



I. **clean code**

principii

(DRY) (KISS) (YAGNI) (SOLID)

- | — codul trebuie să fie:
- | → ușor de citit
- | → ușor de înțeles
- | → ușor de modificat, de către
oricine

1) **DRY** (Don't Repeat Yourself)

- CTRL C + CTRL V bucati de cod
- scriem met./fct. cod care fac acelasi lucru, fara sa ne daram seama

2) **KISS** (Keep It Simple & Stupid)

- cand complicam prea mult si met. din dorinta de a face cat mai multe lucruri

AVANTAJE

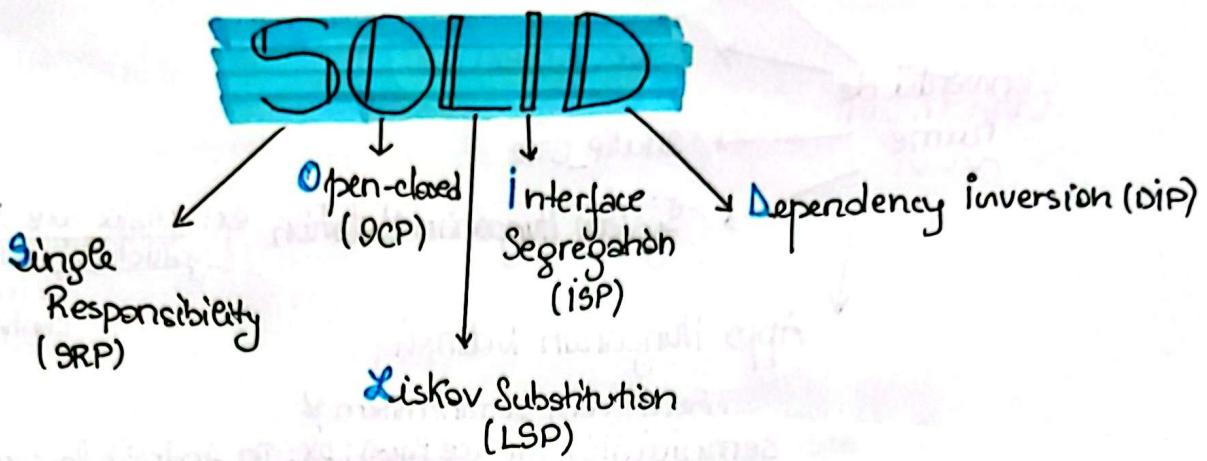
- rezolvam rapid probleme complexe, intr-un mod cat mai simplu
- intretinem mai usor codul, putand fi usor de extins si de modificat

DEZAVANTAJE

- daca simplificam prea mult sau nu ne documentam codul, poate fi greu de intelese pt. ceilalati (sau pt. noi mai tarziu)

3) **YAGNI** (You Aint Gonna Need It) ← *derivat din Kiss

- cand scriem fct./met. pe care de fapt nici nu le folosim



a) SINGLE RESPONSIBILITY → Façade

- o clasă trebuie să aibă întotdeauna o singură responsabilitate și nu mai una, altfel riscul să rescriem întregul cod dacă modificăm ceva.

b) OPEN-CLOSED

- clasele trebuie să fie deschise pt. extindere, dar închise pt. modificări
 codul existent nu trebuie modificat

adăugăm comportamente noi fără a modifica codul existent

c) LISKOV SUBSTITUTION ↔ „DESIGN BY CONTRACT”

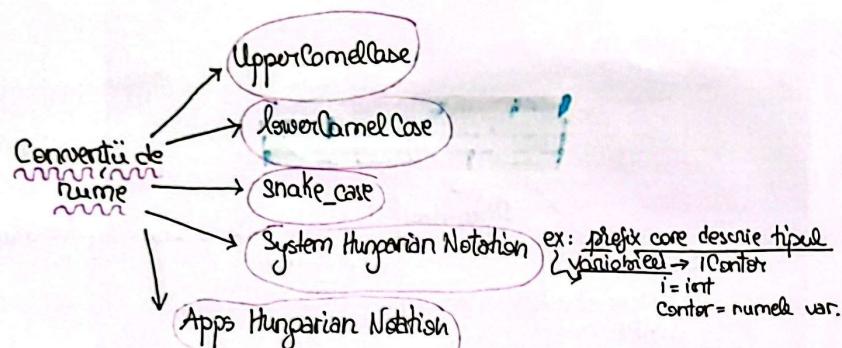
- putem înlocui obiectele oricărui cu instanțe ale claselor derivate fără ca acest lucru să afleze funcționalitatea
 ⇒ dacă B moștenește clasa A, atunci clasa B trebuie să poate fi folosită oriunde se așteaptă un ob. A, fără a cauza probleme

d) INTERFACE SEGREGATION

- mai degrabă folosim mai multe interfețe specificabile decât să avem una singură care face toate lucurile într-un lloc

e) DEPENDENCY INVERSION

- codul trebuie să depindă de interfețe/cl. abstracte



când vrem să transmitem și semnificația var. (ce face); ex: în Android, la componentele UI
btnSend

Reguli de scriere a codului surse.

- fiecare var. se declară pe căte și linie
- blocurile de cod - încep cu { și se termină cu }, chiar dacă avem o singură instrucțiune - le marcam prin indentare
- punem spatiu între headerul unei funcții și {
- același } se pune pe o linie, **cu EXCEPȚIA** cazurilor când avem if-else / try-catch
- separăm met. unele de actele printindu-să singură linie, param. pun virgula și spatiu, **împreună** de operatorii print-un singur spatiu
!CU EXCEPȚIA operatorilor unari
- folosim operatorul ternar ori de câte ori e posibil
- nu comparăm direct cu string-uri → folosim enum
- met. trebuie (e indicat) să aibă maxim 3 niveluri de structuri imbătățite
- ieșirea din funcție trebuie să fie cât mai rapidă
- declarăm var. cât mai aproape de folosirea lor
- de evitat met. cu peste 2 param. și cele cu peste 20 de linii de cod
- complexitate invers proporțională cu nr. liniilor de cod

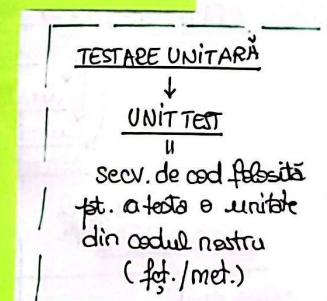
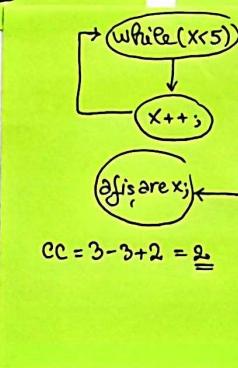
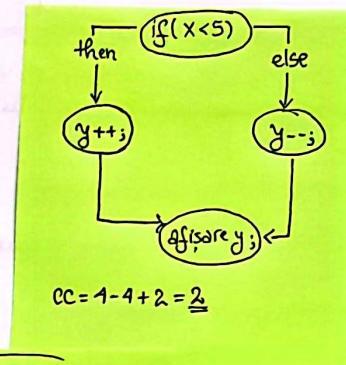
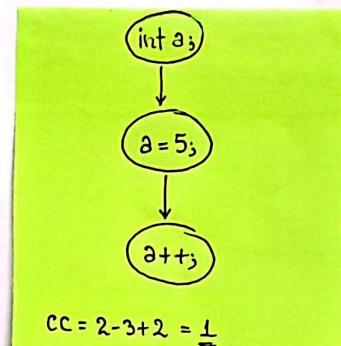
COMPLEXITATEA CICLOMATICĂ → determină numărul de teste!

• 6 metode simple ⇒ $CC = 1$.

• crește edată cu nr. de str. if-else și switch.

$$CC = M - N + 2$$

nr. muchii
nr. noduri



Reguli de Clean Code pt. metode:

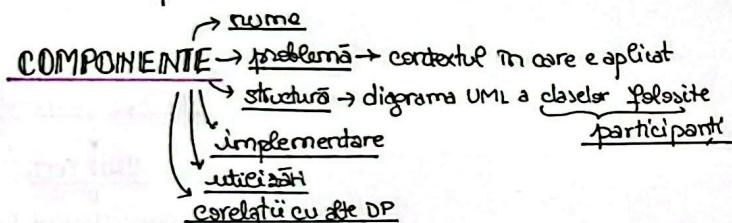
- S.P.P
- K.I.S.S
- ↓ delegă plus poartă/interfețe
- ↓ folosește interfețe

DESIGN PATTERNS

design patterns

Salvoane de protecție care ne ajută să rezolvăm diferite tipuri de probleme

- Sunt deja folosite și testate de comunitate, ceea ce poate ușura comunicarea între programatori → vocabular comun
 - Înțelegerea mai facilă a codului sursă
 - refactoring
 - documentarea codului sursă
- ⇒ recunoscerea unui DP poate produce neclarități, înțelesul codului sursă se complica consumând timp de analiză



CREAȚIONALE	STRUCTUALE	COMPORTAMENTALE
• SINGLETON	• ADAPTER	• OBSERVER
• BUILDER	• COMPOSITE	• CHAIN OF RESPONSIBILITY
• FACTORY	• FAÇADE	• STRATEGY
• FACTORY METHOD	• DECORATOR	• TEMPLATE METHOD
• ABSTRACT FACTORY	• FLYWEIGHT	• COMMAND
• PROTOTYPE	• PROXY	• MEMENTO
	* Bridge	• STATE
		* Visitor, Interpreter, Mediator, Visitor

-5-

DESIGN PATTERNS

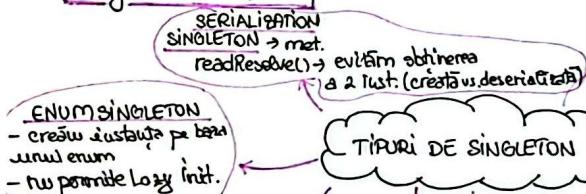
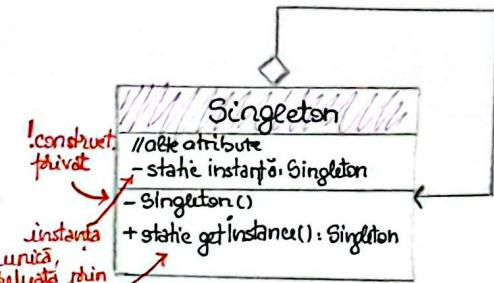
creationale

⇒ inițializarea și configurarea claselor și obiectelor, separând crearea obiectelor de utilizarea lor

(1) SINGLETON

- Este un singur obiect de tipul clasei respective
- clasa însăși se asigură că poate fi creat un singur obiect, fără a permite utilizăriile

= instanță unică / o singură instanță



EAGER INITIALIZATION
- inițializarea instanței de la unul și felicitor ⇒ la declarare
- nu e eficientă

THREAD SAFE INITIALIZATION

- evită apelul mut.
- pe alt fin de execuție
- ⇒ synchronized

INNER STATIC

- HELPER CLASS
- avem o clasa membrată care se ocupă de crearea instanței

SINGLETON REGISTRY → gestiunea obiectelor unice într-o colecție (collection)

Utilizări

- deschiderea unei singure instanțe a unei aplicații
- conexiune unică la BD
- Shared Preferences și DocumentBuilderFactory în Android
- Factory → o singură fabrică de obiecte
- Builder → un singur obiect plus core construim alte obiecte

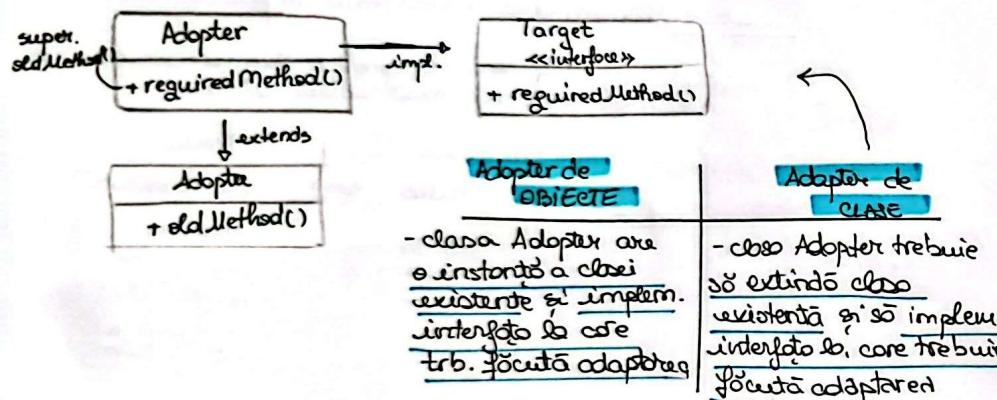
-6-

• DESIGN PATTERNS

structurale

- compunerea + configurarea claselor și obiectelor
- decuplare interioare de clase

1) ADAPTER → utilizat atunci când anumite clase din framework-ului diferă nu au o interfață comună, fiind deci incompatibile
→ NU modifică clasele existente, ci adăugă noi clase pt. a realiza un Adapter între ele
↓
! NU adăugă funcționalități



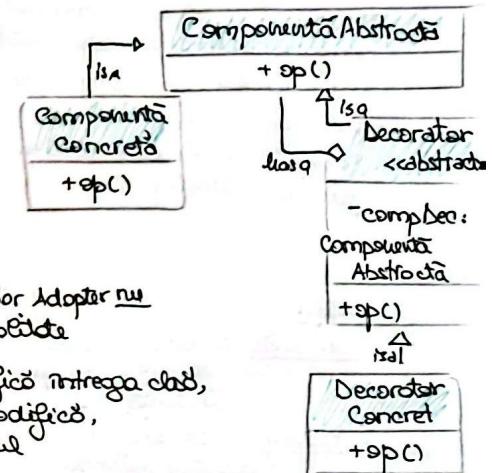
Corelații:

- Facade → ambele sunt wrappere
- Decorator → face același lucru, dar decorator adăugă și funcționalități noi
- Proxy → ambele ascund clasele existente
- * • Bridge → au strucțură asemănătoare

Corelații cu Singleton, dezărește Facade poate fi unică

2) FACADE → rezolvă lucruul cu framework-ului complex, aducând teste metodele într-un singur loc
→ abstr. Facade are metode care utilizează felicită de la framework-ului, pt. a ascunde complexitatea

3) DECORATOR → modificați funcționalitatea unei clase și adăugarea de noi funcționalități
runtime
• modificare continuă
||
decorări multiple
fără a modifica codul existent și fără a face modificare

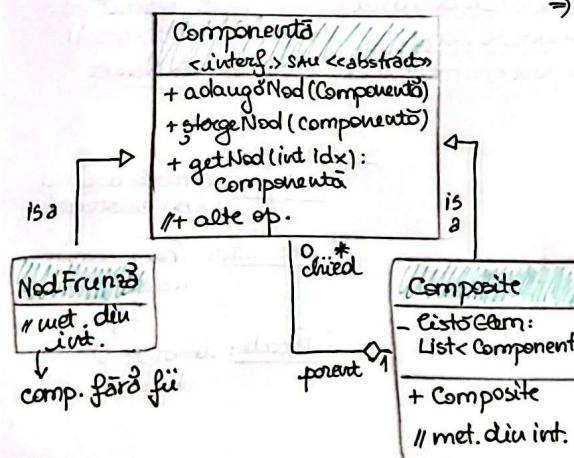


Corelații:

- Adapter → face același lucru, dar Adapter nu adăugă funcționalitate
- Strategy → Decoratorul modifică întregul obiect, iar Strategy modifică comportamentul

4) COMPOSITE → creare structuri ierarhice și arborescente

- meniuuri, organizație, depozite etc.

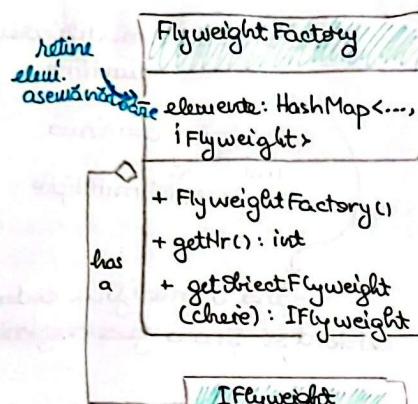


Corelații:

- Decorator → nodurile composite pot fi privite ca noduri decorative
- Chain of Responsibility → legăturile dintre clase

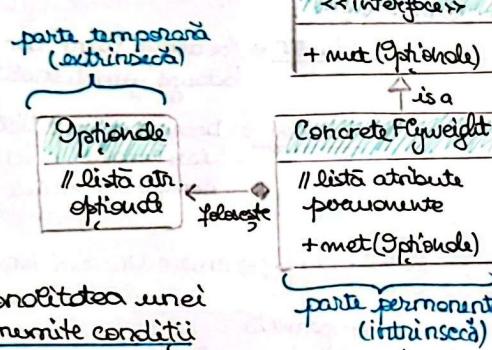
5) FLYWEIGHT

- avem de construit foarte multe instante ale unei clase care au o parte comună / permanentă
- urmărим să reducем consumul de memorie → păstrăm parte comună într-o singură instanță, iar partea care diferă de la un obiect la altul este salvată într-o altă clasă și folosită ulterior



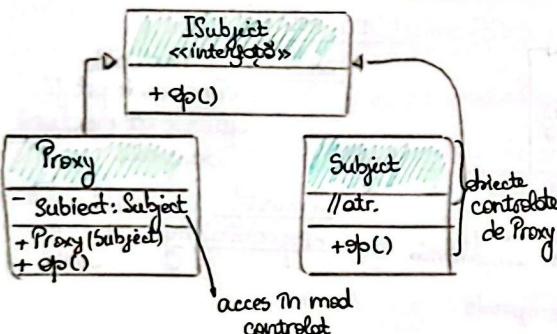
Corelatii:

- Composite: reprezintă grunță și poate reprez. același obiect în mai multe instanțe
- Factory: construirea de obiecte și gestionarea de obiecte



6) PROXY → păstrăm funcționalitatea unei clase, doar în anumite condiții

→ controlăm comportamentul și accesul la un obiect



Corelatii:

- Adaptor: ambele ascund clasa existentă
- Decorator: Proxy permite accesul la unele func.
- Fascade: interfață pt. obiectul real

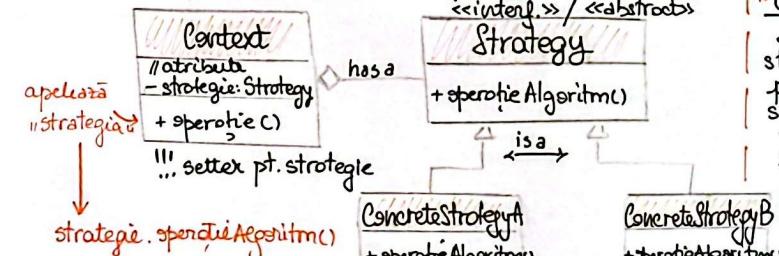
DESIGN PATTERNS comportamentale

- operă soluții care să susțină interacțiunea între obiecte și clase
- controlează relațiile complexe dintre clase
- distribuția responsabilităților pe clase

1) STRATEGY

→ avem mai mulți algoritmi pt. rezolvare. unei probleme, iară implementările se face la runtime!

→ fiecare comportament = CLASA



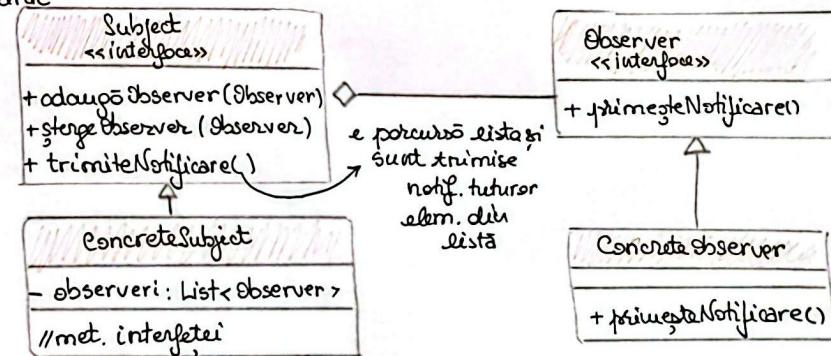
Corelatii:

- State: la strategy, strategia e dată ca parametru, în schimb la state trecerea de la o stare la alta se face controlat

utilizat în MVC

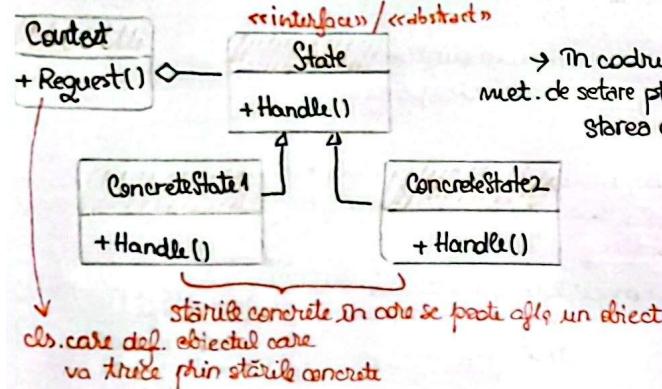
2) OBSERVER

Corelatii cu Singleton, deoarece subiectul poate fi unic



3) STATE → folosit atunci când un obiect nu schimbă comportamentul în funcție de starea în care se află

→ se desvăluie de Strategy prin modul de schimbare a strategiei



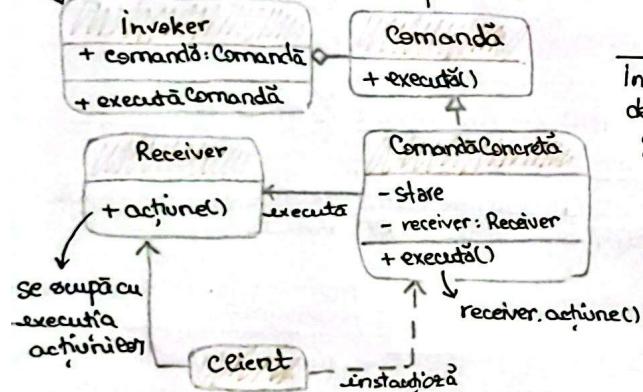
→ în cadrul fiecărei clase de stare avem metode de setare pt. stare și în care modificăm starea contextului

Utilizări:

- stările rezervărilor / comenzi
- pt. evitarea utilizării switch sau if-else

4) COMMAND → implementarea de lose coupling

găsirea comenzii
 → ascunde aplicarea de comenzi fără să găsească concret ce preaștează aceea comandă
 clientul e decuplat de cel care execută acțiunea



utilizări → macro-urile
 → lucru cu fiziere
 → eliberează se dorește revenirea la o stare anterioră fără intermediul comenzielor

Corelații:

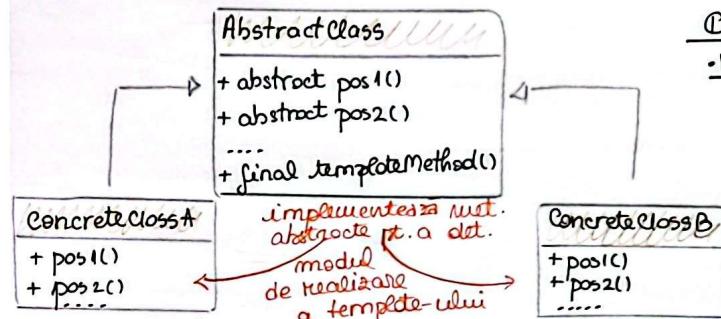
- Memento → revenire la stările anterioare
- Adapter → se folosește funcții deja existente

5) TEMPLATE METHOD

→ avem un algoritm cunoscut, cu un nr. precis și finit de pași

→ fiecare pas = o metodă → evitare suprascriere

→ avem o met. finală!, care implem. algoritmul și apelează toate met. din componentă sa



Corelații:
 • Factory → furniză de obiecte

6) CHAIN of RESPONSIBILITY

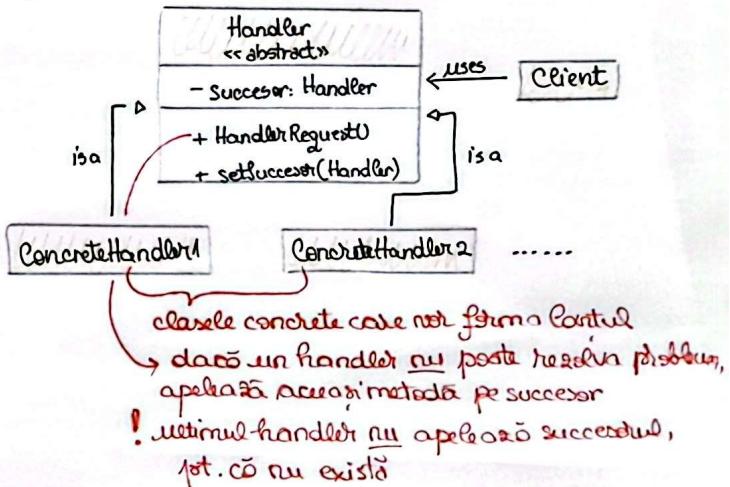
DNS resolver

→ folosit când nu stim exact cine poate rezolva o problemă, dar avem o listă cu obiectivii care să poată opera

→ obiectele sunt ordonate într-un lanț, apoi cel care este de rezolvat problema apelaază prima sa din partea

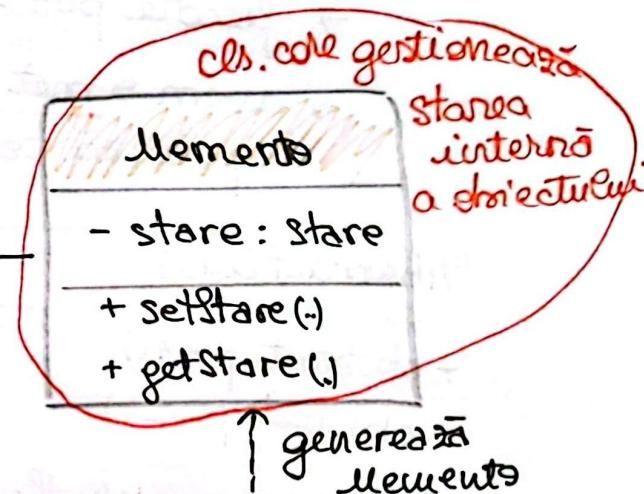
Corelații:

- Composite → structură asemănătoare

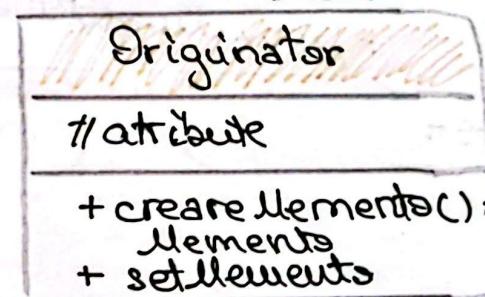


7) MEMENTO → salvarea stării unui obiect și revenirea la acestea din de către ori se dorește

⇒ backup



↑ generează Memento



gestionează directele de tip
memento

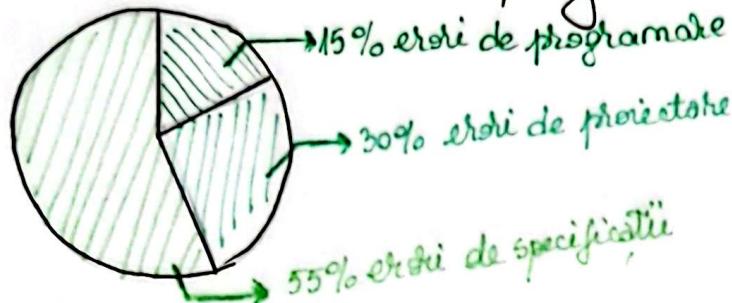
Utilizări:
→ salvarea fișierelor
→ realizarea de backup-uri

Correlanții → Command → revenirea la stările anterioare

INIT TESTING

III. Unit testing

→ căutăm erori și defecte în cod pt. a le putea remedia



TEST DRIVEN DEVELOPMENT

↓ dezvoltare pe baza testelor
↔ scriem teste mai multe de
a serie codură efectiv

	DESCRİERE	AUANTAJE	DEZAVANTAJE
BLACK Box TESTING ~ testare comportamentală	<ul style="list-style-type: none"> metodă de a testa aplicația software de către persoane care nu cunosc arhitectura internă a aplicației testerul stie doar datele de intrare și de ieșire ale aplicației 	<ul style="list-style-type: none"> teste sunt realizate după al userului testerul nu trebuie să stie programare teste sunt efectuate independent de dezvoltator și au o perspectivă obiectivă 	<ul style="list-style-type: none"> căzurile de utilizare sunt greu de proiectat teste vor avea un nr. mic de intrări teste pot fi redundante cu alte teste realizate de dezvoltator
WHITE Box TESTING ↗ Clear Box testing ↗ Open Box -" ↗ Glass Box -" ↗ Transparent Box testing ↗ Code-Based testing ↗ Structural testing	<ul style="list-style-type: none"> flexibilitate de dezvoltatori/testerii care stiu structura internă a aplicației testate 	<ul style="list-style-type: none"> se poate realiza înainte ca aplicația să fie pusă în funcțiune este mai apărată, putând acoperi mai multe posibilități 	<ul style="list-style-type: none"> cunoștințe de progr. profundate multe resurse înțeleptarea codului de testare poate fi delicatesă dacă se fac modificări frecvente

- user de scris, scriind avem nevoie de ele
- putem defini colecții de teste
- pot fi ruleate de fiecare dată când e nevoie
- reduc timpul pt. debugging, nr. de bug-uri în cod
- framework-uri și instrumente care simplifică procesul de scriere și rulare a testelor

De ce să folosim teste unitare?

De ce să nu folosim teste unitare?

→ găsim scuze core ne determină să nu o facem, deși ar trebui.

CONCEPTE JUNIT

- › **Fixture** = set de obiecte utilizate în test
- › **Test Case** = ceeașă unde definim setul de obiecte pt. a rula mai multe teste
- › **Setup** = o metodă/etapă de definire a setului de obiecte utilizate, înainte de testare
- › **Teardown** = o metodă/etapă prin care "dezafectăm" spațiul ocupat de fixture după terminarea testelor
- › **Test Suite** = o colecție de mai multe teste
- › **Test Runner** = instrument de rulare a testelor și de afișare a rezultatelor

ASSERTIONS - JUNIT

valori egale → assertEqual (expected, actual)

lucrăm cu nr. real și avem în evit de o marejă de eroare

assertEqual (expected, actual, delta)

referințe la unu obiect → assertSame (expected, actual)

assertNotSame (expected, actual)

assertNull (object) / assertNull (object)

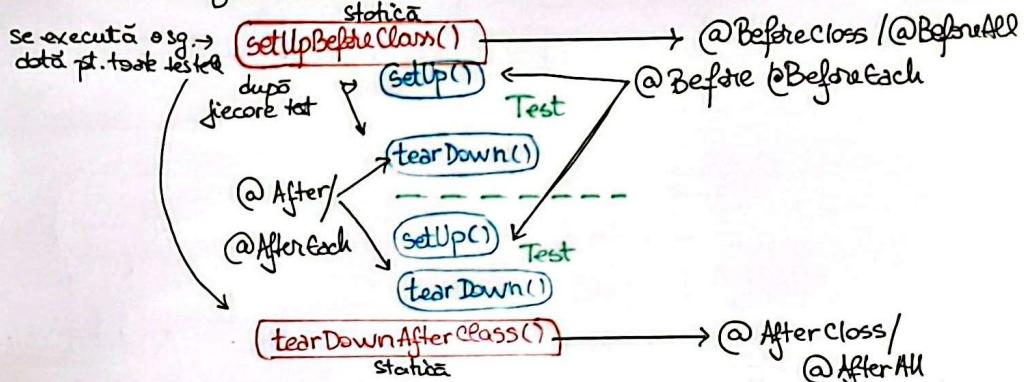
assertTrue (condition) / assertFalse (condition)

fail (message)

intenționat
y fățăm reacția unui test

la trăile puter
da ca prim
parametru și
un
mesaj
afisat în
cadrul
în cadrul
testul prică

Unit testing - JUnit Skeleton



* în JUnit3 nu există adnotăriri, deci sunt obligatorii numele metodelor

JUnit4 / JUnit5

JUnit3

- versiune JDK > 1.2
- clasele de test trebuie să moștenească clasa TestCase

- numele metodelor de test e construit după formația testABC

- dacă se dorește ca un test să nu fie rulat
→ stergem testul, și comentăm sau modificăm numele

JUnit4

- versiune JDK > 5
- nu e necesar

- metodele de test sunt anotate cu @Test

- @Test(timeout = 1000)
↳ milisecunde

→ @Ignore sau @SpringBootTest

Errors → simulăm și fățăm obținerea erorilor pt. a verifica comportamentul metodelor în caz de eroare

Right - BiCEP

→ verifică că rezultatele metodelor sunt corecte

→ Cross-Check → se verifică corectitudinea unei metode folosind metode de calcul similare

Boundary

- verifică teste limită și docă pt. că rezultatele sunt corecte

colectării de rezultate

Right → verificarea conformității cu specificațiile proiectului

Boundary → testăm marginile intervalelor (atât pt. limitele inferioare, cât și cele superioare)
*→ limitele extreme pot fi identificate cu principiul C.O.R.R.E.C.T.

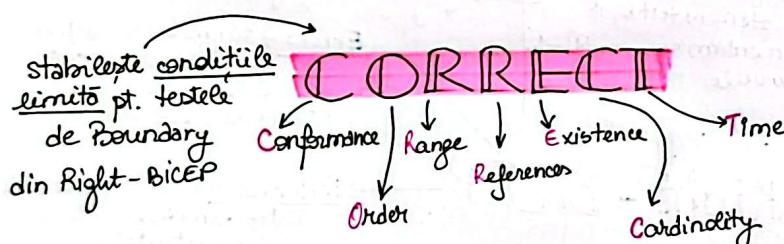
Inverse Relationship → pornind de la rezultat, trebuie să se ajungă la același input
→ aplicat de către pe met. matematiice

Cross-Check → testăm o metodă cu ajutorul altor metode
↳ metoda nouă are rolul de a crește productivitatea sau de a reduce consumul de resurse

Error Conditions → testăm situații în care aplicația ar putea crăpa

Performance → testăm cât de bine funcționează o metodă, verificând performanța → resurse consumate timp necesar
↳ când inputul/rezultatul unei met. e să existe/un nr. mare de elem.

@Test(timeout = 100)



Conformance = Type testing / Compliance testing / Conformity assessment

- ↳ fără neregări în care cauza trb. să îndeplinească anumite standarde
- conformitatea cu un format sau un standard
- ↳ aceste teste verifică dacă datele de intrare nu sunt conforme cu formatul / rezultatul obținut este conform cu formatul specific

Ordering → specifică de către listelor, putem verifica/testa comport. met. dacă primește anumiti param. în altă ordine

Range → trebuie verificată valoarea din interval

Reference → folosit când unele met. depind de ceva din exterior precondiții/condiții preliminare

→ sunt efectuate cu ajutorul dublurilor de test
↳ stub fake dummy mock

Existence → trebuie să verificăm ce se întâmplă cu o metodă dacă un param. nu există, e nul sau e 0

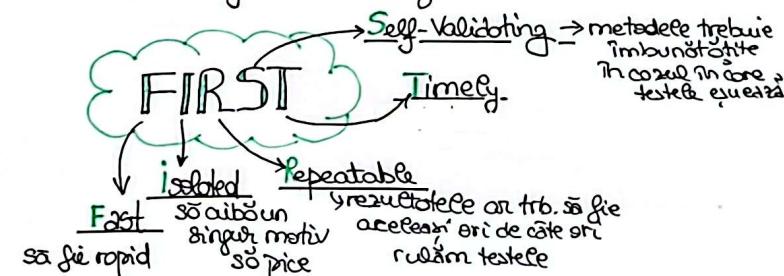
↳ similar cu Error din Right-BICEP

Cardinality ↔ 0-1-n Rule (similar cu teste de Existence și Range)

→ trebuie să verificăm dacă met./lista/colecția are 0, 1 sau n elem.

! test Boundary superior

Time → similar cu Performance din Right-BICEP



DUBLURI DE TESTARE (MOCKS)

↪ obiecte care se potrivesc cu interfața colaboratorului real, pt. a face testarea mai ușoară, mai rapidă și mai ieftină

- **DUMMY OBJECT** = un obiect care respectă interfața dorite metodele nu folosind niciun nou cod
- **STUB** = interacțiuni hardcodate, putând fi folosite cu apeluri false sau un fake
- **SPY** = un Stub care gestionază și contabilizează nr. apelurilor
- **FAKE** = obiect care se comportă similar cu unul real, dar cu o versiune mai simplificată de obicei putem stabili valori ale trb. să se întâarcă
- **MOCK**

↪ Mock testing → derivă din met. testată să nu fie influențată de referințe externe

Mockito EasyMock

@RunWith(....class)
@Suite.SuiteClasses({...})
@Categories.IncludeCategory
@Categories.ExcludeCategory
@Test @Category(....class)

→ cece runner sunt ruleză teste

→ definește care clase de test sunt incluse în sursă

→ rulează doar aceste teste

→ exclude aceste teste

→ stabilește din ce categorie fac parte teste

! Suite! TOATE TESTELE
Categories

TESTELE

TOATE

TESTELE