**Curs CTS**

-seminar 50% - minim 2

-examen 50% - minim 5

1. *Gestiunea versiunilor codului sursa – Git, SVN*

*Necesitatea controlului versiunilor:*

* Caracterul colaborativ al proiectelor
* Backup pt codul scris
* Istoricul modificarilor

*Terminologie:*

GIT = sistem distribuit de versionare

VSC = version control software

SCM = source control management

Repository = componenta server – continue ierarhia de fisiere si istoricul modificarilor

Checkout = preluare in mediul local a anumitor modificari realizate pe server (repository)

Working copy = versiunea locala a proiectului; versiunea in care lucreaza progrmatorul

Commit = cerere de publicare a unor modificari realizate in working copy in repository-ul local

Pull = actiune de actualizare a infomratiilor locale cu cele de pe server

Conflict = apare atunci cand mai multi utilizatori au realizat modificari in aceleasi fisiere din proiect; sistemul de aplicare a versiunilor diferite nu poate imbina modificarile si astfel este nevoie de interventia umana pt a realiza merge

Merge = procesul de unire a doua sau mai multe versiuni de lucru

Branch = ramura secundara de dezvoltare a proiectului, pe langa master

Revert = revenire la o versiune anterioara de pe un anumit fir de dezvoltare (branch)

Stash = arhiva locala pt un set de modificari

*Flux GIT*:

Initializare repository (git init) -> Descarca proiect (git clone) -> Commit modificari (git commit) -> <- Lucru in proiect

Git status -> verificare status proiect (raspunsuri posibile: nu exista nimic pt commit sau afiseaza fisierele modificate)

Git add -> adaugarea noului fisier in track

Git commit -m “mesaj” -> commit

Git push -> trimite fisierul pe server

Git clone -> ma mut in alt folder si clonez un repository de pe git

Git init, git remote add origin, git clone …

Git pull <<branch>> sau origin -> actualizarea la versiunea curenta

*Lucrul cu branch-uri:*

Git branch -> afiseaza branch-urile locale

Git branch -a -> afiseaza branch-urile locale si pe cele din repository

Git checkout <<branch\_local>> -> se schimba branch-ul pe care se lucreaza

Git branch -b <<new\_branch>> -> se creaza un nou branch pe baza celui curent

Git branch -b <<new\_branch>> <<branch\_sursa>> -> creaza un nou branch sincronizat cu branch-ul sursa din repository

Git branch -D <<branch>> -> sterge branch-ul local si de pe repository. Daca se foloseste -D, stergerea se face chiar daca exista modificari ne-merge-uite, iar, daca se foloseste -d si exista modificari, stergerea nu se face.

Git merge <<branch\_cu\_modificari>> -> merge-uirea branch-urilor

1. *Clean Code*

Cand citim cod, creierul nostrum joaca rolul de compilator; putem tinem minte pana la 7 (+-2) valori deodata.

Bad code = greu de citit, se strica daca modifici ceva, are multe dependinte externe (glass breaking code), strans legat (tight coupled) de alte secvente de cod

*Principii:*

*DRY (DON’T REPEAT YOURSELF)*

Se aplica ori de cate ori dam copy-paste la bucati de cod sau cand scriem doua metode care fac acelasi lucru.

*KISS (KEEP IT SIMPLE AND STUPID)*

Cand avem metode care fac mai mult de un lucru.

Beneficiile respectarii KISS:

* Permite rezolvarea rapida de probleme complexe intr-o maniera simpla
* Permite realizarea de produse complexe usor de intretinut
* Codul e mai flexibil si usor de extins si modificat daca apar cerinte noi

*YAGNI (YOU AIN’T GONNA NEED IT)*

Scrierea de metode care nu ne trebuiesc; partial derivat din KISS.

*SOLID*

**Single Responsibility**

* O clasa sau o metoda trebuie sa aiba o singura responsabilitate si un singur motiv de a fi modificata, altfel, orice schimbare de specificatii duce la inutilitatea ei
* Sa nu depinda de mai multi actori diferiti (cum este ex cu clasa Student care depeinde de departamentele de conta, academic, IT) => impartirea clasei in mai multe clase care depend de un singur actor, care contin metoda specifica, si crearea unei interfete in care se regasesc toate aceste metode

**Open-Closed**

* O clasa trebuie sa fie deschisa pt extensii, dar inchisa pt modificari

**Liskov Substitution**

* Obiectele pot fi inlocuite oricand cu instante ale claselor derivate fara ca acest lucru sa afecteze functionalitatea
* Numit si “design by contract”
* Nerespectarea acestui principiu duce la nevoia unor mecanisme suplimentare pt rezolvarea problemei si de utilizare adecvata a clasei (altfel spus, metodele clasei parinte sa fie generale si aplicabile tuturor claselor derivate fara a face modificari in ele anterior)

**Interface Segregation**

* Mai multe interfete specializate, nu o interfata generala cu multe metode
* Clasele nu trebuiesc obligate sa implementeze metode inutile

**Dependency Inversion**

* E mai bine sa ne folosim de tipul abstract decat de cel concret

*Conventii de nume*

* Upper Camel Case
* Lower Camel Case
* System Hungarian Notation -> pun si tipul variabilei in numele ei (ex: btnSave)
* Apps Hungarian Notation -> pun si modulul din care vine variabila in numele ei

Un nume trebuie construit astfel incat sa raspunda la urmatoarele intrebari: de ce exista? Ce face? Cum este folosit?

Unde denumirile prost alese nu pot fi evitate (metode auto generate), e indicat ca in interiorul lor sa fie doar apeluri de alte metode.

Daca denumirea unei metode contine o conjunctie, vorbim probabil de doua metode.

Nu e indicat ca variabilele string sa contina cod din alte limbaje de programare (sql, css).

Variabilele boolene trebuie sa sune ca o intrebare la care se poate raspunde cu true/false.

Cand exista variabile complementare, numele trebuie sa fie simetrice.

Declararea unei variabile pe linie noua.

Blocurile de instructiuni incep si se termina cu {}. Mai sunt marcate si prin identare. In limbajele compilate, acolada e pe acelasi rand cu antetul metodei; in limbajele interpretate, e pe urmatorul rand.

Acolada de inchidere a unui corp de instructiuni e singura pe linie, exceptie fac if-else si try-catch.

Metodele sunt separate printr-o singura linie goala.

Operatorii sunt separate de operanzi printr-un spatiu, inafara de operatorii unari(+=, -=).

*Clean code in structuri conditionale*

Trebuiesc evitate conditiile in care se compara o variabila booleana cu true/falsa; aceasta poate fi instantiate direct daca trece de o conditie sau nu. Prima ramura a if-ului sa fie pt cazul pozitiv.

Folosirea operatorilor ternari. A nu se compara direct cu string-uri, mai bine folosim enum-uri.

Constantele trebuie identificate si denumite la inceputul clasei.

Daca o conditie devine prea mare, mai bine se folosesc variabile intermediare.

De obicei, folosirea enum-ului denota un design gresit al clasei.

Multe constante indica o nevoie de inglobare a lor intr-o bd.

*Clean code in metode*

Orice metoda e indicat sa aiba maxim 3 nivele de structure imbrincate (arrow code).

Sa se iasa cat mai rapid din functie (cu return sau exceptie).

Variabilele sa fie declarate cat mai apropiat de folosirea lor.

Sa se foloseasca pe cat de mult posibil this si o conventie de nume pt parametrii constructorului.

Sa se evite metodele cu mai mult de 2 parametri.

Sa se evite metodele f lungi (>20 linii de cod) (one screen rule).

Complexitatea trebuie sa fie invers proportionala cu nr de linii de cod.

Atentie la ordinea tratarii exceptiilor.

Sa se verifice complexitatea ciclomatica a metodelor. Valoarea determina nr de teste care se vor efectua pe metoda. Metodele simple au complexitate = 1. Structurile de tip if si switch cresc complexitatea ciclomatica.

*Reguli simple de Clean Code pt metode*

* Single Responsibility (SRD)
* KISS
* Deleaga prin pointeri/interfete
* Foloseste interfete

O metoda proasta are multe linii de cod, face mai mult de un singur lucru, creste complexitatea, are multe dependinte.

*Reguli de Clean Code in clase*

Toate metodele dintr-o clasa trebuie sa aibe legatura cu clasa respective.

Sa se evite folosirea claselor generale (clasa Util) si sa se mute preucrarile respective ca metode statice in clasele aferente.

Sa se evite primitivele ca parametri sis a se foloseasca obiecte (prin clase Wrapper in Java). Atentie la ele si in metodele de pe mai multe fire de executie.

Sa se foloseasca fisiere de resurse pt string-urile din GUI (ca la Android).

Sa folosim design patters acolo unde cere o situatie.

Clasele care conlucreaza vor fi asezate una langa alta pe cat posibil.

*Reguli de Clean Code in comentarii*

* Codul bine scris e auto-explicativ; le folosim doar pt TODO comments sau biblioteci ce vor fi folosite de alti programatori (doc comments)
* Nu comentam cod nefolosit; exista solutii de versionare pt recuperarea codului modificat
* Se evita blocurile de comentarii introductive; toate detaliile se vor gasi in solutia de versionare

*Terminologie*

Test Driven Development (TDD) = dezvoltare bazata pe cazuri de utilizare

Refactoring = rescrierea codului intr-o maniera care se adapteaza mai bine noilor specificatii.

Automatic Testing (Unit Testing) = Testarea automata a codului pe baza unor cazuri de utilizare. Foarte utila in Refactoring, pt ca putem verifica daca am pastrat toate functionalitatile sau nu (regression).

Code Review = procedura intalnita in special in AGILE, care presupune revizuirea de un alt programator a codului.

Pair Programming = Tehnica specifica AGILE in care programatorii lucreaza in perechi la tasj-uri dificile, pt a invata sau pt a evita code review.

1. *Unit Testing – Junit*

Specificatii client -> Design si Arhitectura -> Dezvoltare & Codare -> Testare Software -> Implementare -> Mentananta & Suport

1. *Asertii*

Testarea = procesul de cautare a erorilor si defectelor.

* Utilizata pt a depista prezenta defectelor, dar nu garanteaza absenta acestora (Dijkstra)

Termini specifici: esec, defect, exceptie, problema, eroare, incident, anomalie, inconsistenta, aparenta, neajuns, bug.

Cauzele erorilor software: erori de programare, proiectare, de specificatii.

Unit Testing = o cale de testare a codului de catre programatori, inca din faza de dezvoltare a produsului software.

Unit Test = secventa de cod folosita pt testarea unei unitati bine definite din codul aplicatiei software. De obicei unitatea este o metoda.

Testarea unitatii se realizeaza intr-un context bine definit in specificatiile de testare.

Test Driven Development = dezvoltarea pe baza testelor. Este exact modul de gandire al oamenilor pt realizarea metodelor.

*Tipuri de testare*

1. **BlackBox / WhiteBox Testing**

**Testarea Black Box** (testarea comportamentala) = metoda utilizata pt a testa aplicatia software de catre persoane care nu cunosc arhitectura interna a aplicatiei testate.

Tester-ul cunoaste doar datele de intrare si datele de iesire ale aplicatiei.

Testele sunt realizate din punctul de vedere al utilizatorului.

Tester-ul nu trebuie sa stie progrmare, limbajul folosit pt dezvoltare sau structura de cod a aplicatiei.

Testele sunt realizate independent de dezvoltatori si au o perspectiva obiectiva.

Dezavantaje: - Cazurile de testare sunt dificil de proiectat, deoarece testerul nu are caietul de sarcini al aplicatiei

* Testele vor avea un nr mic de intrari
* Testele pot fi redundante cu alte teste realizate de dezvoltator

**Testarea White Box** = metoda de testare folisita de dezvoltatori sau de testeri care cunosc structura interna a aplicatiei testate.

Este cunoscuta si sub denumirile: Clear/Open/Glass/Transparent Box Testing, Code-Based/Structural Testing.

Testerul cunoaste si arhitectura aplicatiei testate si datele de intrare si iesire.

Avantaje: - Testarea poate fi inceputa intr-o etapa anterioara punerii in functiune. Nu trebuie sa se astepte realizarea interfetei pt a incepe testarea.

* Testarea e mai aprofundata, cu posibiliatea de a acoperi mai multe posibilitati.

Dezavantaje: - Din moment ce testale pot deveni complexe, sunt necesare resurse de inalta calitate, cu o cunoastere aprofundata a programarii si a punerii in aplicare

* Intretinerea codului de testare poate fi o povara in cazul in care punerea in aplicare se schimba f des

*Motive pt folosirea testarii unitare*

* usor de citit
* testele pot fi scrise ad-hoc cand ai nevoie de ele
* desi sunt simple, pe baza lor se pot define colectii de teste, Test Suites
* Pot fi rulate automat de fiecare data cand e nevoie (write once, use many times)
* Exista multe framework-uri si instrumente care simplifica procesul de scriere si rulare
* Reduc timpul pierdut pt debugging si pt gasirea bug-urilor
* Reduc nr de bug-uri in codul livrat sau integrat
* Creste rata bug-urilor identificate in faza de scriere a codului

*JUnit*

Este un framework ce permite realizarea si rularea de teste pt diferite metode din cadrul proiectului dezvoltat.

Este pt Java, este o adapatare a xUnit.

Instrument standard pt procesele de dezvoltare TDD in Java, integrat in multiple IDE-uri de Java.

Functioneaza conform a doua design patterns: Composite si Command.

O clasa TestCase = un obiect Command, iar o clasa TestSuite este compusa din mai multe instante TestCase sau TestSuite.

*Concepte Junit*

**Fixture** = set de obiecte utilizate in test.

**Test Case** = clasa ce defineste setul de obiecte (fixture) pt a rula mai multe teste.

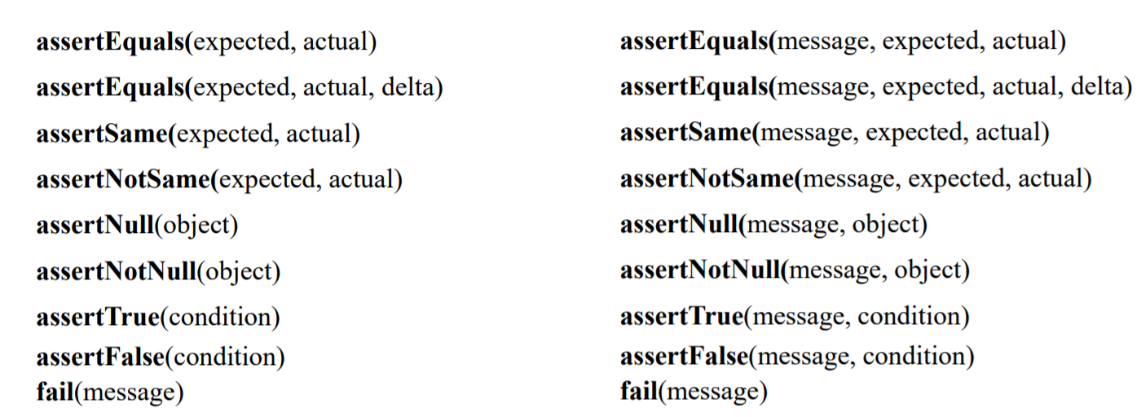
**Setup** = o metoda/etapa de definire a setului de obiecte utilizate (fixture), inainte de testare.

**Teardown** = o metoda/etapa de distrugere a obiectelor(fixture) dupa terminarea testelor.

**Test Suite** = colectie de cazuri de testare (test cases).

**Test Runner** = instrument de rulare a testelor (test suite) si de afisare a rezultatelor.

*Asertii*

**

*Junit Skeleton*

setUpBeforeClass()

setUp()

Test

tearDown()

…………………………..

setUp()

Test

tearDown()

tearDownAfterClass()

Adica metodele pe care le debifam si alegem sa le implementam singuri atunci cand cream un nou TestCase.

*Diferente JUint 3 si JUnit 4 (noi folosim JUnit 4)*

* Versiuni de JDK actualizate
* In JUnit 3 clasele de test sunt derivate din clasa TestCase, iar in JUnit 4 nu
* In JUint 3 numele metodelor de test este construit dupa sablobul testAAA; in JUnit 4 nu e important, dar metodele care sunt rulate ca teste au adnotarea @Test
* In JUnit 4 este folosita adnotarea timeout pt verificarea finalizarii testului intr-un interval de timp precizat, valoarea fiind in milisecunde (@Test(timeout=1000))
* Pentru ca un test sa nu fie rulat in JUnit 3, el trebuie sters, comentat sau numele lui modificat astfel incat sa nu respecte formatul testAAA; in JUnit 4 testul care nu se doreste a fi rulat primeste adnotarea @Ignore sau se sterge adnotarea @Test
* Adnoratile utilizate in JUnit 4 pt metodele automate skeleton sunt diferite:

*O imagine care conține interior, masă, pasăre

Descriere generată automat*

* In JUnit 3 nu exista adotari, deci numele metodelor setUp() si tearDown() sunt obligatorii

*JUnit 5*

Adnotarile pt metodele automate skeleton s-au modificat:

O imagine care conține captură de ecran, ținând, gata, verde

Descriere generată automat

1. *Right-BICEP & CORRECT*

Acum stim cum testam, dar ce testam? Cel mai cunoscut principiu de testare este **Right-BICEP.**

**RIGHT** -> daca rezultatele furnizate de metoda sunt corecte

**B -> Boundery**. Trebuie verificate toate limitele si daca in cazul acestor limite, rezultatele furnizate de metoda testata sunt corecte

**I -> Inverse**. Trebuie verificate relatiile inverse

**C -> Cross-Check**. Trebuie verificata corectitudinea printr-o verificare incrucisata, folosind metode de clacul asemanatoare, testate si validate de catre o comunitate mare de programatori

**E -> Errors**. Trebuie simulata si fortata obtinerea erorilor pt verificarea comportamentului metodei in cazul anumitor erori.

**P -> Performance**. Trebuie verificata pastrarea performantei intre limitele acceptantei pt produsul software final

*Right*

Prima data verificam corectitudinea rezultatelor returnate de metoda testate.

Se face in conform cu specificatiile proiectului dezvoltat.

*Boundery*

Trebuie testate metoda pt limitele intervalelor.

Pt fiecare metoda, trebuie determinat intervalul in care pot fi valorile parametrilor de intrare, precum si intervalul de rezultate furnizat de metoda.

Odata ce limitele intervalelor au fost determinate, se fac teste pe aceste valori.

Teste Boundary presupun verificarea corectitudinii acestor valori limita, nu a valorilor din afara limitelor.

De regula, exista limite inferioare si superior si testele se fac pe aceste valori.

Pentru a identifica mai usor limitele extreme, putem sa utilizam principiul CORRECT:

C -> Conformance

O -> Ordering

R -> Range

R -> References

E -> Existence

C -> Cardinality

T -> Time

*Inverse Relationship*

Anumite metode pot fi testate prin aplicarea reguli inverse: pornind de la un rezultat, trebuie sa se ajunga la aceeasi intrare de la care s-a pornit initial.

De obicei se aplica metodelor matematice.

Pt o baza de date se poate verifica daca s-a efectuat o operatie de inserare prin operatia inversa, select.

*Cross-Check*

Pentru fiecare metoda, putem incerca sa o testam folosind o alta metoda.

De obicei, exista mai multe modalitati de a rezolva o problema. Astfel, se poate utiliza o alta metoda pt rezolvarea problemei pt verificarea/testarea metodei nou implementate.

Aceasta situatie este posibila atunci cand metoda implementata a fost conceputa pt a creste productivitatea sau daca metoda veche consuma prea multe resurse.

Testarea metodei noi se face prin metoda veche, chiar daca aceasta consuma mai multe resurse.

*Error Conditions*

Atunci cand testam fiecare metoda unitar, testam si situatiile in care aceasta ar putea sa crape.

Daca am efectuat deja la testele de Boundary teste asupra valorilor extreme pt valorile de intrare sau valorile rezultat, testarea pt furnizarea erorilor ar trebui sa utilizeze valori inafara acestor intervale.

Aceasta metoda se face pt toate metodele. Toate metodele au cel putin o situatie in care vor oferi erori. Testarea se face pt aceste situatii si se verifica daca metoda trateaza acel caz si arunca sau ofera o exceptie.

*Performance*

Pentru diferite metode, este posibil sa se testeze cat de bine functioneaza metoda respectiva.

Pe langa corectitudinea rezultatelor returnate de metoda, este important sa se verifice si performanta procesarii.

Se verifica consumul de resurse si timpul necesar obtinerii rezultatelor.

Acest test se face cand parametrii de intrare sau rezultatele sunt de forma unei liste sau a unui nr foarte mare de elemente, iar aceste valori pot creste f mult.

In JUnit 4, pt a testa timpul necesar rularii unei metode, se foloseste adnotarea timeout.

*CORRECT*

C -> Conformitatea formatului

O -> Ordinea

R -> Intervalul

R -> Referinte externe

E -> Existenta obiectelor sau a rezultatelor

C -> Cardinalitatea rezultatelor

T -> Timpul

**Conformance**

Cunsocut si sub numele de: Type Testing, Compliance Testing, Conformity assessment.

Se aplica in domenii unde trebuie sa se aplice anumite standard specifice.

Pt orice intrare sau iesire, trebuie sa se verifice conformitatea cu un standard.

Teste pot fi effectuate pt a verifica ce se intampla daca daca datele de intrare nu sunt in conformitate cu formatul sau daca rezultatul obtinut este in conformitate cu specificatiile proiectului.

**Ordering**

Sunt specific listelor, dar nu numai.

In cazul listelor, trebuie sa se verifice daca ordinea elementelor este cea dorita.

De asemenea, putem testa comportamentul metodei daca primeste anumiti parametri intr-o alta ordine sau o lista de elemente intr-o ordine diferita de cea asteptata.

**Range**

Trebuie verificate intervalele pt valorile de intrare si iesire. Pt anumite metode, exista mai multe interval, se vor testa toate aceste intervale.

Toate functiile care au un index trebuie sa fie testate pt interval, deoarece acel index are un domeniu bine stabilit.

De obicei, trebuiesc verificate:

* Daca valorile initiale si finale pt index au aceeasi valoare
* Primul element este mai mare sau mai mic decat ultimul element
* Ce se intampla daca indicele este negative
* Ce se intampla daca indicele e mai mare decat limita superioara
* Nr de articole nu este acelasi cu cel pe care-l doresc – dimensiunea

**Reference**

Anumite metode depind de lucruri sau metode externe lor. Aceste elemente trebuiesc verificate si controlate.

Ex: o extragere din stiva functioneaza daca exista elemente in stiva; o aplicatie web necesita conectarea utilizatorului etc.

Aceste elemente sunt numite preconditii sau conditii preliminare. Ele sunt necesare pt ca metoda sa functioneaza ok.

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

**Existence**

Ce se intampla cu metoda daca un parametru e null, 0 sau nu exista.

Pt sistemele software care functioneaza folosind fisiere externe sau conexiunea la internet, e necesara verificarea existentei fisierelor sau a conexiunii. Daca nu exista, utilizatorul va fi avertizat de problema intampinata si aplicatia va functiona normal, nu va da eroare.

Asemanatoare cu **conditia de eroare din Right-BICEP**.

**Cardinality**

0-1-n rule

Similar cu testele Existence si Range.

Trebuie sa verificam daca metoda/lista/colectia are 0, 1 sau n elemente.

Daca functioneaza pt 2, 3, 3 elemente, se considera ca functioneaza pt mai multe elemente. Sa nu uitam de **Boundery superior**.

**Time**

Similar cu testul de Performance din Right-BICEP.

Poate fi folosit si pt a testa daca sablonul de apeluri e respectat. Similar cu design pattern-ul **Template**.

Ex: Pt a apela metode de logout(), trebuie mai intai apelata metoda de conectare().

*FIRST*

Fast

Isolated/Independent

Repetable

Self-Validating

Timely

Testele unitare trebuie sa fie FIRST, pt a fi utile si eficiente.

**Fast**

Testul trebuie sa fie rapid, pt ca daca avem multe teste, nu trebuie sa asteptam mult dupa ele sa fie executate.

**Isolated**

**Single Responsibility din SOLID**

Fiecare test trebuie sa aibe doar un motiv sa pice.

Testul trebuie sa fie izolat sis a spuna exact unde e problema si care e problema, nu sa trebuiasca sa facem debug ca sa descoperim.

**Repetable**

Rezultatele obtinute sa fie identice indifferent de cate ori rulam testele.

Testele sa se desfasoare in mod repetat, fara alte interventii.

**Self-Validating**

Incredere in testele scrise. Daca testul nu trece, metoda e gresita, nu testul. Daca metoda trece, sa nu avem dubii ca nu e corecta.

**Timely**

Punem in aplicare testele oricand pe parcursul dezvoltarii.

Am facut majoritatea testelor cand am acoperit toata complexitatea ciclomatica, adica toate if-urile si switch-urile.

1. *Dubluri de testare*

In testarea automata se folosesc obiecte care arata si se comporta ca echivalentele lor din productie, dar sunt de fapt simplificate.

Acest lucru reduce complexitatea, permite verificarea codului independent de restul sistemului si, uneori, este chiar necesar sa se efectueze teste de auto-validare.

Ele se numesc dubluri de testare.

O dublura = un obiect care se potriveste cu interfata colaboratorului necesar si poate fi trecut in locul sau.

*Tipuri de dubluri de testare*

Dummy Object = un obiect care respecta interfata, dar metodele nu fac nimic sau returneaza 0 sau null.

Stub = spre deosebire de Dummies, metodele vor returna raspunsuri conservate/hardcodate.

Spy = un Stub sau un Fake care gestioneaza si contorizeaza nr de apeluri.

Fake = un obiect care se comporta asemanator cu unul real, dar are o versiune simplificata

Mock = diferit de toate celelalte.

**Obiecte Dummy**

Cand trebuie sa folosim in cadrul testelor obiectul real, de fapt folosim un Dummy.

Un dummy e folosit cand nu trebuie sa apelam metodele din acel obiect, pt ca nu fac nimic.

In ex, avem nevoie de obiectul dummy care implementeaza interfata IPersoana, doar pt a initializa obiecte asemanatoare cu cele reale, Persoana, dar nu apelam nicio metoda din aceasta dublura.

**Obiecte Stub**

Putem folosi aceste obiecte pt ca au apeluri reale.

Le folosim cand testam metode care necesita apelul unei metode din clasa reala Persoana.

Avem doua faze: definim obiectul si facem asertie, nu avem control asupra valorilor atributelor.

**Obiecte Fake**

O varianta mai simplificata a obiectului real.

Putem stabili ce valoare sa intoarca, deci nu sunt valori hardcodate.

Are setter si getteri.

Aici avem 3 faze la testare: definim obiect, ii setam valorile cu ajutorul setterilor si apoi facem asertie.

**Obiecte Spy**

Este un Stub sau un Fake care gestioneaza si nr de apeluri realizate pt metodele acestui obiect.

Altfel spus, in getter mai intai incrementez valoarea si apoi o returnez.

*Mock Testing*

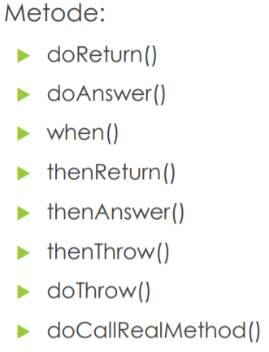
Mock testing = utilizat atunci cand dorim ca metoda testata sa nu fie influentata de referinte externe.

Mock Object (obiect mock-uit) = un obiect care simuleaza comportamentul unui obiect real, dar intr-un mod controlat.

Cele mai folosite framework-uri sunt Mokito si EasyMock.

O imagine care conține interior, pasăre, masă

Descriere generată automat



Folosesc metodele astea in cadrul testului. Nu mai fac clasa separate ca si la dubluri, ci folosesc direct metodele ca sa mock-uiesc un obiect si apoi sa-I setez comportamentul.

*Utilizare fisierelor pt datele de test*

Se va realiza un singur TestCase cu metoda de assert apelata intr-un loop.

Pt fiecare set de date de test din fisier, se apeleaza metoda assert.

Fisierul sau fluxul este considerat un fixture si de aceea trebuie deschis in setUp() si inchis in tearDown().

*Custom TestSuite in JUnit 3*

O imagine care conține captură de ecran, pasăre

Descriere generată automat

O imagine care conține interior, pasăre

Descriere generată automat

*Custom TestSuite in JUint 4*

Se implementeaza o clasa sau o interfata CustomSuite.

Pt fiecare test droit sa faca parte din aceasta suita, se adauga adnotarea:

@Category(CustomSuite.class)

Pt crearea unei suite custom, sunt incluse toate categoriile dorite.

O imagine care conține captură de ecran, pasăre

Descriere generată automat

*JUint 5*

* In asertii, paramatrul optional, adica acel mesaj, se afla pe ultima pozitie in lista de parametri.
* Adnotarile pt metodele automate din skeleton s-au schimbat -> tabelul verde.
* Pt testul in care fortam aparitia unei erori, Error din Right-BICEP, nu mai folosim parametrul expected din adnotarea @Test ca si in JUnit 4, ci apare assertThrows().

O imagine care conține captură de ecran, pasăre

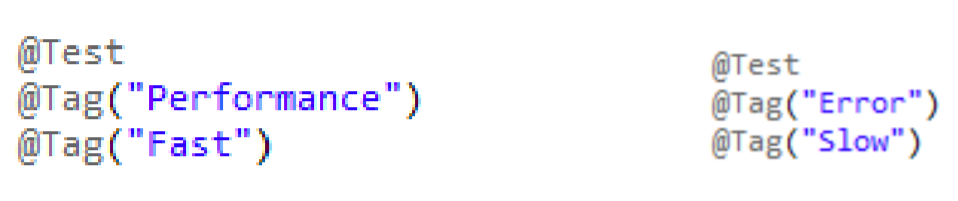
Descriere generată automat

* Pentru testul de performanta, nu se mai foloseste parametrul timeout din adnotarea @Test, ci apare assertTimeout().

O imagine care conține pasăre

Descriere generată automat

* @Tag pt suitele custom.



Avem nevoie si de Integartion Tests cand avem mai mult de un Unit Test.

1. *Design Patterns*

-folosite pt intelegerea mai buna a codului sursa scris

Specificatii client -> Design si Arhitectura -> Dezvoltare & Codare -> Testare Software -> Implementare -> Mentananta & Suport

= Sabloane/tipare de proiectare folosite in rezolvarea unor probleme similare cu alte probleme pt care au fost deja descoperite rezolvari.

Au aparut ca urmare a clasificarii tipurilor de probleme si s-a observat ca multe probleme se rezolva in acelasi mod sau urmand aceeasi serie de pasi.

“Gang of Four Book”

**Avantaje Design Patterns** -> de scris mai tarziu

Sa nu uit de cartea lui Zamfi!!

**Dezavantaje Design Patterns**

**Pasi**

* Identificare problema
* Mapare design pattern -> problema
* Identificare participant
* Alegerea numelor participantilor
* Implemnetarea interfetelor si a claselor
* Implementarea metodelor

**Componente Design Pattern** -> scriu mai tarziu

**Tipuri de design patterns**

1. Design patterns **creationale**

* Singleton
* Builder
* Factory
* Factory method
* Abstract factory -> nu la examen
* Prototype

1. Design patterns **structurale**

Contribuie la compozitia claselor si a obiectelor, realizand decuplarea interfetelor de clase.

* Adapter
* Bridge
* Composite
* Decorator
* Façade
* Flyweight
* Proxy

1. Design patterns **comportamentale**

* Chain of responsibility
* Command
* Iterator -> nu
* Interpreter -> nu
* Mediator -> nu
* Memento
* Observer
* State
* Strategy
* Visitor -> nu
* Template

*Design Patters creationale*

Ajuta la initializarea si configurarea claselor si obiectelor.

Separa crearea obiectelor de utilizarea lor concreta. De aici si numele lor de creationale.

Ofera o foarte mare flexibilitate in ceea ce priveste: cine este creat, cine creaza obiectul, cum este creat, cand este creat.

**Singleton**

**Nume**

Numele sugereaza ca putem avea un singur obiect de tipul clasei respective: Single.

Clasa este cea care se ocupa de faptul ca poate fi create un singur obiect. Nu lasa acest lucru pt utilizatori sau apelatori.

Sintagma cheie pt utilizarea acestui design pattern este: **instanta unica** sau **o singura instanta**.

Problema