instante ale unei aplicatii
- conexiune unica la baza de date
- accesarea resurselor dispozitivelor mobile(SharedPreferences in Android)
- DocumentBuilderFactory in Android

SIMPLE FACTORY

DESCRIERE

- Oferă posibilitatea creării de obiecte concrete dintr-o familie de obiecte, **fără să se știe exact tipul concret al obiectului.**
- Sintagma după care este recunoscut este: **familie de obiecte** sau **obiecte din aceeași familie.**
- Simple factory este un design pattern care nu este prezentat în cartea GoF, însă este folosit în practică deoarece este foarte ușor de implementat.

UTILIZARI

- crearea de view-uri pentru GUI
- existenta unei familii de obiecte intr-o aplicatie

FACTORY METHOD

DESCRIERE

- asemanator cu Simple Factory, doar ca in loc sa fie folosit enum, se abstractizeaza nivelul de creare
- are interfata care defineste obiectele, clasele concrete care implementeaza interfata si metoda **createObject()** care returneaza instantele claselor concrete
- se mai numeste si **Virtual Constructor**
- util cand nu poate fi anticipat tipul de obiect ce trebuie create
- poate fi implementat ca **Singleton**

DEZAVANTAJE

- incalca OCP **daca e folosit incorect** (de regula daca sunt folosite multe if-uri/switch-case-uri in loc sa fie totul structurat prin apelare de metode, *vezi stackoverflow*:
<https://softwareengineering.stackexchange.com/questions/302780/does-the-factory-pattern-violate-the-open-closed-principle/395056#395056>)
- se complica codul de fiecare data cand sunt adaugate subclase noi

ABSTRACT FACTORY

DESCRIERE

- introduce un nou nivel de abstractizare
- fiecare fabrica de obiecte creeaza obiecte din doua sau mai multe familii de obiecte

PROTOTYPE

DESCRIERE

- ajută la crearea de clone pentru obiectele a căror construire **durează foarte mult sau consumă foarte multe resurse**.
- prin intermediul acestui design pattern se creează un obiect considerat prototip, acest prototip urmând a fi **clonat** pentru următoarele instanțe din acea clasă.
- este **solutia optima** pentru crearea de obiecte asemanatoare

UTILIZARI

- atunci cand obiectele seamana intre ele, iar crearea unui obiect **dureaza foarte mult sau consuma foarte multe resurse**
- aplicabil ori de cate ori folosim **clone()**

BUILDER

DESCRIERE

- folosit pentru crearea de obiecte concrete, cu multe attribute/extraoptiuni
- se construiesc obiecte complexe prin specificarea anumitor proprietati dorite din multitudinea existenta
- obiectele create sunt finale

DEZAVANTAJE

- poate crește complexitatea codului, pentru că se creează un obiect complex pas cu pas

IMPLEMENTARE

Există două variante de implementare ale acestui Design Pattern:

- 1) Crearea obiectului complex în constructorul clasei Builder și modificarea atributelor conform cerințelor. În această situație Builder (PachetTransportBuilder) are un singur atribut de tipul Produs (PachetTransport);
- 2) Crearea obiectului complex se realizează în metoda build() pe baza setărilor realizate. În această situație clasa Builder (PachetTransportBuilder) conține aceleași atribute ca și clasa Produs (PachetTransport).

Există și o a treia variantă de implementare care presupune utilizarea unei clase imbricate (Inner class). Pentru această situație se folosește una dintre variantele prezentate anterior.

UTILIZARI

- în general pentru construirea de obiecte complexe cu foarte multe atribute
- StringBuilder
- DocumentBuilder în Android

DESIGN PATTERNS STRUCTURALE

ADAPTER

DESCRIERE

- Design pattern-ul Adapter rezolva problema utilizării anumitor clase din framework-uri diferite, care nu au o interfață comună.
- Clasele existente nu se vor modifica ci se va adăuga noi clase pentru realizarea unui Adapter între acestea. Clasa Wrapper.
- Utilizarea claselor existente se va face mascat prin intermediul adapterului creat.

Important: Adapterul nu adaugă funcționalitate. Funcționalitatea este realizată de clasele existente.

AVANTAJE

- este o implementare a **DIP (Dependency Inversion Principle)**
- crește transparența și ușurează reutilizarea clasei
- urmează **OCP (Open-Closed Principle)** și **SRP (Single Responsibility Principle)**
- flexibilitate (mai multe obiecte incompatibile pot interacționa)

DEZAVANTAJE

- în Java, clasa Adapter poate fi adaptată o singură dată (pentru că nu avem moștenire multiplă)
- poate crește complexitatea codului

STRUCTURA

Există două tipuri de Adapter, care diferă prin modul de implementare:

- **Adapter de obiecte**
- **Adapter de clase**

IMPLEMENTARE

Adapter de clase:

- Clasa Adapter moștenește clasa existentă și implementează interfața la care trebuie să facă adaptarea.
- Prin implementarea interfeței se asigură implementarea unui set de metode. Aceste metode vor face apeluri/call-uri ale metodelor specifice clasei existente prin intermediul părintelui (**super**).

Atenție: Java nu permite moștenire multiplă.

UTILIZARI

- Se folosește ori de câte ori este necesară conlucrarea mai multor frameworkuri și nu se dorește modificarea codului existent.

FAÇADE

DESCRIERE

- Ușurează lucrul cu framework-uri foarte complexe.
- Realizează o **fațadă** pentru aceste framework-uri, iar cine dorește să utilizeze acele framework-uri, poate folosi această fațadă, fără a fi necesară cunoașterea tuturor claselor, metodelor și atributelor din cadrul framework-ului.
- Se utilizează când avem de-a face cu **proces intermediare, fatade, simplificarea unui proces sau efectuarea mai multor acțiuni la un loc**.

AVANTAJE

- folosește principiul de **loose coupling**, deci un subsistem este mai ușor de extins
- minimizează complexitatea subsistemelor

DEZAVANTAJE

- implementare complicate
- instalarea unei interfete fațade necesită un nivel indirect/ocolitor
- nivel înalt de dependență de interfața fațade

IMPLEMENTARE

- Clasa Facade cuprinde metode care să utilizeze metodele din clasele frameworkului.
- Clasa Facade ascunde complexitatea prin apelurile sale.

UTILIZARI

- pentru framework-uri open-source disponibile.

DECORATOR

DESCRIERE

- Este folosit pentru adaugarea si modificarea de funcționalități ale unui obiect la runtime.
- Se pot face decorări multiple prin moștenire continuă.
- **Se foloseste pentru a adauga functionalitati si imbunatati clase deja existente in momentul executiei, fara a modifica codul existent**

AVANTAJE

- flexibilitate
- nu e nevoie de mostenire pentru extinderi
- codul este usor de citit
- functionalitati resource-optimized
- urmeaza OCP si SRP

DEZAVANTAJE

- complexitate inalta (mai ales a interfetei Decorator)
- poate incalca OCP
- nu e beginner-friendly
- numar mare de obiecte
- procesul de debugging e dificil

IMPLEMENTARE

- În cadrul clasei abstracte se implementează interfața care definește familia de obiecte și se creează o instanță de tipul acelei interfețe.

- Pentru metoda din interfață se furnizează o implementare și se adaugă noi funcții abstracte.
- În cadrul clasei **decorator concret** se implementează și **noile metode din decoratorul abstract**.

UTILIZARI

- adaugarea de noi functionalitati claselor existente
- imbunatatirea claselor existente fara a modifica codul existent si fara a face extindere/mostenire

COMPOSITE DESCRIERE

- Este un design patterns structural folosit atunci când este necesară crearea unei **structuri ierahice** sau o **structură arborescentă** prin compunerea de obiecte.

ATENȚIE: Composite nu este o structură de date (arbore). Composite este un design pattern.

DEZAVANTAJE

- poate incalca LSP (Liskov Substitution Principle) (componentele au proprietate ca le poti adauga subcomponente (copii), dar frunza nu poate avea copii, chiar daca e un subtip al Component, deci principiul este incalcat (vezi *stackoverflow*: <https://stackoverflow.com/a/1580695>)
- poate incalca si ISP (Interface Segregation Principle)

Although the Composite class *implements* the **Add** and **Remove** operations for managing children, an important issue in the Composite pattern is which classes *declare* these operations... Should we declare these operations in the Component and make them meaningful for Leaf classes, or should we declare and define them only in Composite?

The decision involves a trade-off between safety and transparency:

- Defining the child management interface at the root of the class hierarchy gives you transparency, because you can treat all components uniformly. It costs you safety, however, because clients may try to do meaningless things like add and remove objects from leaves.
- Defining child management in the Composite class gives you safety, because any attempt to add or remove objects from leaves will be caught at compile-time in a statically typed language like C++. But you lose transparency, because leaves and composites have different interfaces.

We have emphasized transparency over safety in this pattern.

IMPLEMENTARE

- Clasele Composite conțin o listă cu elemente de tip ComponentAbstracta. În această listă se adauga si se șterg noduri noi.
- Clasele NodFrunza implementeaza metodele de adaugare nod sau stergere nod, chiar dacă nu au nicio implementare pentru ele.

UTILIZARI

- meniuri de aplicatii
- meniuri pentru restaurant
- reprezentarea unei arborescente

FLYWEIGHT

DESCRIERE

- Este utilizat atunci când trebuie să construim **foarte multe obiecte/instanțe** ale unei clase, însă **majoritatea obiectelor au o parte comună, sau permanentă.**
- Astfel prin utilizarea design pattern-ului Flyweight **se reduce consumul de memorie**, păstrându-se într-o singură instanță partea comună.
- Partea care diferă de la un obiect la altul este salvată într-o altă clasă și este adăugat după construirea obiectelor.

IMPLEMENTARE

- Clasa FabricaDePachete conține un HashMap pentru reținerea pachetelor asemănătoare.
- Clasa PachetTuristic reține codul, hotelul, destinația și daca are sau nu mic dejun, deoarece aceste attribute au aceleași valori pentru mai mulți turiști.
- Clasa Optionale retine doar numărul de excursii si daca are sau nu cina, deoarece aceste attribute diferă foarte mult.

UTILIZARI

- in jocuri, atunci când foarte multe modele seamănă, însă diferă prin culoare sau prin poziție (copaci, mașini, oameni, etc).

PROXY **DESCRIERE**

- Este utilizat atunci când se dorește **păstrarea funcționalității unei clase, însă aceasta se va realiza doar în anumite condiții.**
- Prin Proxy **se controlează comportamentul și accesul la un obiect.**

UTILIZARI

- De fiecare dată când se dorește realizarea de permisiuni pentru anumite obiecte sau pentru accesul la anumite modificări ale obiectelor.

DESIGN PATTERNS COMPORTAMENTALE

STRATEGY

DESCRIERE

- Este folosit atunci când avem mai mulți algoritmi pentru rezolvarea unei probleme, iar **alegerea implementării se face la run-time**.
- Fiecare comportament este dat de o clasă.
- Definește **strategia adoptată la run-time**.

AVANTAJE

- satisface SRP, OCP, LSP, DIP

DEZAVANTAJE

- desi, by default, satisface ISP, programatorul poate sa scrie cod care sa il incalce

IMPLEMENTARE

- În cadrul fiecărei clase Strategy concret se implementează metoda de procesare.
- În cadrul clasei care gestionează instanța Strategy, se pune la dispoziție un mecanism de schimbare a metodei, și se are grija ca implementarea să folosească implementarea furnizată de instanța concretă a obiectului Strategy.

UTILIZARE

- furnizarea metodei de comparare pentru metodele de sortare
- utilizarea validatoarelor pentru anumite controale

OBSERVER

DESCRIERE

- Observer definește o relație de “1 : n”, în care un subiect **notifică** mai mulți observeri.
- Acest design pattern este folosit atunci când **anumite elemente (obiecte) trebuie să fie anunțate de schimbarea stării altor obiecte.**

AVANTAJE

- satisface OCP
- poate fi considerat o aplicare a DIP

DEZAVANTAJE

- poate încalca SRP și ISP

IMPLEMENTARE

- În cadrul clasei concrete a subiectului observabil se gestionează o listă de obiecte care observă.
- În metoda de trimitere notificare se parcurge lista de observatori și se trimite un mesaj fiecăruia prin apelul funcției specifice.

UTILIZARI

- Model View Controller

STATE

DESCRIERE

- este un design pattern comportamental folosit atunci când **un obiect își schimbă comportamentul pe baza stării în care se află.**
- Este foarte asemănător cu Strategy, diferența constă în modul de schimbare a stării respectiv a strategiei.

AVANTAJE

- satisface OCP și SRP
- face codul mai ușor de citit și de extins
- minimizează complexitatea condițională, deoarece nu mai trebuie să folosim if-uri și switch

DEZAVANTAJE

- stările trebuie cunoscute
- poate incalca LSP, deoarece metodele din subclasa pot arunca exceptii noi fata de cele din superclasa (*vezi link:*
<https://softwareengineering.stackexchange.com/questions/181922/does-the-state-pattern-violate-liskov-substitution-principle>)
- poate creste complexitatea odata cu adaugarea de noi clase
- posibila incalcare a YAGNI (clasele contextuale nu au o logica proprie, ci doar clasele specific starilor)

IMPLEMENTARE

- În cadrul fiecărei clase de stare exista metoda de setare a stării prin care se modifică starea contextului sau a rezervării, în cazul de față.
- Modificarea stării nu se face prin setare din programul apelator ca la Strategy, ci prin apelul acestei stări.

UTILIZARI

- Stările rezervărilor sau a comenzilor în orice aplicație.
- Folosit pentru evitarea utilizării structurii switch sau if-else.

COMMAND

DESCRIERE

- Este folosit pentru implementarea **loose coupling**.
- In acest mod ascunde aplicarea de comenzi, fără se știe concret ce presupune acea comandă.
- Astfel clientul este decuplat de cel ce execută acțiunea.

AVANTAJE

- satisface principiile de Clean Code (poate satisface toate principiile SOLID)
- este DRY
- usor de extins si de inteles
- separa clientul de obiectul care proceseaza request-urile

DEZAVANTAJE

- sunt trimise request-uri fara a fi cunoscute acestea sau obiectele care le primesc
- multe clase mici pentru comenzi

IMPLEMENTARE

Pentru implementare există două situații:

- Invoker-ul să fie folosit doar pentru a invoca comenzile;
- Invoker-ul poate salva comenzile invocate, și astfel se poate face undo pe comenzile executate sau invocate.

Pentru a doua situație avem nevoie de o listă de comenzi în Invoker și de metode apelate pe undo() în toate clasele.

UTILIZARI

- macros
- lucrul cu fisiere
- oriunde se dorește revenirea la o stare anterioară prin intermediul comenzilor

TEMPLATE METHOD

DESCRIERE

- Folosit atunci când **un algoritm este cunoscut și urmează anumiți pași preciși.**
- Fiecare pas este realizat de câte o metodă.
- Există o metodă care implementează algoritmul și apelează toate celelalte metode

AVANTAJE

- ușor de înțeles și de citit
- standardizează ordinea operațiilor

DEZAVANTAJE

- poate abuza de mosteniri (exemplu: Android framework)
- nu trebuie folosit doar de dragul de a-l folosi (daca nu stim cu certitudine ca avem nevoie sa impartim codul in clase abstracte si concrete, vom incalca **YAGNI**)

IMPLEMENTARE

- În clasa abstractă, metoda template se declară finală, astfel încât să nu poată fi suprascrisă.
- În cadrul claselor concrete sunt implementate doar metodele folosite în metoda template.

UTILIZARI

- Atunci când modul de procesare sau de rezolvare a unei probleme urmează **un număr finit și cunoscut de pași**
- Backtracking

CHAIN OF RESPONSIBILITY

DESCRIERE

- este folosit atunci când cel care are nevoie de rezolvarea unei probleme nu știe exact cine poate să rezolve problema, însă are o listă de posibile obiecte ce pot rezolva problema.
- obiectele posibile se ordonează într-un lanț, apoi cel care are problema de rezolvat apelează pentru prima dată din lanț.

AVANTAJE

- ordinea de prelucrare a request-urilor poate fi controlată (flexibilitate)
- satisface SRP și OCP

DEZAVANTAJE

- unele request-uri pot ajunge să nu fie prelucrate, pentru că nu au un receiver anume

IMPLEMENTARE

- Gestiunea următorului handler se face în clasa abstractă.
- În cazul în care un handler concret nu poate rezolva problema apelează următorul handler.
- Ultimul handler din lanț trebuie implementat astfel încât să nu apeleze la un următor handler, deoarece acesta nu există.

UTILIZARI

- DNS Resolver

MEMENTO

DESCRIERE

- Folosit atunci când se dorește salvarea anumitor stări pentru obiectele unei clase
- Permite salvarea și revenirea la stările salvate ori de câte ori acest lucru este dorit

AVANTAJE

- satisface SRP
- un mod ușor de a ține evidența istoricului ciclului de viață al unui obiect
- nu compromite încapsularea

DEZAVANTAJE

- tightly coupled
- consumă mult RAM dacă clienții creează memento-uri prea des
- în limbajele de programare dinamice (exemplu: Python, JavaScript, PHP), nu putem fi siguri că doar Originator poate accesa starea Memento-ului

IMPLEMENTARE

- Clasa Memento poate fi una externă și se ocupă de atributele pentru care trebuie realizată imaginea intermediară.
- Deoarece această clasă este folosită doar de către obiectele de un anumit tip, clasa Memento poate fi inclusă în cadrul clasei respective (ex. obiecte de tip PachetTuristic, clasa PachetTuristic)

UTILIZARI

- salvarea fișierelor
- realizarea de backup

DESIGN PATTERNS RECAP

DP CREATIONALE:

Singleton – se poate crea o singură instanță pentru o clasă, are constructorul privat;

Factory – crează obiecte dintr-o familie de clase. Simple Factory, Factory Method, Abstract Factory;

Builder – ajută la crearea obiectelor complexe cu foarte multe atribute;

Prototype – folosit atunci când crearea unui obiect consumă foarte multe resurse. Se creează un prototip și este folosit pentru clonare.

DP STRUCTURALE:

Adapter – adaptează un framework, astfel încât să lucreze cu un alt framework. Nu adaugă funcționalitate;

Facade – simplifică lucrul cu framework-uri complexe;

Decorator – adaugă noi funcționalități la run-time;

Composite – folosit pentru crearea și gestiunea ierarhiilor;

Flyweight – gestionează eficient obiectele pentru o utilizare optimă a memoriei;

Proxy – controlarea comportamentului și impunerea de condiții;

DP COMPORTAMENTALE

Strategy – schimbă la run-time metoda apelată;

Observer – când un subiect își schimbă starea – n observatori sunt notificați;

Chain of Responsibility – se formează un lanț pentru rezolvarea unei probleme. În cazul în care o verigă a lanțului nu poate rezolva problema, apelează la veriga succesoare;

State – schimbă comportamentul pe baza stării în care se află obiectul;

Command – gestiunea de comenzi aplicate asupra obiectelor;

Template Method – pentru gestiunea unui pattern de pași;

Memento – salvarea și gestiunea stărilor anterioare ale obiectului;

EXTRA: RELATII INTRE DESIGN PATTERNS

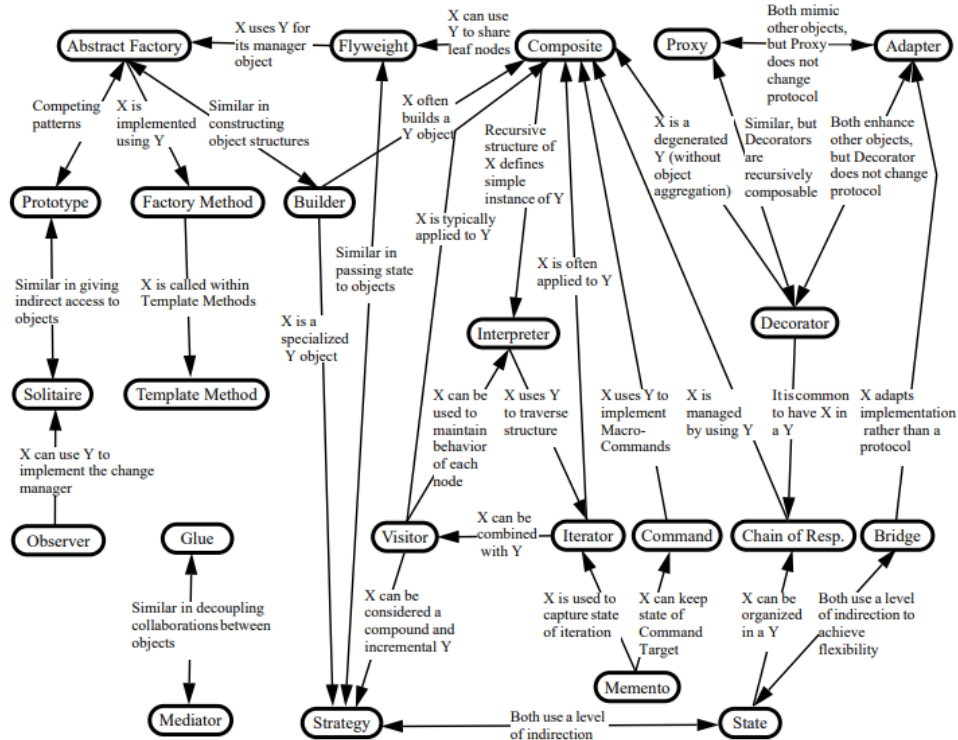


Figure 1 The starting point: The overall structure of the design pattern catalogue

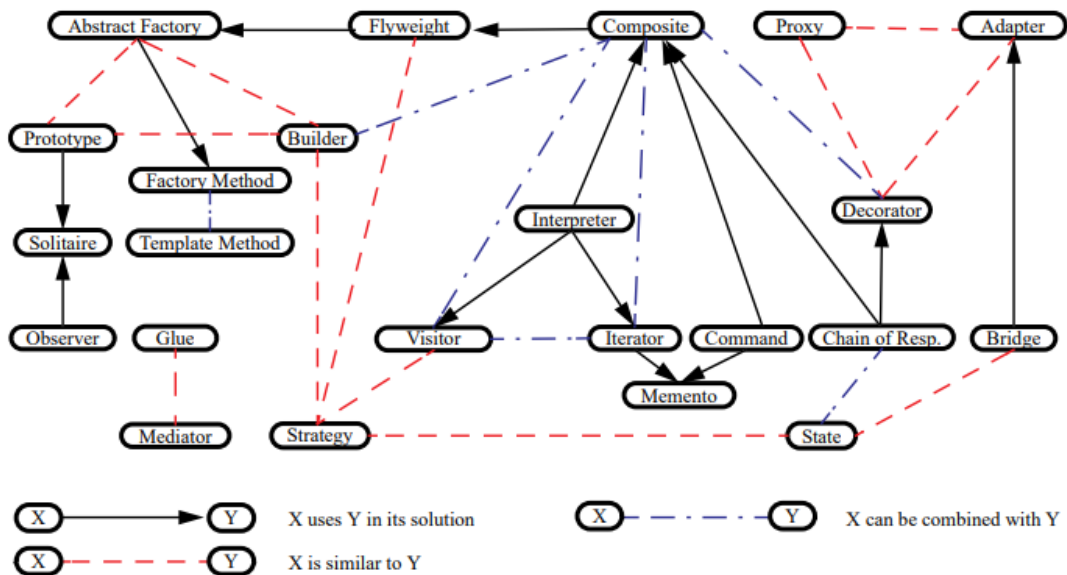


Figure 2 Classification of Relationships

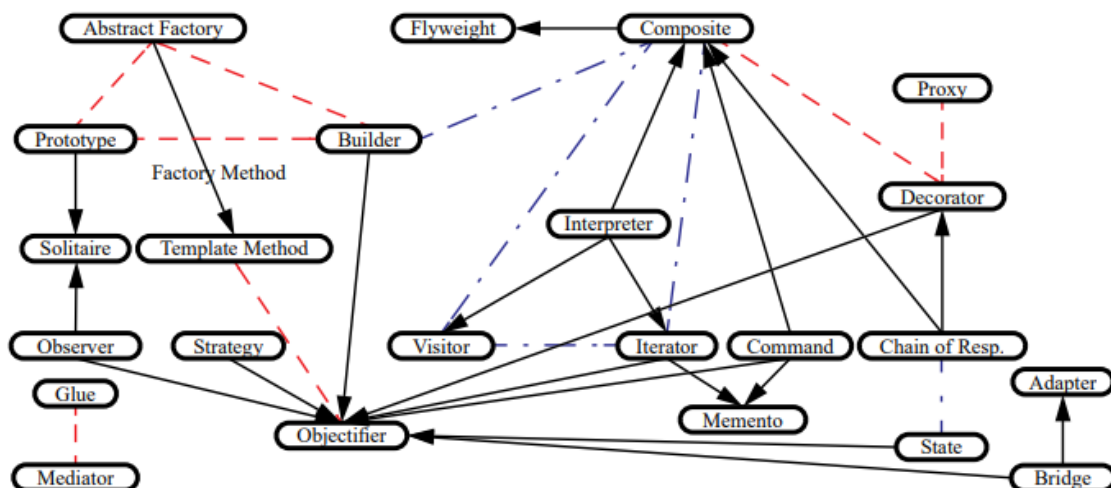


Figure 3 Revised Classification

O clasa **FAÇADE** poate fi transformata într-un **SINGLETON**.

FLYWEIGHT seamana cu **SINGLETON** daca toate starile commune pot fi reduce într-un singur obiect flyweight.

ABSTRACT FACTORIES, **BUILDERS** si **PROTOTYPES** pot fi implementate ca **SINGLETONS**.

O **FACTORY** unica poate fi implementata ca **SINGLETON**.

Prin intermediul unei **FACTORY** sunt create nodurile container si nodurile frunza din **COMPOSITE**.

Multe design-uri incep de la **FACTORY METHOD** si evolueaza spre **ABSTRACT FACTORY**, **PROTOTYPE**, **BUILDER**.

FACTORY METHOD este o specializare a **TEMPLATE METHOD**. In acelasi timp, **FACTORY METHOD** poate fi o etapa într-un **TEMPLATE METHOD** mai amplu.

Clasele **ABSTRACT FACTORY** sunt deseori bazate pe un set de **FACTORY METHODS**, dar poate fi folosit si **PROTOTYPE** pentru a compune metodele din clasele respective.

ABSTRACT FACTORY poate fi o alternativa pentru **FAÇADE** cand dorim sa ascundem modul in care sunt create obiectele din subsistem de codul pentru client.

BUILDER poate fi folosit pentru a crea arbori **COMPOSITE** mai complecsi.

PROTOTYPE poate fi folosit pentru a salva copii ale **COMMANDS** in istoric.

PROTOTYPE poate fi o alternativa mai simpla pentru **MEMENTO**, in situatia in care obiectul, adica starea care trebuie salvata in istoric, nu are legaturi cu surse externe, sau daca legaturile sunt usor de restabilit.

ADAPTER ofera o interfata diferita pentru obiectul wrapped, **PROXY** ofera aceeasi interfata, iar **DECORATOR** ofera o interfata extensa.

ADAPTER, **STATE** si **STRATEGY** au structuri foarte asemanatoare, dar rezolva probleme diferite.

CHAIN OF RESPONSIBILITY este deseori folosit in conjunctura cu **COMPOSITE** in felul urmatoar: cand un nod frunza primeste un request, il trimite prin lantul tuturor nodurilor parinte pana la radacina obiectului arbore.

COMPOSITE si **DECORATOR** au diagrame de structura similare. **DECORATOR** are un singur copil si adauga responsabilitati aditionale obiectului, pe cand **COMPOSITE** rezuma toate rezultatele copiilor.

DECORATOR poate extinde comportamentul obiectului din arborele **COMPOSITE**.

Nodurile comune din arborele **COMPOSITE** pot fi implementate ca **FLYWEIGHTS** pentru a folosi mai putin RAM.

Solutiile care folosesc **COMPOSITE** si **DECORATOR** pot folosi in schimb **PROTOTYPE** pentru a clona structurile complexe in loc sa le reface de la 0 mereu.

ADAPTER modifica interfata unui obiect existent, pe cand **DECORATOR** adauga functionalitati obiectului fara sa schimbe interfata. In plus, **DECORATOR** poate folosi compunere recursive, cee ace nu este posibil cand implementam **ADAPTER**.

CHAIN OF RESPONSIBILITY si **DECORATOR** au structurile claselor foarte asemanatoare.

“Decorator lets you change the skin of an object, while Strategy lets you change the guts.” (sursa: <https://refactoring.guru/design-patterns/decorator>)

DECORATOR si **PROXY** au structure asemanatoare, dar scopuri diferite. **PROXY** controleaza ciclul de viata al obiectului, pe cand **DECORATOR** lasa asta pe seama clientului/utilizatorului.

FAÇADE defineste o noua interfata pentru obiectele existente, pe cand **ADAPTER** utilizeaza interfata deja existenta. **ADAPTER** lucreaza cu un obiect, pe cand **FAÇADE** cu un intreg subsistem de obiecte.

FLYWEIGHT arata cum se pot crea multe obiecte de mici dimensiuni, pe cand **FAÇADE** arata cum se poate crea un singur obiect care sa reprezinte un intreg subsistem.

FAÇADE si **PROXY** sunt similare, in sensul ca ambele amortizeaza* o entitate complexa si o initializeaza pe cont propriu. (* - asa mi-a tradus, original “buffer”)

CHAIN OF RESPONSIBILITY, **COMMAND** si **OBSERVER** se adreseaza mai multor cai de stabilire a legaturii dintre emitatorii si receptorii request-urilor:

- 1) **CHAIN OF RESPONSIBILITY** transmite secvential un request de-a lungul unui lant dinamic de potentiali receptori pana cand unul din ei il gestioneaza.
- 2) **COMMAND** stabileste conexiuni unidirectionale intre emitatori si receptori.
- 3) **OBSERVER** permite receptorilor sa se aboneze in mod dinamic si sa se dezaboneze de la primirea request-urilor.

Handler-ii din **CHAIN OF RESPONSIBILITY** pot fi implementati ca si **COMMANDS**.

COMMAND si **MEMENTO** pot fi folosite impreuna pentru a implementa “undo”.

Comenzile sunt responsabile pentru operatiuni, pe cand memento-urile salveaza starile obiectului inainte de fiecare comanda.

COMMAND si **STRATEGY** arata similar, pentru ca amandoua parametrizeaza un obiect folosind o actiune. Totusi, au scopuri diferite. **COMMAND** converteste orice operatiune intr-un obiect, pe cand **STRATEGY** descrie de obicei diferite moduri in care se poate face acelasi lucru.

STATE poate fi considerat o extensie a lui **STRATEGY**. **STRATEGY** face obiectele independente intre ele si fara sa stie unul de existenta altuia. **STATE** nu restrictioneaza dependintele dintre starile concrete, lasandu-le sa modifice starile contextului dupa cum doresc.

TEMPLATE METHOD este bazat pe mostenire: permite alterarea partilor din algoritm prin extinderea acelor parti in subclase. **STRATEGY** este bazat pe compunere: permite alterarea partilor comportamentului unui obiect prin suplimentarea acestuia cu diferite strategii care corespund comportamentului respective. **TEMPLATE METHOD** lucreaza la nivel de clasa, deci e static. **STRATEGY** lucreaza la nivel de obiect, comportamentele putand fi schimbate la runtime.

TESTARE UNITARA

Ce este UNIT TESTING?

O cale de testare a codului, de către programatori încă din etapa de dezvoltare a produsului software.

UNIT TEST = secvență de cod folosită pentru testarea unei unități bine definite din codul aplicației software. De obicei unitatea este reprezentată de o metodă.

Testarea unității se realizează într-un context bine definit în specificațiile de testare.

De ce sa folosim teste unitare?

- Ușor de scris
- Testele pot fi scrise ad-hoc atunci când ai nevoie de ele
- Deși sunt simple, pe baza lor se pot defini colecții de teste – Test Suites
- Pot fi rulate automat de fiecare dată când e nevoie (write once, use many times)
- Există multe framework-uri și instrumente ce simplifică procesul de scriere și rulare
- Reduc timpul pierdut pentru debugging și pentru găsirea bug-urilor
- Reduc numărul de bug-uri în codul livrat sau integrat
- Crește rata bug-urilor identificate în faza de scrierea a codului
- Există multe framework-uri și instrumente ce simplifică procesul de scriere și rulare
- Reduc timpul pierdut pentru debugging și pentru găsirea bug-urilor
- Reduc numărul de bug-uri în codul livrat sau integrat
- Crește rata bug-urilor identificate în faza de scrierea a codului

TESTARE BLACKBOX

= o metodă utilizată pentru a testa aplicația software de către persoane care nu cunosc arhitectura internă a aplicației testate.

Testerul cunoaște doar datele de intrare și datele de ieșire ale aplicației.

AVANTAJE:

- Testele sunt realizate din punctul de vedere al utilizatorului.
- Testerul nu trebuie să știe programare, limbajul folosit pentru dezvoltare sau structura de cod a aplicației.
- Testele sunt efectuate independent de dezvoltatorii și au o perspectivă obiectivă.

DEZAVANTAJE:

- Cazurile de testare sunt dificil de proiectat, deoarece testerul nu are caietul de sarcini al aplicației;
- Testele vor avea un număr mic de intrări;
- Testele pot fi redundante cu alte teste realizate de dezvoltator.

TESTARE WHITEBOX

= o metodă de testare folosită de către dezvoltatori sau de testeri care cunosc structura internă a aplicației testate.

Testarea WhiteBox mai este cunoscută și sub formele:

- Clear Box Testing;
- Open Box Testing;
- Glass Box Testing;
- Transparent Box Testing;
- Code-Based Testing;
- Structural Testing.

AVANTAJE:

- Testarea poate fi începută într-o etapă anterioră punerii în funcțiune. Nu trebuie să se aștepte realizarea interfeței pentru a fi realizată testarea.
- Testarea este mai aprofundată, cu posibilitatea de a acoperi cele mai multe posibilități.

DEZAVANTAJE:

- Din moment ce testele pot fi foarte complexe, sunt necesare resurse de înaltă calificare, cu o cunoaștere aprofundată a programării și a punerii în aplicare.
- Întreținerea codului de testare poate fi o povară în cazul în care punerea în aplicare se schimbă foarte des.

JUnit

– este un framework ce permite realizarea și rularea de teste pentru diferite metode din cadrul proiectelor dezvoltate.

= Cel mai folosit framework pentru testarea unitară a codului scris în JAVA.

= Reprezintă o adaptare de la xUnit.

JUnit funcționează conform a două design patterns: Composite și Command.

O clasă `TestCase` reprezintă un obiect `Command` iar o clasă `TestSuite` este compusă din mai multe instanțe `TestCase` sau `TestSuite`.

CONCEPTE

Fixture – set de obiecte utilizate în test

Test Case – clasă ce definește setul de obiecte (fixture) pentru a rula mai multe teste

Setup – o metodă/etapă de definire a setului de obiecte utilizate (fixture), înainte de testare.

Teardown – o metodă/etapă de distrugere a obiectelor (fixture) după terminarea testelor

Test Suite – colecție de cazuri de testare (test cases)

Test Runner – instrument de rulare a testelor (test suite) și de afișare a rezultatelor

JUnit ASSERTIONS

`assertEquals(expected, actual)` – verifică dacă valorile `expected` și `actual` sunt egale

`assertEquals(expected, actual, delta)` – `delta` = diferența maximă dintre `expected` și `actual` pentru care numerele sunt considerate egale

`assertSame(expected, actual)` – verifică dacă variabilele fac referire la același obiect

`assertNotSame(expected, actual)` – verifică dacă variabilele nu fac referire la același obiect

`assertNull(object)` – verifică dacă obiectul este null

`assertNotNull(object)` – verifică dacă obiectul nu este null

`assertTrue(condition)` – verifică dacă condiția este adevărată

`assertFalse(condition)` – verifică dacă condiția este falsă

`fail(message)` – pica un test și aruncă `AssertionFailedError`; se poate folosi să verificăm dacă este aruncată o excepție

`assertEquals(message, expected, actual)` – `message` = mesajul afișat dacă testul pica

`assertEquals(message, expected, actual, delta)`

`assertSame(message, expected, actual)` – verifica dacă variabilele fac referire la același obiect, se afișează mesajul dacă testul pica

`assertNotSame(message, expected, actual)`

`assertNull(message, object)`

`assertNotNull(message, object)`

`assertTrue(message, condition)`

`assertFalse(message, condition)`

`fail(message)`

Right-BICEP

RIGHT – dacă rezultatele furnizate de către metodă sunt corecte;

B – trebuie verificate toate limitele (**Boundery**) și dacă în cazul acestor limite rezultatele furnizate de metoda testată sunt de asemenea corecte;

I – trebuie verificate relațiile inverse (**Inverse**);

C – trebuie verificată corectitudinea printr-o verificare încrucișată (**Cross-Check**), folosind metode de calcul asemănătoare, testate și validate de către o comunitate mare de programatori;

E – trebuie simulată și forțată obținerea erorilor (**Errors**) pentru verificarea comportamentului metodei în cazul anumitor erori;

P – trebuie verificată păstrarea performanței (**Performance**) între limitele acceptanței pentru produsul software final.

RIGHT

De fiecare dată când testăm o metodă, primul lucru care ar trebui verificat este că această metodă oferă rezultatele corecte.

De aceea prima direcție este de a verifica corectitudinea rezultatelor.

Această verificare se face în conformitate cu specificațiile proiectului dezvoltat.

BOUNDARY

Problemele apar de obicei la “marginii”, deci trebuie să fim atenți **să testăm metodele pentru limitele intervalelor.**

Pentru fiecare metodă, trebuie să determinăm intervalul în care pot fi valorile parametrilor de intrare, precum și intervalul de rezultate furnizat de metodă.

Odată ce aceste limite au fost determinate, se efectuează teste exact pentru aceste valori.

Testele Boundary nu presupun testarea valorilor din afara acestor valori, ci verificarea corectitudinii acestor valori - valori limită.

Există de obicei limite inferioare și limite superioare. Testele se fac pentru ambele situații.

Pentru a identifica mai ușor limitele extreme, putem să utilizăm principiul **CORRECT**.

INVERSE RELATIONSHIP

Anumite metode pot fi testate prin aplicarea regulii inverse: **pornind de la rezultat, trebuie să se ajungă la aceeași intrare de la care a început inițial.**

Nu se aplică pentru toate metodele. De obicei se aplică metodelor matematice.

De asemenea, pentru bazele de date se poate verifica dacă a fost efectuată o inserare prin operația inversă: **select**.

CROSS-CHECK

Pentru fiecare metodă, putem încerca să o testăm utilizând altă metodă.

De obicei, există mai multe modalități de a rezolva o problemă. Astfel, se poate utiliza o altă metodă pentru rezolvarea problemei pentru verificarea / testarea metodei nou implementate.

Această situație este posibilă atunci când metoda implementată a fost concepută pentru a crește productivitatea sau dacă metoda veche consuma prea multe resurse.

Testarea metodei noi se face prin metoda veche, chiar dacă aceasta consumă mai multe resurse.

ERROR CONDITIONS

Probabil cel mai urât scenariu pentru o aplicație este să crape. De aceea, atunci când testăm fiecare metodă unitar, trebuie să testăm și situațiile în care aplicația ar putea să crape.

Dacă am studiat limitele extreme pentru valorile de intrare sau valori rezultate, **testarea pentru furnizarea erorilor ar trebui să utilizeze valori în afara acestor intervale.**

Testarea de forțare a erorilor se face pentru toate metodele. Toate metodele au cel puțin o situație în care vor oferi erori. Testarea se face pentru aceste situații și se verifică dacă metoda tratează acel caz și aruncă sau oferă o excepție.

PERFORMANCE

Pentru diferite metode, este posibil să se testeze cât de bine funcționează metoda respectivă.

Pe lângă testarea corectitudinii rezultatelor metodelor, este foarte important să se verifice performanța procesării.

Verificarea performanței se face atât din punctul de vedere al resurselor consumate, cât și din punctul de vedere al timpului necesar pentru obținerea rezultatelor.

Testarea performanței este efectuată atunci când input-ul sau rezultatul unei metode este reprezentat de o listă sau de un număr foarte mare de elemente, iar aceste valori pot crește foarte mult.

Pentru JUnit4 pentru a testa timpul în care rulează o anumită metodă, este folosită următoarea adnotare : **@Test(timeout=100)**

CORRECT

C – Conformitatea formatului (Conformance);

O – Ordinea (Order);

R – Intervalul (Range);

R – Referințe externe (References);

E – Existența obiectelor sau a rezultatelor (Existence);

C – Cardinalitatea rezultatelor (Cardinality);

T – Timpul (Time).

Fiecare sub-principiu are o întrebare care ar trebui să fie în mintea testerului.

Acest principiu este folosit și pentru a stabili condițiile limită pentru testele de Boundary din Right-BICEP.

CONFORMANCE

Este, de asemenea, cunoscut sub numele de:

- **Type testing**
- **Compliance testing**
- **Conformity assessment**

Se aplică în numeroase domenii în care ceva ar trebui să îndeplinească anumite standard specifice.

De obicei, pentru orice intrare și pentru orice ieșire, trebuie să se verifice conformitatea cu un format sau cu un standard.

Testele pot fi efectuate pentru a verifica ce se întâmplă dacă datele de intrare nu sunt conforme cu formatul sau pentru a vedea dacă rezultatul obținut este conform cu formatul specific proiectului.

ORDERING

Testele de ordine sunt specifice listelor, dar nu numai.

În cazul listelor, trebuie să verificăm dacă ordinea articolelor este cea dorită.

De asemenea, putem testa comportamentul metodei dacă primește anumiți parametri într-o altă ordine sau o listă de elemente într-o ordine diferită de cea așteptată.

RANGE

Pentru valorile de intrare și de ieșire, sunt setate anumite intervale. Aceste intervale trebuie verificate.

Pentru anumite metode sunt stabilite mai multe intervale. Acest lucru va fi testat pentru toate aceste intervale.

Toate funcțiile care au un index trebuie să fie testate pentru interval, deoarece acel index are un domeniu bine stabilit.

De obicei, este necesar să verificați :

- valorile inițiale și finale pentru index au aceeași valoare;
- primul element este mai mare sau mai mic decât ultimul element;
- ce se întâmplă dacă indicele este negativ;
- ce se întâmplă dacă indicele este mai mare decât limita superioară;
- numărul de articole nu este același cu cel pe care îl doriți - dimensiunea;
- etc.

REFERENCE

Anumite metode depind de lucrurile externe sau de obiectele externe acestor metode. Aceste elemente trebuie verificate și controlate.

Exemple:

- o aplicație web necesită conectarea utilizatorului;
- o extragere din stivă funcționează dacă există elemente în stivă;
- etc.

Aceste elemente sunt numite precondiții sau condiții preliminare.

Condiții preliminare pentru ca metoda să funcționeze în mod normal.

Aceste teste sunt efectuate folosind dubluri de test (**stub, fake, dummy, mock**).

EXISTENCE

Trebuie să ne întrebăm ce se întâmplă cu metoda dacă un parametru nu există, dacă este nul sau dacă este 0.

De asemenea, pentru sistemele software care funcționează cu fișiere sau cu conexiune la internet, este necesar să se verifice existența acestor fișiere sau disponibilitatea conexiunii la internet. În caz contrar, aplicația nu trebuie să dea eroare, ci trebuie să se comporte normal cu avertizarea utilizatorului de problema întâmpinată.

Este asemănătoare cu condiția de eroare din Right-BICEP.

CARDINALITY

Este similar cu testele de existență (Existence) și testele privind intervalul (Range).

Trebuie să verificăm dacă metoda/lista/colecția are 0 elemente, 1 element sau elemente n.

Dacă funcționează pentru 2, 3 sau 4 elemente, se consideră că va funcționa pentru mai multe elemente, însă nu trebuie să uităm de testul de Boundary superior.

TIME

Este similar cu testul de performanță din **Right-BICEP**.

De asemenea, poate fi testat dacă șablonul de apeluri este respectat. Similar cu design pattern-ul **Template**.

De exemplu, pentru a apela metoda **logout()**, trebuie mai întâi să apelăm metoda de **conectare()**.

F.I.R.S.T.

- **Fast** – testul dezvoltat ar trebui să fie rapid, deoarece dacă avem prea multe teste, nu trebuie să așteptăm prea mult timp când le executăm.
- **Isolated/Independent** – atunci când un test eșuează, acesta ar trebui să fie izolat și să spună exact unde este problema și ce problemă există.
- **Repeatable** – rezultatele obținute ar trebui să fie identice indiferent de numărul rulări ale acestor teste.
- **Self-Validating** – în cazul în care testele trec, dezvoltatorul ar trebui să aibă mare încredere că codul este corect și fără erori. Dacă un test nu reușește să treacă, dezvoltatorul trebuie să aibă încredere în faptul că metoda trebuie îmbunătățită, ci nu să considere ca testul este greșit.
- **Timely** – când trebuie să punem în aplicare testele pentru metoda noastră? Când considerăm că am făcut toate testele?

CODE COVERAGE

(Code lines executed by tests / total code lines) * 100 = code coverage in percent

COMPLEXITATEA CICLOMATICĂ

1 pct. pentru începutul metodei

1 pct. per structura conditională (if)

1 pct. per structura iterativă

1 pct. per case sau bloc default din switch

1 pct. pentru condiții adiționale de tip boolean (&& sau ||)

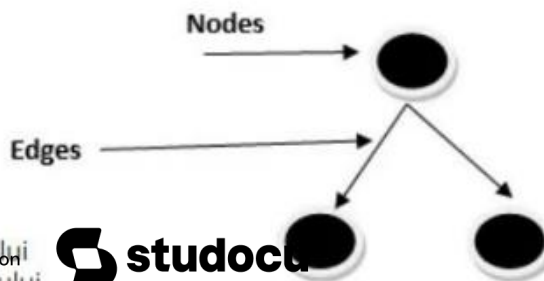
! Se face abstracție de excepțiile aruncate sau prinse (fiecare ar adăuga câte 1 pct), deoarece pot fi daunatoare în cazul unui cod bine scris

Cum se calculează complexitatea ciclomatică?

$$CC = E - N + 2;$$

Unde:

- E => Numărul de muchii (EDGES) ale graficului
- N => Numărul de noduri (NODES) ale graficului



DUBLURI DE TESTARE

O dublură de testare este pur și simplu un alt obiect care se potrivește cu interfața colaboratorului necesar și poate fi trecut în locul său. Există mai multe tipuri de dubluri de testare.

Dummy object – un obiect care respecta interfața dar metodele nu fac nimic sau null.

Stub – Spre deosebire de Dummies, metodele dintr-un Stub vor întoarce răspunsuri conservate sau hardcodate.

Spy – este un Stub care gestionează și contorizează numărul de apeluri.

Fake – este un obiect care se comportă asemănător cu unul real, dar are o versiune simplificată.

Mock – diferit de toate celelalte.

DUMMY OBJECT

= un obiect care respectă interfața, dar metodele nu fac nimic sau returnează 0 sau null.

Când trebuie să folosim obiectul real, de fapt folosim un obiect dummy.

Aceste dubluri sunt folosite atunci când nu trebuie să apelăm metodele din acel obiect (pentru că nu fac nimic). Cu alte cuvinte, **testăm ca o parte de cod nu este executată în condițiile specificate pentru test.**

```
@Test
public void test() {
    Companie company=new Companie("Company", new PersoanaDummy(), 0);

    List<IPersoana> lista=new ArrayList<>();
    lista.add(new PersoanaDummy());
    lista.add(new PersoanaDummy());
    lista.add(new PersoanaDummy());
    company.setSalariati(lista);

    assertEquals(3, company.getNumarSalariati());
}
```

STUB

= metodele de la un Stub vor returna răspunsuri conservate / hardcodate.

În acest fel, putem folosi aceste obiecte cu apeluri reale.

```
public int getVarsta(){
    return 33;
}
```

Se foloseste ca sa ne asiguram ca codul testat are date cunoscute, predictibile cu care opereaza.

```
@Test
public void test_verificareLegalitate() {
    IPersoana persoana=new PersoanaStub("Nume Prenume", "43");
    Companie c=new Companie("Companie",persoana,1000);
    assertTrue(c.verificareLegalitate());
}
```

FAKE

= este un obiect care se comportă ca unul real, dar are o versiune simplificată.

De cele mai multe ori e folosit pentru componente stateful (de ex. un serviciu ce are in spate o baza de date)

De obicei, pentru un fake, putem stabili ce valoare ar trebui să se întoarcă. Nu va fi o valoare hardcodată.

```
@Override
public int getVarsta() {
    return valoareGetVarsta;
}
```

SPY

= este un Stub sau Fake care gestionează și numărul de apeluri realizate pentru metodele acestor obiecte.

```
public int getVarsta() {
    numberGetVarsta++;
    return 33;
}
```

```
public int getVarsta() {
    numberGetVarsta++;
    return valoareGetVarsta;
}
```

Se foloseste atunci cand vrem sa ne asiguram ca, in conditiile de testare, o anumita dependinta este apelata, cu anumiti parametri.

MOCK

= diferit de toate celelalte, dar funcționează similar.

Il folosim atunci cand vrem sa controlam datele returnate de metoda si sa confirmam ca metoda e apelata cu valorile corecte ale parametrilor.



MOCK TESTING

Mock testing – Este utilizat atunci când dorim ca metoda testată să nu fie influențată de referințe externe.

Mock object (obiect mock-uit) – un obiect care simulează comportamentul unui obiect real, însă într-un mod controlat.

Cele mai folosite framework-uri sunt Mockito, EasyMock, etc.

JUnit5

JUnit4	JUnit5 - Jupiter
@BeforeClass	@BeforeAll
@AfterClass	@AfterAll
@Before	@BeforeEach
@After	@AfterEach

assertThrows – pentru testarea condițiilor de eroare.

assertTimeout – pentru testarea timpului de rulare.

@Tag – pentru suitele customizate.

```
@Test
@Tag("Performance")
@Tag("Fast")
```

```
@Test
@Tag("Error")
@Tag("Slow")
```

GRILE

Ce pattern permite incapsularea unui obiect pentru a-i oferi un comportament specific unei alte interfete? **ADAPTER**

Ce pattern permite incapsularea unui obiect in vederea controlarii accesului la acesta? **PROXY**

Ce model permite subclaselor sa decida cum sa implementeze pasii intr-un algoritm? **TEMPLATE**

Ce pattern permite incapsularea unui obiect pentru a-i oferi un comportament specific unei alte interfete? **ADAPTER**

Ce model va permite sa extindeti / modificati functionalitatea obiectului in timpul rularii? **DECORATOR**

Care dintre urmatoarele enunturi descrie modelul **Façade** corect?

Acest model ascunde complexitatea sistemului/modului si ofera clientului o interfata pe care o poate folosi pentru a accesa/utiliza sistemul.

Care dintre urmatoarele enunturi descrie modelul **Flyweight** corect?

Acest model este utilizat in principal pentru a reduce numarul de obiecte create si pentru a reduce amprenta de memorie.

Care dintre urmatoarele enunturi descrie modelul **Composite** corect?

Acest model este utilizat acolo unde trebuie sa tratam un grup de obiecte in mod similar ca un singur obiect.

Este necesara o aplicatie de vizualizare a imaginilor care accepta diferite forme (jpeg, gif, bmp etc.). Aplicatia poate fi extensa pentru a gestiona alte formate in viitor. Care este design pattern-ul care va ajuta sa gestionati efficient dezvoltarea viitoare a aplicatiei:

FACTORY

Care dintre modele sunt structurale?

Adapter, Composite, Decorator, Façade, Flyweight, Proxy.

Dezvolți o soluție software pentru asociațiile de proprietari dintr-un cartier de blocuri. În cadrul cartierului sunt mai multe meseriași (instalatori, electricieni, etc) care repară și îndreaptă lucrurile din tot cartierul. Atunci când apare o problemă, locatarii solicită intervenția acestora și solicitarea lor este salvată într-o coadă de așteptare, iar atunci când unul dintre meseriașii cartierului este liber, va prelua această problemă și o va rezolva. Să se implementeze modulul care permite gestionarea cozii de comenzi de probleme către meseriașii cartierului.

De asemenea, se dorește implementarea unei soluții care să permită locatarilor unui bloc să aleagă modul în care sunt publicate anunțurile din bloc.

Asociația de proprietari poate decide în cadrul ședințelor anuale ca anunțurile să fie printate la avizier sau să fie trimise prin email sau prin sms, etc. Odată soluția aleasă ea va fi menținută până la ședința următoare.

Care sunt cele 2 design patternuri care rezolvă optim aceste 2 probleme?

COMMAND – locatarii solicita interventia meseriasilor fara a specifica ce problema trebuie rezolvata;

– solicitarile sunt intr-o coada de asteptare, deci coada respectiva se ocupa cu gestiunea comenzilor (Invoker/Operator).

STRATEGY – sedinta anuala este pe post de runtime;

– problema care trebuie rezolvata e modul/strategia de publicare a anunturilor din bloc (printate la avizier, trimise pe mail sau prin sms etc.)

Dezvolți o soluție software pentru asociațiile de proprietari dintr-un cartier de blocuri care să le permită să gestioneze detaliile legate de costurile de întreținere aferente unei luni pentru fiecare proprietar de apartament. Aceste date (numărul apartamentului, costul apei calde, costul căldurii, costul gazelor, etc.) sunt stocate în fiecare lună. Soluția trebuie să permită salvarea acestor date la finalul lunii astfel încât să poată oferi în orice moment informații cu privire la istoricul acestor costuri pe fiecare apartament.

Independent de cerința anterioară, blocul are la intrare un interfon prin care este solicitat/controlat accesul. Deoarece copiii din cartier abuzează de acest serviciu și deranjează locatarii, se dorește adăugarea în sistem a unei camere video care să verifice dacă apelantul este adult sau copil. Această camera va fi conectată la sistemul existent (nu se modifica interfonul) și va controla dacă se poate fi folosit interfonul sau nu. Dacă camera detectează un copil atunci interfonul nu va suna la apartamentul apelat.

Care sunt cele 2 design patternuri care rezolva optim aceste 2 probleme?

MEMENTO – se dorește salvarea datelor la finalul lunii și posibilitatea de a vedea istoricul

PROXY – interfon care controlează accesul pentru intrarea în bloc, camera video care controlează accesul la interfon

Dezvolți o soluție software pentru asociația de proprietari dintr-un bloc care să permită gestiunea eficientă a evenimentelor ce pot apărea, evenimente care necesită luarea unor măsuri adecvate și care să protejeze viața locatarilor. Soluția procesează diferite mesaje/ evenimente primite de la senzori/sisteme din bloc și va lua următoarele decizii

- dacă s-a declanșat alarma de incendiu atunci este notificat serviciul 112
- dacă este anunțată o inundație atunci este notificat administratorul
- dacă este doar o informare atunci sunt notificați locatarii
- (opțional) pentru alarmele de incendiu și inundație sunt anunțați și locatarii

Independent de implementarea scenariului anterior găsiți o soluție care să permită integrarea sistemului existent de procesare a alertelor (cu interfața IProcesareEveniment) în sistemul național de alertare ce este construit în jurul interfeței IAlertarePublica.

Care sunt cele 2 design patternuri care rezolva optim aceste 2 probleme?

OBSERVER – trebuie notificați mai multe parti (112, admin, locatari)

ADAPTER – trebuie adaptat sistemul existent de procesare a alertelor în sistemul național de alertare (cele două nu au interfața comună)

– nu se adaugă funcționalități

Dezvoltă o soluție software pentru asociațiile de proprietari dintr-un cartier. Fiecare bloc are locatari, are un administrator și fiecare scară de bloc are un președinte de scară. Președinții de scară se ocupă doar de problemele locatarilor de pe scara sa, iar administratorii gestionează toate scările unui bloc. Administratorii la rândul lor vor raporta unui reprezentant al primăriei aceste probleme. Să se implementeze modulul care permite realizarea unei structuri pentru reprezentarea ierarhică a locatarilor, președinților de scara și a administratorilor de bloc.

În cadrul soluției un administrator trebuie să poată anunța toți locatarii din bloc atunci când afișează la avizier costurile cu întreținerea. Să se realizeze un modul care să permită administratorului să gestioneze eficient o listă de locatari pe care să-i anunțe automat atunci când face acest lucru.

Care sunt cele 2 design patternuri care rezolvă optim aceste 2 probleme?

COMPOSITE – avem compunere de obiecte (locatari, scări dintr-un bloc, administrator)

– avem structura arborescentă/ierarhică (exemplu: adminii raportează unui reprezentant al primăriei)

OBSERVER – adminul gestionează lista de locatari și îi anunță automat

ACME Inc dezvoltă o soluție software pentru un restaurant, astfel încât ospătarul să poată lua comenzi direct pe telefonul mobil.

Comenzile sunt preluate de la client și sunt create la fața locului, fiind repartizat automat bucătarul specializat pe acel fel de mâncare, fiind definite ingredientele utilizate și alte cerințe speciale ale clientului.

Aceste detalii sunt furnizate de aplicație, fără a fi necesară intervenția chelnerului care selectează doar produsul necesar. Comenzile sunt trimise bucătăriei la finalul comenzii pentru masa respectivă și vor fi executate în funcție de gradul de încărcare al fiecărui bucătar. Ce model oferă o soluție optimă la această problemă?

COMMAND – chelnerul nu trebuie să știe concret ce presupune comenzile.

În contextul JUnit, obiectivul este de a scrie

teste care esuează pentru a corecta erorile din timp

teste care contin toate cazurile particulare ale metodei testate

Urmatoarea secventa de cod incalca principiul:

SRP – Single Responsibility (pentru ca face mai multe lucruri odata)

```
class Prelucrari {  
    public void afiseazaUtilizator(){}  
    public void prelucrareData(){}  
    public Boolean salveazaConstanteInDb(){}  
}
```

Implementați clasa Product care conține atributul cantitate Atributul acceptă valon între 5 și 100 (inclusiv aceste 2 valori).

Pentru a permite clienților să modifice cantitatea de produs, implementați și metoda setQuantity (int value) Metoda ar trebui să valideze valoarea de intrare

Dacă implementați următorul test. ce valori ar trebui să folosiți pentru „X” dacă testul este unul de tip

BOUNDARY=>INTERVAL [5,100] => 5 si 100

RANGE=> 4, 5, 6, 99, 100

RIGHT=> 5, 6, 99, 100

```
@Test  
public void testSetQuantity() {  
    Product product = new Product();  
  
    int testValue = X;  
    product.setQuantity(testValue);  
    int obtainedValue = product.getQuantity();  
  
    assertEquals(testValue, obtainedValue);  
}
```

Ce model GoF trebuie implementat daca doriti sa implementati o solutie care sa permita clientului sa aleaga in timpul rularii algoritmul / functia necesara procesarii unui set de date. Solutia trebuie sa permita modificarea bibliotecii de functii, dar nu a clasei care gestioneaza datele.

STRATEGY

Dezvoltă o soluție online pentru o companie care oferă spre închiriere clienților săi diferite categorii de vehicule mașini, motociclete și autoutilitare

Datele despre structura flotei sunt stocate în mod unic la nivelul soluției și pot fi solicitate de clienți

Clienții pot verifica detaliile pentru fiecare vehicul. Componenta care furnizează această descriere prezintă pentru fiecare vehicul selectat detalii în funcție de starea în care poate fi,

defect, disponibil sau închiriat. Ce modele de proiectare (unul sau mai multe) sunt potrivite pentru a fi utilizate pentru a implementa scenariul prezentat.

FACTORY – trebuie create obiecte de tip masina, motocicleta, autoutilitara

SINGLETON – mod unic de stocare la nivelul solutiei

STATE – trebuie prezentate detaliile pt fiecare vehicul in functie de starea in care poate fi

Ce afirmatii descriu cel mai bine conceptul de “spaghetti-code”?

Este cunoscut si ca *big ball of mud*

Poate fi evitat folosind design pattern-uri

Ce design pattern incapsuleaza un request/task intr-un obiect ce va fi executat mai tarziu?

COMMAND

Pentru urmatoarea metoda, ce afirmatie este corecta pentru a obtine 100% code coverage prin testarea ei?

```
public int aSimpleMethod(int value) throws Exception
{
    int result = 0;
    switch(value) {
        case 1:
            result = 10;
            break;
        case 5:
            result = 20;
            break;
        case 10:
            result = 30;
            break;
        default:
            throw new Exception("Wrong value");
    }
    return result;
}
```

4 teste

Care sunt principiile Clean Code folosite aici?

```
private boolean perfectSquare(int number) {  
    return ((int) Math.sqrt(number)) * ((int) Math.sqrt(number)) == number;  
}  
  
public boolean isFibonacciNumber(int number) {  
    int val = 5 * number * number;  
    return perfectSquare(val - 4) || perfectSquare(val + 4);  
}
```

DRY (Don't Repeat Yourself) – nu se repeat nicaieri cod scris, ci se foloseste metoda deja implementata

SRP (Single Responsibility) – implementarea face un singur lucru (verifica daca nr e Fibonacci sau nu)

KISS (Keep It Simple, Stupid) – se vede ca e simplu codul

LSP (Liskov Substitution) – nu se aplica, pentru ca nu avem clase

OCP (Open-Closed) – nu se aplica

WET (We Enjoy Typing) – e antiprincipiu, opusul lui DRY

Fiind date semnaturile unor functii ce fac diferite prelucrari, indicate pentru care dintre acestea se justifica teste de tip Ordering?

int calcul(ArrayList<Integer> valori)

int calcul(int[] valori)

(testele de ordine se fac pe liste)

Dezvolți o soluție software de tip trading care sa permită clienților să cumpere acțiuni pe diferite burse prin intermediul unor brokeri. Clienții pot comanda cumpărarea sau vanzarea unui număr de acțiuni pentru anumite companii listate la bursa. Clienții își aleg brokerul în momentul în care lansează comanda în sistem. Comenzile sunt înregistrate în sistem și vor fi procesate în ordinea în care au fost generate de platforma de trading în functie de gradul de încărcare al platformei. În momentul în care comanda este procesata ea va fi executata de broker.

În timp platforma se va extinde și vor fi oferite diferite tipuri de servicii financiare schimb valutar, vanzare sau cumpărare de crypto monede, investitii în fonduri cumpărare de bonuri de trezorerie Aceasta dezvoltare trebuie susținuta de o soluție care sa permită extinderea tipurilor de servicii.

Care sunt cele 2 design patternuri care rezolva optim aceste 2 probleme ?

COMMAND – comenzile sunt înregistrate și vor fi procesate în ordinea în care au fost generate de platforma, apoi vor fi executate de broker (invoker-ul este platforma)

FACTORY – familie de servicii financiare

Dezvoltă o soluție software/plugin care să permită verificarea fișierelor descărcate de pe Internet și a link-urilor Web accesate de utilizatori prin intermediul browser-ului. Această soluție este bazată pe module ce vor verifica acțiunile utilizatorilor module ce au interfața IAntivirus (enumerarea se poate muta într-o clasă separată). Modulele vor face o verificare completă a acțiunii utilizatorului conform următoarelor condiții

- dacă fișierul descărcat este de tip exe (dacă numele fișierului conține exe) se verifică dacă conține viruși dacă download-ul este întrerupt (se afișează mesaj), altfel se trece la următoarea verificare.
- dacă fișierul descărcat este de tip pdf se verifică dacă conține malware. dacă da atunci download-ul este întrerupt (se afișează mesaj) altfel se trece la următoarea verificare;
- dacă se accesează un link se verifică dacă este unul de tip https dacă nu este atunci este alertat utilizatorul printr-un mesaj; în ambele situații link-ul este accesat;
- ultimul modul va accesa link-ul sau va descărca fișierul și va consemna această acțiune în istoricul acțiunilor (istoricul se poate simula printr-o colecție standard ce este gestionată de acest modul)

În timp trebuie să se găsească o soluție prin care un modul al plugin-ului să poată fi modificat/customizat în timp real prin adăugarea de setări noi dorite de utilizatori. Această soluție trebuie să permită utilizatorului să modifice modulul în timpul execuției plugin-ului prin adăugarea de setări noi. Acestea vor afecta și modul în care funcționează modulul respectiv verificarea acțiunii utilizatorilor. Modulele pot fi modificate doar dacă utilizatorii doresc, altfel modulul va funcționa normal.

Care sunt cele 2 design pattern-uri care rezolvă optim aceste 2 probleme?

CHAIN OF RESPONSIBILITY – dacă fișierul/linkul finalizează o verificare și este safe, se trece la următoarea verificare

DECORATOR – adăugare de funcționalități la runtime

Dacă se consideră funcția următoare, indicate cu ce valoare se poate implementa un test de tip Existence pentru care este așteptată și o excepție?

```
public float prelucrareValori(ArrayList<Integer> valori){  
    float rezultat = 0;  
  
    for(int valoare : valori) {  
        if(valoare %2 == 0)  
            rezultat += valoare;  
    }  
  
    return rezultat;  
}
```

ArrayList<Integer> input = null;

Care dintre urmatoarele design pattern-uri implementeaza principiul Hollywood – Don't Call Us, We'll Call You ?

OBSERVER(zice google), TEMPLATE(zice pdf cu grille)

HOLLYWOOD PRINCIPLE = Inversion of Control – consta in a raspunde la evenimente din exterior. E strans legat de DIP.

Ce presupune o clasa care foloseste varianta Eager Initialization?

Presupune initializarea instantei inca de la momentul declararii in clasa

Va exista o metoda in interiorul clasei care va returna mereu instanta creata la momentul declararii

Constructorul clasei este privat

Ce reprezinta Test Case?

Clasa ce defineste setul de obiecte pentru a rula mai multe teste

Care dintre urmatoarele adnotari pentru structura unui test nu este prezenta in JUnit 4?

AfterEach

Ce design pattern permite obiectului să fie notificat atunci când are loc un eveniment?

Observer

Ce design pattern permite sa construiesi un obiect in pasi?

Builder

Ce design pattern permite unui client sa creeze familii de obiecte fara sa specifice clase concrete?

Factory(toate tipurile)

Ce design pattern încapsulează comportamentele state-dependent și folosește delegarea pentru a comuta între ele?

State

Ce design pattern permite definirea de dependente între obiecte, când un obiect își schimbă starea, toate dependentele sunt notificate?

Observer

Ce design pattern oferă abilitatea să restabilească un obiect după starea precedentă?

Memento

Ce design pattern încapsulează un request/ task ca obiect, care va fi executat mai târziu?

Command

Ai o clasă care acceptă și returnează valori în unități British (feet, miles etc) dar trebuie folosit pentru unități metrice. Care design pattern ar rezolva problema?

adapter

Dezvolti o soluție software care permite călătorilor să aleagă eficient o combinație de mijloace de transport care să le permită să ajungă la destinație. Soluția va sugera o soluție în funcție de preferințele lor (ceea ce înseamnă că preferă. Dacă doresc o călătorie directă. Dacă au restricții de cost etc.) și disponibilitatea diferitelor mijloace de transport în apropiere. Orașul este împărțit în zone de acces, așa că dacă un călător dorește să călătorească în aceeași zonă (maximum 5 km) sugerează autobuzul. dacă vrea să călătorească în zona 2 (maximum 10 km) sugerează tramvaiul. dacă dorește să călătorească în zona 3, folosește metroul și dacă dorește să călătorească în afara orașului, ar trebui să folosească trenul. Implementați soluția care permite pasagerilor să aleagă mijloacele de transport public adecvate pe baza criteriilor lor.

Soluția software sugerează o soluție de fiecare dată când un mijloc de transport intră în zona de proximitate a pasagerilor. Deocamdată soluția permite pasagerilor să plătească biletele

online. Deoarece anumite mijloace de transport (de exemplu, metroul) permit plata călătoriei numai prin intermediul lucrătorilor fizici, soluția trebuie să permită în viitor plăți precum funcția NFC sau bluetooth a telefonului mobil, iar operatorii pot oferi interfața specifică vameșilor.

Care sunt cele 2 modele de proiectare care rezolvă în mod optim aceste 2 probleme?

CHAIN OF RESPONSIBILITY

ADAPTER

Dezvolți o soluție software care permite implementarea unei bariere autonome care tratează diferit situațiile specifice unei treceri de cale ferată. Bariera nu este controlată din exterior. În funcție de starea unor senzori, bariera implementează acțiuni specifice. Bariera este ridicată în mod normal și pornește avertizarea sonoră atunci când se identifică apropierea trenului. După 10 secunde bariera scade automat. De asemenea, mesajele afișate de barieră diferă în funcție de starea barierei. Pentru a evita situațiile în care bariera nu este ridicată după ce trenul a trecut sau este ridicată chiar dacă trenul este încă în zonă, se dorește adăugarea la sistem a unui senzor extern pentru a efectua verificări suplimentare.

Senzorul existent va notifica noul senzor în loc să transmită informațiile direct la barieră, dar soluția propusă nu ar trebui să implice modificarea sistemului actual. Găsiți o soluție care vă permite să adăugați noua componentă, dar să mențineți soluția curentă în uz.

Care sunt cele 2 modele de proiectare care rezolvă în mod optim aceste 2 probleme?

STATE

PROXY

Dezvolți o soluție software care permite clienților să comande alimente prin servicii similare Glovo / FoodPanda. Clienții generează o comandă cu produsele respective. Comanda va fi preluată de un curier disponibil în sistem. Comanda va fi livrată în viitor, dar timpul de livrare depinde de mai mulți factori externi (trafic. Aglomerație în magazin etc.). Comenzile înregistrate în sistem nu mai pot fi anulate și nu se pierd.

În acest moment, fiecare client poate include în comandă orice fel de produse, dar soluția trebuie să permită adăugarea de filtre / reguli în viitor (nu permite comenzile mai mici de o anumită sumă, nu permite comenzile pentru anumite produse. Adaugă costul de transport la comandă sau crește prețurile restaurantelor cu o sumă) care limitează / controlează opțiunile clienților. Aceste modificări trebuie incluse în serviciile back-end atunci când clientul adaugă un produs.

Care sunt cele 2 modele de proiectare care rezolvă în mod optim aceste 2 probleme?

COMMAND

PROXY

Dezvoltați o soluție software care permite gestionarea diferiților clienți de e-mail. Clienții de e-mail trebuie să poată trimite și primi e-mailuri pentru o adresă de e-mail specifică, dar pot fi folosiți și pentru gestionarea grupurilor de e-mail compuse din mai multe adrese de e-mail. Utilizatorii ar trebui să poată gestiona această structură din soluția lor. Utilizatorii au posibilitatea de a modifica / personaliza clientul de e-mail în timpul utilizării. Soluția propusă trebuie să permită clienților să modifice / personalizeze soluția prin adăugarea de filtre sau criterii de sortare a e-mailurilor. De asemenea, funcția de transmisie a unui mesaj poate fi modificată prin adăugarea unei confirmări suplimentare sau prin întârzierea trimiterii sau specificarea orelor specifice.

Care sunt cele 2 modele de proiectare care rezolvă optim? aceste 2 probleme?

COMPOSITE

DECORATOR

Dezvoltați o soluție software pentru un echipament. de la un laborator medical, echipament care este utilizat pentru automatizarea completă a procesului de efectuare a analizelor de sânge. mesaje specifice stării echipamentului medical (preluarea probei, încărcarea eprubetei, procesarea datelor, afișarea rezultatelor etc.) La pornire, echipamentul este în așteptare, dar din momentul încărcării unei eprubete , echipamentul trece printr-o fază de la sine până când rezultatul este afișat. Afișajul arată și numele echipamentului. Mesajele sunt afișate imediat ce echipamentul trece la o nouă stare. Implementarea modulului Mat permite managementul eficient sau echipamentele medicale, având în vedere că este autonom și trece prin mai multe stări / faze pentru a afișa rezultatul final.

De asemenea, se dorește ca software-ul dispozitivului să poată gestiona eficient diferite tipuri de teste medicale. care derivă din familia de clase MedicaLAnalysis. Soluția trebuie să genereze în mod eficient diferite tipuri de analize (Test-COVID. Colesterol, vitamine) fiind decuplate de detaliile creării acestor tipuri. Peste orar. detaliile acestor tipuri de teste medicale se pot modifica. dar această soluție nu ar trebui să fie afectată.

STATE

FACTORY

Dezvoltați o soluție software utilizată de un laborator medical pentru a gestiona diferite solicitări de analiză a probelor de material biologic (teste de sânge) Laboratorul primește cererile de la diferite cabinete medicale și le salvează într-o coadă. Analizele sunt efectuate de diferite departamente medicale care au echipamente speciale pentru fiecare tip de analiză. Laboratoarele solicită aceste teste indicând tipul lor și numele pacienților. Atunci când o cerere de analiză este înregistrată în sistem, aceasta este asociată cu departamentul specializat în acel tip de analiză Odată ce rezultatul analizei este obținut, departamentul care a procesat-o va trimite rezultatul către e-mailul pacienților (e-mailul pacienților și numele sunt înregistrate în cerere). De asemenea, se dorește ca în soluția software de laborator. instanța de manager responsabilă de coada comenzilor sau lista de clienți ar trebui să fie unică, astfel încât să nu genereze probleme legate de modul în care sunt gestionați clienții testelor medicale.

COMMAND

SINGLETON

Dezvolti o soluție software care să permită gestionarea eficientă a situațiilor școlare pentru mii de elevi din același profil. Principala problemă a clientului este reducerea spațiului de memorie ocupat de aceste date în RAM. Soluția dvs. trebuie să permită acest lucru având în vedere că informațiile (fișa disciplinei, obiectivele, subiectele etc.) referitoare la cursurile urmate de studenți sunt comune. Soluția trebuie să permită și susținerea examenelor. Pot fi orale, pe bază de test, cu întrebări deschise, care necesită încărcarea fișierelor etc. Modul de susținere a examenelor va fi stabilit de către profesorul coordonator al cursului, indiferent de opțiunile altor profesori care coordonează același tip de curs.

Care sunt cele 2 modele de proiectare care rezolvă în mod optim aceste 2 probleme?

FLYWEIGHT

STRATEGY

Fiind data functia urmatoare, care este numarul minim de teste unitare ce trebuie implementate pentru a asigura un code coverage de 100%? **R: 3 cica**

```
public int calcul(int valoare){
    int rezultat = 0;

    if(valoare > 0)
        rezultat = 100;
    else
        rezultat = 200;

    if(valoare > 100)
        rezultat += 5;
    else
        rezultat += 10;

    return rezultat;
}
```

Data fiind urmatoarea clasa, selecteaza afirmatia (una sau mai multe) care o descrie corect.

```
public class DbConnection {
    String ip;
    String details;

    public final static DbConnection conexiune;

    static {
        Settings settings = new Settings();
        conexiune = new DbConnection(settings.getIp(), settings.getDetalii());
    }

    private DbConnection(String ip, String detalii) {
        super();
        this.ip = ip;
        this.details = detalii;
    }
}
```

Este un exemplu de Eager Initialization

Este o implementare corecta de Singleton

Fiind data functia pentru calcularea pretului unui bilet, indica numarul minim de teste (cu combinatii diferite de valori) care trebuie sa implementeze o acoperire a codului 100%?

```
public float getTicketPrice(int area, float basePrice, boolean groupReduction){
    float finalPrice = 0;
    switch(area){
        case 1:
            finalPrice = basePrice;
        case 2:
            finalPrice = basePrice * 1.2f;
            break;
        case 3:
            finalPrice = basePrice * 1.5f;
            break;
    }

    if(groupReduction)
        finalPrice = (float) (0.85 * finalPrice);

    return finalPrice;
}
```

R: 6

Considerand urmatorul TestCase in Junit4. Ce afiseaza dupa executie?

```
public class TestCaseExamen {

    @Before
    public void before () { System.out.print("Before ");}

    @Test
    public void test1() { System.out.print("Test1 ");}

    @Test
    public void test2() { System.out.print("Test2 ");}

    @AfterClass
    public static void after() { System.out.print("After ");}

}
```

Before Test1 Before Test2 After

Care dintre variantele de cod reprezinta implementarea design pattern-ului Singleton?

```
public class PrintManager1 {
    public PrintManager1 INSTANCE = new PrintManager1();
    public PrintManager1() {}
    public static PrintManager1 getInstance() {
        return INSTANCE;
    }
}

public class PrintManager2 {
    private PrintManager2 INSTANCE = new PrintManager2();
    private PrintManager2() {}
    public static PrintManager2 getInstance() {
        return INSTANCE;
    }
}

public class PrintManager3 {
    private static final PrintManager3 INSTANCE = new PrintManager3();
    private PrintManager3() {}
    public static PrintManager3 getInstance() {
        return INSTANCE;
    }
}

public class PrintManager4 {
    private final PrintManager3 INSTANCE = new PrintManager4();
    private PrintManager4() {}
    public PrintManager4 getInstance() {
        return INSTANCE;
    }
}
```

PrintManager3

Dezvolti o solutie software pentru editarea imaginilor de tip Bitmap destinată copiilor. Solutia trebuie sa le permită utilizatorilor sa vadă istoricul modificărilor si să le permită revenirea la un moment In timp ales de ei. Imaginile ce pot fi procesate in cadrul aplicatiei sunt de tip Image. Solutia software trebuie să permită stocarea salvărilor și trebuie sa permită utilizatorului să le vadă si să le recupereze in functie de data salvării. In timp solutia aceasta este achizitionată de o altă companie care are un produs similar dar care permite procesarea de imagini de tip PhotoImage. Ca parte a vanzării trebuie găsită o solutie care să permită utilizarea imaginilor Bitmap in noul context.

Care sunt cele 2 design pattern-uri care rezolva optim aceste 2 probleme?

ADAPTER

MEMENTO

Cand este aplicabil principiul **KISS**?

Ori de cate ori vrem ca o metoda sa faca de toate

Ce reprezinta **One screen rule**?

Evitarea metodelor foarte lungi (peste 20 linii de cod)

Pentru metoda calculValoare(int[] valori) ce proceseaza un vector de numere, care dintre urmatoarele combinatii de teste continuate teste de **Cardinalitate**

1. **Test pentru vectorul cu 0 elemente**
 2. Test cand referinta valorii este null
 3. **Test pentru vectorul {10}**
 4. **Test pentru vectorul {10,20}**
 5. **Test pentru vectorul {10,20,30}**
 6. **Test pentru vectorul {10,20,30,40,50}**
-

Realizarea unui test de **Cross-check** presupune:

Utilizarea unor metode asemanatoare cu metoda testata pentru a obtine valoarea asteptata

Urmatoarea ierarhie de clasa aplica principiul:

```
interface Redimensionabil {
```

```
// ...
```

```
}
```

```
class EcranMare implements Redimensionabil{
```

```
// ...
}
class EcranMic implements Redimensionabil{
// ...
}
```

Design by contract = Liskov substitution = LSP

Ce design pattern poate fi folosit pentru a implementa **Lazy Loading**?

COMMAND

Daca vrei sa iti testezi propria functie Fibonacci (bazata pe algoritmul $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$) utilizand urmatoarea functie, ce strategie de test folosesti?

```
public int getFibonacciNumber(int n) {
    double phi = 1.61803;
    int val1 = (int) Math.pow(phi, n);
    int val2 = (int) Math.pow(-1, n);
    int val3 = (int) Math.pow(phi, n);

    return ((int)(val1 - (val2 / val3)) / (int)Math.sqrt(5));
}
```

CROSS-CHECK

Dacă efectuăm un test pentru metoda `getFibonacciNumber ()` cu valoarea 2, efectuăm un test de tip

RIGHT

Pentru un test de tip RIGHT test, daca apelam metoda urmatoare cu valoarea 4 (unde 4 reprezinta un index), asteptam sa primim valoarea ?

Index = 4

Fibonacci – 0, 1, 1, 2, **3**, 5, 8, 13, 21, 34

=> al 5-lea element = index 4

=> **R: 3**

Pentru un test de conditii de eroare (Error conditions) pentru metoda `isFibonacciNumber()`, trebuie sa utilizam valoarea:

Val = $5*0*0 = 0$

perfectSquare 0-4 = -4 da error

=> **R: 0**

Selectati una sau mai multe optiuni care reprezinta diferentele dintre **Prototype** si **Flyweight**

Prototype optimizeaza viteza de crearea a obiectelor si Flyweight optimizeaza memoria necesara pentru stocarea obiectelor

Prototype este creational si Flyweight este structural

Folosind Prototype, crearea obiectelor se face prin clonare, iar in Flyweight obiectele sunt reutilizate

Un **design pattern** reprezinta

O solutie pentru o problema comuna in POO

Trebuie sa testezi o functie care determina minimul dintr-un array de valori de tipul int[]. Pentru a efectua unit tests (diferite unul de celalalt) de tipul Cardinality = 2 si Ordering, ce date vei folosi?

Pentru **ORDERING** trebuie sa aiba o ordine (cresc/descresc) => {7,8,9,10} , {10,9,6,4}

Pentru **CARDINALITY = 2** => {9,10}

Ce linii contin "Numere magice"?

A screenshot of a code editor showing C# code. The code is as follows:
32 var sum = Sum(1, 5, 6, 7, 10);
33 sum += 1;
34
35 for (int i = 0; i < 5; i++)
36 {
37 Console.WriteLine(\$"{i}, {sum * i}");
38 }

32 – nu e sigur pentru ce sunt 1, 5, 6, 7, 10

33 – nu e sigur ce este 1 (putea fi sum++ in loc de sum+=1)

35 – loop counter 5

Daca doresti sa impleentezi o solutie unde toti clientii care folosesc clasa A vor avea access la aceeasi instanta de tipul A, ce pattern GoF vei folosi?

Singleton este implementat pentru Clasa A

Care este un dezavantaj al **Singleton**?

In unele situatii, un blocaj al comunicarii sau accesului la resurse

Care sunt cateva dintre avantajele testarii unitare?

Poate imbunatati design-ul codului, mai ales daca folosim Test Driven Development (TDD)

Reduce nivelul de bug-uri in codul de productie

Usureaza schimbarea/modificarea codului

NU este principiu de tip **CORRECT Boundary Conditions** specific analizei datelor folosite in Unit Testing?

Valoarea are tipul cerut de tipul variabilei

Accepting Testing (Testare de Acceptanta), in contextual de testare a solutiei, face referire la:

testarea pe baza testelor realizate de client

```
public class GeneratorStudenti {  
    private static HashMap<Integer, IStudent> studenti = new HashMap<Integer, IStudent>();  
  
    public GeneratorStudenti() {  
        studenti = new HashMap<Integer, IStudent>();  
    }  
  
    public static IStudent getStudent(int nrMatricol, String nume) {  
        IStudent student=studenti.get(nume);  
  
        if(student==null) {  
            student=new Student(nrMatricol, nume);  
            studenti.put(nrMatricol,student);  
        }  
        return student;  
    }  
}
```

Nu poate fi considerate o implementare a unui Design Pattern

Este un exemplu de lazy instantiation pentru crearea studentilor

Dezvolti o soluție software pentru editarea imaginilor de tip Bitmap destinată copiilor. Soluția trebuie sa le permită utilizatorilor sa vadă istoricul modificărilor si să e permită

revenirea la un moment. În timp ales de ei, Imaginile ce pot fi procesate în cadrul aplicației sunt de tip imagine Bitmap. Soluția software trebuie să permită stocarea salvărilor și trebuie să permită utilizatorului să le vadă și să le recupereze în funcție de data salvării.

În timp soluția aceasta este achiziționată de o altă companie care are un produs similar dar care permite procesarea de imagini de tip PhotoImage. Ca parte a vânzării trebuie găsită o soluție care să permită utilizarea imaginilor Bitmap în noul context.

Care sunt cele 2 design pattern-uri care rezolvă optim aceste 2 probleme?

MEMENTO – trebuie să permită utilizatorilor să vadă istoricul și să permită revenirea

ADAPTER – folosirea imaginilor Bitmap în contextual procesării imaginilor PhotoImage, cele două nu au interfața comună și doar trebuie să se adauge clasele noi, fără modificări (adapter)

Dezvoltă o aplicație software destinată unei companii care realizează instalarea de geamuri Termopan pentru clienți

Pentru montarea ferestrelor termopan unei clădiri se urmează un set prestabilit de etape cunoscute de către membrii echipei de instalare. Să se implementeze un modul ce oferă posibilitatea de a simula montarea de geamuri termopan mai multor tipuri de clădiri ținând cont de faptul că modul de montare a acestora este același indiferent de tipul de clădire iar pașii parcurși sunt aceiași.

Managerul lucrării furnizează în fiecare dimineață comenzile către echipele formate. În timpul zilei aceștia trebuie să execute toate comenzile primite de dimineață. Să se implementeze modulul care permite trimiterea de comenzi de către managerul lucrării către echipele de lucru. Comenzile sunt gestionate de către șeful de echipă.

Care sunt cele 2 design patternuri care rezolvă optim aceste 2 probleme?

TEMPLATE – set prestabilit de etape cunoscute

COMMAND – comenzile sunt trimise către manager, gestionate de șeful de echipă (invoker) și efectuate de echipe

Dezvoltă o aplicație software pentru un restaurant. Restaurantul are mai mulți Șefi de sală care au în subordine ospătari. Ospătarii nu au nicio persoană în subordine. Se dorește implementarea unui modul pentru reprezentarea diagramei organizaționale a ospătarilor și a șefilor de sală.

În cadrul restaurantului există un singur Chef care gătește toate felurile de mâncare. Trebuie implementată clasa Chef, astfel încât să existe unul singur în restaurant la un moment dat, și care să primească toate feluri de mâncare de gătit. În cazul în care sunt

mai mulți clienți la un moment dat în restaurant, toate felurile de mâncare pe care trebuie să le gătească Chef, vor fi ordonate într-o coadă de așteptare.

Care sunt cele 3 design patternuri care rezolvă optim aceste probleme?

COMPOSITE – ierarhia sefi de sala – ospatari (sefi de sala au în subordine ospatari care nu au subordonați)

SINGLETON – chef-ul este unic

COMMAND – comenzile sunt ordonate într-o coadă de așteptare până când invoker-ul Chef le gestionează

Dezvoltați o soluție software bazată pe 2 microservicii ce au rolul de a gestiona repartizarea locurilor în camin pentru o universitate cu foarte mulți studenți. Soluția trebuie să proceseze un număr mare de solicitări într-un timp scurt și din acest motiv se implementează un mecanism asincron de prelucrare a solicitărilor studenților. Un microserviciu are rolul de a valida și înregistra cererile studenților, ele urmând a fi stocate și procesate printr-un mecanism de tip FIFO. Deoarece volumul mare de cereri nu permite procesarea lor pe același microserviciu, detaliile cererilor sunt încapsulate și transferate pe un alt microserviciu unde sunt și executate.

Care sunt cele 2 design patternuri care rezolvă optim aceste 2 probleme?

PROTOTYPE – nr mare de solicitări într-un timp scurt; solicitările seamănă între ele teoretic

COMMAND – mecanism de tip FIFO (listă); cererile sunt trimise către alt microserviciu (invoker)

Primăria municipiului București dorește implementarea unei aplicații software pentru gestiunea tuturor comercianților din cadrul orașului. Comercianții/magazinele sunt grupate la nivel de primărie de sector, urmând ca sectoarele să aparțină primăriei generale. Să se implementeze o modalitate de stocare a tuturor comercianților și a primăriilor din cadrul cărora acestea aparțin în scopul reprezentării ierarhice generale gestionate de primăria municipiului.

Existând mai multe tipuri de comercianți: magazine alimentare, benzinarii, restaurante, etc., să se implementeze un modul de creare de obiecte de tipul Comerciant. Modulul trebuie să ofere posibilitatea extinderii cu noi tipuri de comercianți ținând cont de principiile Solid de implementare a aplicațiilor.

Care sunt cele 2 design patternuri care rezolvă optim aceste 2 probleme?

COMPOSITE – ierarhie primăria generală – primărie sector - comercianți

FACTORY – mai multe obiecte de tipul Comerciant (alimentare, benzinarii, restaurant); posibilitatea de extindere cu noi tipuri de comercianți; respecta principiile SOLID

Testul unitar de mai jos poate fi un test de tip?

```
@Test(timeout = 1000)
public void testCalculeazaMedie() {
    Student s = new Student();
    double result = s.calculeazaMedie();
    assertEquals(0, result, 0.01);
}
```

PERFORMANCE – timeout = 1000

TIME – seamana cu Performance

Ce Design Pattern este implementat corect si complet in clasa de mai jos?

```
public class DesignPattern implements Cloneable {
    private static DesignPattern field;

    public static DesignPattern getField() {
        if(field == null) {
            field = new DesignPattern();
        }
        return field;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        return field;
    }
}
```

NICIUNUL

```
public double teoremaLuiPitagora(int cateta1, int cateta2) {
    return Math.sqrt(cateta1 * cateta1 + cateta2 * cateta2);
}
```

```
@Test
public void test1() {
    int cat1=3;
    int cat2=4;
    double ipotenuza=obiect.teoremaLuiPitagora(cat1,cat2);
    assertEquals(cat1, Math.sqrt(Math.pow(ipotenuza,2)-Math.pow(cat2,2)), delta: 0.01);
}

@Test
public void test2() {
    int cat1=3;
    int cat2=4;
    double ipotenuza=obiect.teoremaLuiPitagora(cat1,cat2);
    assertEquals(ipotenuza, Math.sqrt(Math.pow(cat1,2)+Math.pow(cat2,2)), delta: 0.01);
}

@Test
public void test3() {
    int cat1=3;
    int cat2=4;
    double ipotenuza=obiect.teoremaLuiPitagora(cat1,cat2);
    assertEquals( expected: 5, ipotenuza, delta: 0.01);
}
```

TEST1 = Cross-Check (este testata o metoda alternativa pentru a verifica corectitudinea metodei `teoremaLuiPitagora(cat1,cat2)`)

TEST2 = Cross-Check (este testata o metoda alternativa pentru a verifica corectitudinea metodei `teoremaLuiPitagora(cat1,cat2)`)

TEST3 = RIGHT (se verifica corectitudinea metodei prin verificarea rezultatului)

Test RIGHT pentru 9 si 40:

```
public double teoremaLuiPitagora(int cateta1, int cateta2) {
    return Math.sqrt(cateta1 * cateta1 + cateta2 * cateta2);
}
```

Valoarea pe care ne asteptam sa o primim: **$41 = \sqrt{9*9 + 40*40}$**

Ce principiu SOLID este incalcat in urmatoarea secventa de cod sursa?

```

public class NumarNatural {
    private int valoare;

    public void setValoare(int valoare) {
        if(valoare > 0) {
            this.valoare = valoare;
        }
    }

    public int getValoare() {
        return valoare;
    }
}

class NumarIntreg extends NumarNatural {
    private boolean semn;
    //...
}

```

Este incalcat **DIP** (clasa concreta ar trebui sa depinda de una abstracta, nu de alta concreta)

Ce principii SOLID sunt inculcate de metoda de mai jos?

```

public long calcul(int flag, int value1, int value2) {
    if(flag == 1) {
        return value1 + value2;
    } else if(flag == 2) {
        return value1 * value2;
    } else {
        return value1 - value2;
    }
}

```

SRP, OCP

Ce va afisa urmatorul test case la rulare?

```

public class TestCase {

    @BeforeClass
    public static void setUpClass() {
        System.out.print("A");
    }

    @Before
    public void setUp() {
        System.out.print("B");
    }

    @After
    public void tearDown() {
        System.out.print("C");
    }

    @Test
    public void test1() {
    }

    @Test
    public void test2() {
    }
}

```

ABCBC

Urmatoarea implementare de Singleton este gresita deoarece:

```

public class Singleton {
    private int atribut;
    private static Singleton instanta=null;

    private Singleton(int parametru) { this.atribut=parametru; }

    public static Singleton getInstanta(int parametru){
        if(instanta==null){
            return new Singleton(parametru);
        }else{
            return instanta;
        }
    }
}

```

instanta nu este initializata niciodata prin apelul constructorului

```

public class TestCase {

    @Test
    public void test1() {
        fail();
    }

    @Test
    public void test2() {
    }

    @Test
    public void test3() {
        Object o1 = new Object();
        Object o2 = o1;
        assertEquals(o1, o2);
    }
}

```

test1 -> pica (AssertionError/AssertionFailedError)

test2 si test3 -> trec si nu afiseaza nimic

Ce DP este implementat corect si complet in clasa de mai jos?

```

public class DesignPattern implements Cloneable {
    private static DesignPattern field;

    public static DesignPattern getField() {
        if(field == null) {
            field = new DesignPattern();
        }
        return field;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        return field;
    }
}

```

PROTOTYPE

```

int oMetoda(int valoare) throws Exception {
    int rez = 0;
    switch (valoare) {
        case 1:
            rez = 10;
            break;
        case 5:
            rez = 20;
            break;
        case 10:
            rez = 30;
            break;
        default:
            throw new Exception("Valoare gresita");
    }
    return rez;
}

```

TEST RANGE – 1, 5, 10 cred

TEST BOUNDARY – 1, 10

TEST RIGHT – 1, 5, 10 cred

TEST INVERSE RELATIONSHIP – idk

TEST ERROR CONDITIONS – 1

TEST EXISTENCE – testam daca parametrul nu exista/e nul/e 0 – 0

CODE COVERAGE 100% - prin minim 4/1 = 4 teste

COMPLEXITATE – sper sa nu dea

O companie doreste realizarea unui editor de texte online asemanator Google Documents/Microsoft Word online. Ce Design Patterns (unul sau mai multe) sugerati sa fie implementate pentru a da utilizatorilor posibilitatea sa revina la versiunile anterioare ale documentelor si la fiecare modificare locala sa se transmita automat si asincron un mesaj catre server pentru a face salvarea documentului?

MEMENTO – posibilitatea de a reveni la versiuni anterioare

OBSERVER – la fiecare modificare, serverul este notificat


```

public class NumarNatural {
    private int valoare;

    public void setValoare(int valoare) {
        if(valoare > 0) {
            this.valoare = valoare;
        }
    }

    public int getValoare() {
        return valoare;
    }
}

class NumarIntreg extends NumarNatural {
    private boolean semn;
    //...
}

```

Ce principiu SOLID e incalcat?

SRP – Single Responsibility

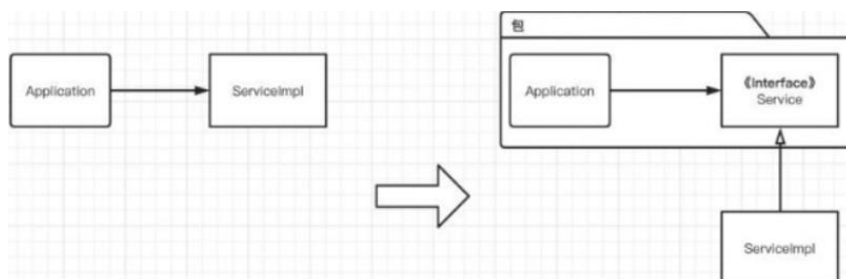
```

public int cmmdc(int param1, int param2){
    while(param1!=param2){
        if(param1 > param2)
            param1 -= param2;
        else
            param2 -= param1;
    }
    return param1;
}

```

Trebuie realizat un test de RIGHT. Metoda e apelata pentru 120 si 90. Rezultatul este comparat cu: **120**

Ce principiu de clean code a fost aplicat in imaginea urmatoare:



DIP – Dependency Inversion (este introdusa o interfata de care sa depinda modulele)

Pentru functia urmatoare, selectati combinatia de valori ce pot fi utilizate pentru a testa metoda si pentru a asigura un code coverage de 100%

```
public int oMetodaSimpla(int valoare) throws Exception
{
    int rezultat = 0;
    switch(valoare) {
        case 10:
            rezultat = 10;
            break;
        case 50:
            rezultat = 20;
            break;
        case 100:
            rezultat = 30;
            break;
        default:
            throw new Exception("Valoare gresita");
    }

    return rezultat;
}
```

10, 50, 100 (pentru a testa case-urile) **si 101** (sau orice alta valoare pentru a testa default)

Editati o aplicatie Windows Forms. Realizati faptul ca modificarea unui control definit de utilizator duce la modificarea formularului in care este definit controlul, fapt ce duce la modificarea formularului principal ce porneste formularul precedent. Ce principiu SOLID este incalcat?

OCP – Open Closed (deoarece se modifica codul pentru formularul principal)

O metoda ce realizeaza un back-up al bazei de date trebuie executata in fiecare noapte la ora 02:00. Doriti sa scrieti un test unitar ce verifica daca metoda chiar este apelata la acea ora. Acest test este de tip:

TIME (cred)

Ce reprezinta dezvoltarea de tip test-driven (TDD)?

Testele sunt scrise in mod normal inainte de implementarea codului

Ce design pattern defineste o familie de algoritmi, incapsuleaza fiecare algoritm si permite interschimbarea acestora?

ABSTRACT FACTORY

Realizati o aplicatie ce permite completarea de chestionare online. Indiferent de tipul chestionarului (Google Forms, Microsoft Forms, etc) este nevoie ca datele sa fie colectate intr-o anumita ordine: mai intai se colecteaza datele personale, mai apoi datele legate de chestionarul efectiv si la final datele legate de platforma utilizata. Ce Design Pattern puteti utiliza pentru a va asigura ca indiferent de tipul chestionarului regulile de mai sus sunt respectate?

TEMPLATE METHOD

Ce principiu SOLID a fost urmarit in implementarea urmatoare?

```
1
2 package ro.ase.acs.mesaje;
3 package ro.ase.acs.mesaje.service.LaRevedere
4
5 public class ControlMesaje {
6
7     private optiuneLaRevedere = new LaRevedere();
8
9     public String spuneLaRevedere() {
10         return "Mesaj: " + this.optiuneLaRevedere.getMessage();
11     }
12 }
```

SRP - Single Responsibility

Testati o metoda ce depinde de o conexiunea la o baza de date specificata prin interfata IConnecton. Cum procedati?

realizati un **FAKE** al interfetei IConnection

utilizati conexiunea aferenta aplicatiei

Care dintre următoarele variabile respecta convențiile de nume specific Java?

variabilaLocala, ceasFaraBaterie, mailPrimit, formulaMatematica

intrebareRetorica, dispozitivMobil, peisajCuMunti, studentIntegralist

Pentru urmatoarea implementare de tip Builder, indicate una sau mai multe **greseli** de implementare:

```
1 public class Tesla {
2
3     private int maxSpeedLimit;
4     private float price;
5     private String color;
6     private int batteryLevel;
7
8     public Tesla() { }
9
10    private Tesla(int maxSpeedLimit, float price, String color, int batteryLevel) {
11        super();
12        this.maxSpeedLimit = maxSpeedLimit;
13        this.price = price;
14        this.color = color;
15        this.batteryLevel = batteryLevel;
16    }
17
18    public static class TeslaConfigurator {
19        private Tesla tesla = new Tesla();
20        public TeslaConfigurator(float price) {
21            tesla.price = price;
22        }
23
24        public TeslaConfigurator setColor(String color) {
25            tesla.color = color;
26            return this;
27        }
28
29        public TeslaConfigurator setMaxSpeedLimit(int maxSpeedLimit) {
30            tesla.maxSpeedLimit = maxSpeedLimit;
31            return this;
32        }
33
34        public Tesla build() {
35            return new Tesla();
36        }
37    }
38 }
```

metoda **build()** din clasa builder, TeslaConfigurator, este implementata gresit
constructorul default din clasa Tesla este public

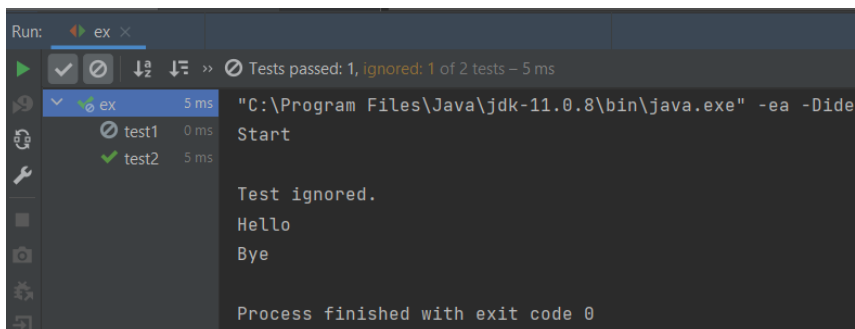
Ce principiu SOLID incalca fragmentul de cod din imagine?

```
public interface IStudent{
    void getVarsta();
    void eat();
    void move();
    void rentASstudio();
    void sustineExamen();
    void run();
    void purchaseHouse();
    void packForTrip();
}
```

SRP – Single Responsibility

Ce va fi afisat la consola?

```
1*import static org.junit.Assert.*;
9
10 public class TestStudent {
11
12     @BeforeClass
13     public static void setUpBeforeClass() throws Exception {
14         System.out.println("Start");
15     }
16
17     public static void tearDownAfterClass() throws Exception {
18         System.out.println("End");
19     }
20
21     @Before
22     public void setUp() throws Exception {
23         System.out.println("Hello");
24     }
25
26     @After
27     public void tearDown() throws Exception {
28         System.out.println("Bye");
29     }
30
31     @Ignore
32     @Test
33     public void test1() {
34         fail("Not yet implemented");
35     }
36
37     @Test
38     public void test2() {
39         assertTrue(true);
40     }
41 }
42
43
```



@BeforeClass setUpBeforeClass -> se apeleaza primul

tearDownAfterClass -> nu afiseaza nimic

@Before setUp -> se apeleaza dupa @BeforeClass

@After tearDown -> se apeleaza dupa @Before

@Ignore test1 -> Test ignored

test2 -> trece

RASPUNS:

Start

Hello

Bye

Ordinea efectuării testelor:

```
public class ex {
    @BeforeClass
    public static void test1() throws Exception {
        System.out.println("TEST1");
    }
    @AfterClass
    public static void test2() throws Exception {
        System.out.println("TEST2");
    }
    @Before
    public void test3() throws Exception {
        System.out.println("TEST3");
    }
    @After
    public void test4() throws Exception {
        System.out.println("TEST4");
    }
    @Test
    public void test5() {
        System.out.println("TEST5");
    }
    @Test
    public void test6() {
        assertTrue( condition: true);
        System.out.println("TEST6");
    }
    @Test
    public void test7() {
        assertTrue( condition: false);
        System.out.println("TEST7");
    }
    @Test
    public void test8(){
        System.out.println("TEST8");
    }
}
```

Run: ex

Tests failed: 1, passed: 3 of 4 tests - 29 ms

Test	Duration	Status
test5	10 ms	Passed
test6	0 ms	Passed
test7	19 ms	Failed
test8	0 ms	Passed

TEST1
TEST3
TEST5
TEST4
TEST3
TEST6
TEST4
TEST3
TEST4

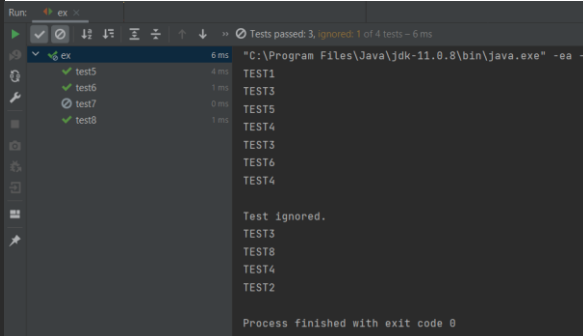
java.lang.AssertionError <3 internal lines>
at ex.test7(ex.java:31) <31 internal lines>

TEST3
TEST8
TEST4
TEST2

Process finished with exit code -1

Ordinea efectuării testelor:

```
public class ex {
    @BeforeClass
    public static void test1() throws Exception {
        System.out.println("TEST1");
    }
    @AfterClass
    public static void test2() throws Exception {
        System.out.println("TEST2");
    }
    @Before
    public void test3() throws Exception {
        System.out.println("TEST3");
    }
    @After
    public void test4() throws Exception {
        System.out.println("TEST4");
    }
    @Test
    public void test5() {
        System.out.println("TEST5");
    }
    @Test
    public void test6() {
        System.out.println("TEST6");
    }
    @Ignore
    @Test
    public void test7() {
        System.out.println("TEST7");
    }
    @Test
    public void test8(){
        System.out.println("TEST8");
    }
}
```



@BeforeClass

@Before

@Test test5

@After

@Before

@Test test6

@After

Pattern-ul FAÇADE promovează decuplarea (“weak coupling”) între subsistem și clienți.

ADEVARAT

Dorești să reduci costurile de dezvoltare prin reutilizarea metodelor implementate în componente din alte proiecte. Ce design pattern îți va permite să faci asta fără să le modifice?

ADAPTER

Doriti realizarea unei solutii software de gestionare a unor componente electronice smart ce isi schimba culoarea. Utilizatorii pot crea componente noi prin combinarea de componente existente, iar acestea isi vor schimba culoarea toate odata. Ce Design Patterns (unul sau mai multe) propuneti pentru a da utilizatorilor posibilitatea de a schimba culoarea tuturor componentelor detinute in acelasi timp avand in vedere urmatoarele:

1. se pot realiza modele utilizand orice combinatii de componente, iar acestea sunt interschimbabile;
2. interfata unei componente este complexa si doriti sa expuneti doar functia de schimbare a culorii catre utilizatorii finali.

DECORATOR – se adauga functionalitati noi obiectelor existente

FAÇADE – interfata complexa; se doreste sa se expuna doar functia de schimbare a culorii

O colectie de teste unitare se numeste:

Suite

Test Case

In clasa Student este implementata metoda setVarsta (int varsta). Stiind ca varsta acceptata este cuprinsa in intervalul (14, 90] indicat valorile ce pot fi folosite pentru teste de tip Range (una sau mai multe valori)

1000

0

13

15

14

Cele cinci principii SOLID conduc la scrierea codului intr-un mod:

usor de modificat

Actualizati (refactorizati) o aplicatie web. Realizati faptul ca de fiecare data cand un nou tip de utilizator este necesar trebuie sa mai adaugati un caz intr-o instructiune de selectie multipla (switch). Ce principiu SOLID este incalcat?

OCP

Pentru sistemul software al unei universități trebuie adăugată o nouă funcționalitate de susținere a unui examen de către student a finalul unui stagiu în mod unic. Programatorii decid să realizeze modificări în codul original. Modificările sunt semnificative pentru a accepta această caracteristică. Ce principiu SOLID încalcă acest lucru?

OCP

Care dintre următoarele DP încalcă Dependency Inversion Principle?

DIP = modulele depind de clasa abstractă/interfață, nu de clase concrete

Factory Method – dependența de interfața Factory

State – dependența de interfața State

Observer = o aplicare a DIP – dependența de interfețele Observer și Subject

Strategy = satisface DIP – dependența de interfață

Command – dependența de interfața Command

Simple Factory –

Decorator – dependența de DecoratorAbstract

Interfețele specializate pe client sunt mai bune decât o interfață generalizată:

ISP Interface Segregation

O instituție bancară realizează plăți cu cardul prin intermediul unei interfețe Viza. Banca dorește să renunțe la acest furnizor de carduri și să folosească un altul ce expune o interfață de tip ManagerCard, fără a modifica foarte mult soluția existentă. În plus, deoarece crearea unui card bancar nou durează minim 1 săptămână, banca dorește o modelare prin care să i furnizeze, titularului o clonă temporară digitală a cardului până la furnizarea celui nou. Ce design Patterns (unul sau mai multe) ați putea utiliza pentru a rezolva cele 2 probleme?

ADAPTER – interfața care adaptează la codul vechi

PROTOTYPE – clonă temporară a cardului

```

interface IDP {
    default void execute() {
        String message = receive();
        send(message);
    }
    void send(String message);
    String receive();
}

class Class implements IDP {
    @Override
    public void send(String message) {
        System.out.println(message);
    }
    @Override
    public String receive() {
        String result = "";
        //...
        return result;
    }
}

public class DP implements IDP {
    private IDP idp;
    //...
    @Override
    public void send(String message) {
        System.out.println("Message: ");
        idp.send(message);
    }
    @Override
    public String receive() {
        return idp.receive();
    }
}

```

COMMAND – metoda execute()

OBSERVER ??

La rularea suitei din imagine vor fi rulate:

```

@RunWith(Suite.class)
@SuiteClasses({ TestCase1.class, TestCase2.class })
@IncludeCategory({Categorie1.class})
@ExcludeCategory({Categorie2.class})
public class SuitaCustom {
}

```

Toate testele din TestCase1 si TestCase2

```

@Test(expected = InvalidNameException.class)
public void testSetNameForNull() throws InvalidNameException {
    Student s = new Student();
    s.setNume(null);
}

```

Test Error Condition si Existence

Metoda assertEquals verifica daca:

cele doua valori sunt salvate la aceeași adresă de memorie

Ce fel de test e?

```
public double calculCredit(double salariu, int varsta) throws Exception {
    if(salariu<2100 || varsta<18)
        throw new Exception();
    return salariu*0.4;
}

@Test
public void test() throws Exception {
    assertEquals(1200,obiect.calculCredit(3000, 18),0.01);
}
```

RIGHT cred

In cadrul unei suite de teste pot fi specificate spre a fi executate:

toate testele cu exceptia unor categorii

doar anumite categorii de teste

test cases

alte suite

Refference din CORRECT se refera la

posibilitatea de a folosi referinte catre fisiere

Daca se considera functia urmatoare, indicate cu ce valori se poate implementa un test unitar tip Cardinalitate=0?

```
public float prelucrareValori(ArrayList<Integer> valori){
    float rezultat = 0;

    for(int valoare : valori) {
        if(valoare %2 == 0)
            rezultat += valoare;
    }

    return rezultat;
}
```

```
ArrayList<Integer> input = new ArrayList<>();
```

Ce test se realizeaza?

```
public int getMaxNota(int[] note) {
    int max=note[0];
    for(int nota:note) {
        if(max<nota)
            max=nota;
    }
    return max;
}

public boolean verifNota(int nota, int[] note) {
    for(int i:note) {
        if(nota==i)
            return true;
    }
    return false;
}

@Test
public void test2() {
    int notaMaxima= obiect.getMaxNota(note);
    assertTrue(obiect.verifNota(notaMaxima, note));
}
```

RIGHT cred

```
@Test
public void test3() {
    int a=4;
    int b=6;
    int max1=obiect.getMax(a, b);
    int max2=obiect.getMax(b, a);
    assertEquals(max1,max2);
}
```

RIGHT
