Laborator Nr 3

Tema : Unit testing, Dependency Injection, Mockups

Scopul: parcurgerea etapelor de testare si deployment din ciclul de dezvoltare a unui program si folosirea unor tool-uri Java specifice acestor etape.

A se susține pe 19/10/2020, orice întârziere se taxează cu -2 de la nota primita.

Subiecte atinse:

- Agile Software Development
- Extreme Programming (XP)
- Test-Driven Development
- Unit Testing
- JUnit
- Stubs
- Dependency Injection
- Mockups

Agile Software Development

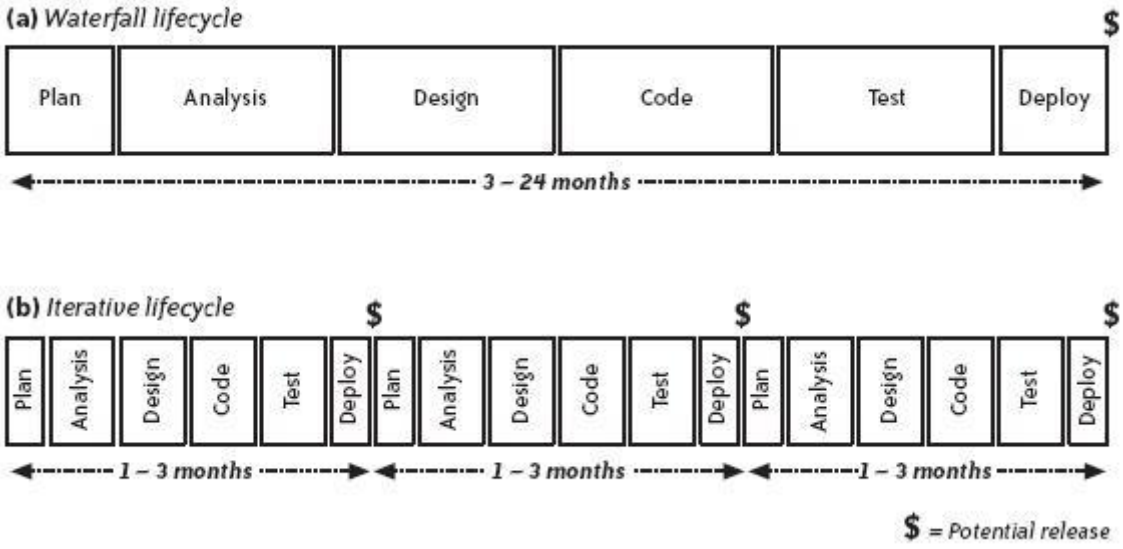
Un model **traditional**, foarte raspandit in dezvoltarea de sisteme software, este modelul **in cascada** (*waterfall model*). El propune secventierea unor etape cu scop distinct, de la formularea cerintelor, pana la furnizarea produsului final catre client. Principalele neajunsuri ale acestei abordari sunt:

- asamblarea **tarzie** a sistemului final. Daca, din diferite motive, derularea proiectului depaseste durata preconizata, nu exista nici o forma consistenta a produsului ce poate fi prezentata clientului la deadline-ul stabilit.
- dificultatea integrarii **schimbarilor**. Daca, intr-un anumit moment, cerintele existente se schimba, etapele deja incheiate trebuie, de cele mai multe ori, refacute de la zero.

Cele doua probleme de mai sus sunt, oarecum, firesti:

- un grad satisfacator de **control** se poate obtine doar asupra unui sistem suficient de **palpabil**. In conditiile in care componentele, dezvoltate independent, sunt integrate de-abia la sfarsit, imaginea de ansamblu este, practic, inexistenta.
- procesele naturale sunt inerent **dinamice** si **iterative**, astfel incat orice demers trebuie realizat veghind asupra posibilelor schimbari. In acelasi mod, nu este neobisnuit ca, pe masura ce dezvoltarea sistemului avanseaza, sa transpara anumite inconsistente ale cerintelor initiale, sau nevoia de cerinte noi.

Metodele **agile** de dezvoltare (*agile software development*) au aparut ca o **replica** la demersul clasic, din nevoia de a adresa o serie de probleme, cum sunt cele de mai sus. Prin contrast cu modelul cascada, metodele agile propun o dezvoltare **iterativa**, in etape scurte, la finele carora este furnizata cate o varianta **functionala** a produsului, chiar daca aceasta solutioneaza doar un subset al cerintelor. Astfel, se urmareste generarea de **valoare** inca din primele stadii ale procesului de dezvoltare (prin *valoare* intelegem *ceva pentru care clientul este dispus sa plateasca*).



Putem trasa cateva **caracteristici** ale acestei metodologii:

- utilizarea de iteratii **scurte** (*releases*), in care diferitele etape de formulare a cerintelor, design, implementare si testare se **intrepatrund** (vezi Fig. 1)
- **integrarea** continua a componentelor constitutive din proiect
- intrebuintarea unor **practici** avansate de dezvoltare
- **colaborarea** directa si permanenta a dezvoltatorilor cu clientii (*cross-functional*)
- incurajarea **autoorganizarii** echipelor si a **responsabilizarii**, in locul unor reguli si documente stricte care sa normeze comunicarea
- **alinierea** procesului de dezvoltare cu nevoile clientilor si obiectivele organizatiei
- **revizuirea** si **imbunatatirea** permanenta a muncii
- cautarea **simplitatii** in rezolvarea problemelor
- acceptarea si usurinta in integrarea **schimbarilor**.

Mergand mai departe, echipele experimentate in implementarea unei metodologii agile pot chiar **urmari** aparitia de oportunitati, pe care sa le integreze in demersul lor.

Agilitatea este un concept **cuprinzator**, ce nu se adreseaza numai dezvoltatorilor, ci **tuturor** participantilor la proiect: clienti, project manager, programatori, testeri. In cele ce urmeaza, ne vom concentra pe implicatiile asupra echipei de dezvoltare.

De remarcat ca aceasta abordare este asimilata adesea unei dezvoltari rapide si superficiale. Aceasta asociere provine dintr-o mica eroare de perceptie: viteza de dezvoltare este o **consecinta** a metodologiei propuse, si **nu** obiectivul principal al echipei!

Puteti citi crezul Agile.

Extreme Programming (XP)

XP reprezinta cea mai cunoscuta tehnica **agila**. Dupa cum ii spune si numele, se refera la exacerbarea unor practici utile:

- *If **testing** is good, let everybody test all the time*
- *If **code reviews** are good, review all the time*
- *If **design** is good, **refactor** all the time*
- *If **integration** testing is good, integrate all the time*
- *If **simplicity** is good, do the simplest thing that could possibly work*
- *If **short iterations** are good, make them really, really short*

XP se bazeaza, intr-o masura deosebita, pe o abordare ***test-driven***, atat in cazul **dezvoltatorilor** (*unit & integration tests*), cat si al **clientilor** (*customer/acceptance tests*). Vom vorbi despre aceasta mai departe.

O alta caracteristica este lucrul in echipe de **doi** programatori la un calculator (***pair programming***): unul se poate concentra pe rezolvarea unei chestiuni punctuale, in timp ce al doilea poate pastra o perspectiva de ansamblu asupra sarcinii de realizat. Programarea in perechi are avantajul ca orice portiune de cod este inspectata de cel putin doi oameni, fapt ce sporeste calitatea codului scris. In cazul in care autorul paraseste echipa, exista o rezerva.

Test-Driven Development (TDD)

TDD propune scrierea testului pentru o functionalitate ce se doreste adaugata **inaintea** implementarii acesteia. Astfel, adaugarea unui nou comportament este **declansata** intotdeauna de esecul unui test: *write code so the test passes*. Drumul urmat ar trebui sa fie **cel mai scurt** pentru obtinerea rezultatului dorit, fara a implica prea multe considerente de design. Ulterior, dupa executia cu succes a testului, se poate **reveni** pentru imbunatatirea design-ului (*refactoring*), **fara** a modifica functionalitatea existenta. **Pasii** in adaugarea de noi functionalitati ar trebui sa fie suficient de **mici**.

TDD presupune executarea de **cicluri**, avand urmatoarele etape:

1. scrierea **testului**: imaginarea unui scenariu de utilizare a noii functionalitati
2. verificarea **esecului** testului, in caz contrar fiind inutil
3. implementarea **functionalitatii**, pe drumul **cel mai rapid**
4. verificarea **succesului** testului
5. imbunatatirea **design**-ului, cu asigurarea faptului ca testele trec si dupa modificarile efectuate
6. **repetarea** ciclului pentru o noua functionalitate

Testele ar trebui scrise dintr-o perspectiva **comportamentala**, privind, de exemplu, interfata unei clase, fara sa vizeze aspecte specifice implementarii. Acest lucru are un dublu **avantaj**:

- programatorul intelege de la bun **inceput** tinta sa finala
- acesta se va concentra, in mod firesc, pe definirea unor interfete usor de **folosit** (si de… testat), mai mult decat de implementat, lucru ce sporeste **calitatea** design-ului

Dupa implementare, testele vor servi drept:

- **documentare** a functionalitatii respective, oferind exemple de utilizare
- modalitate de **garantare** a faptului ca alte comportamente, mai noi, **nu** altereaza functionalitatile deja implementate (*regression testing*). Dupa **adaugarea** unei functionalitati, se ruleaza, din nou, **intreaga** suita de teste, inclusiv cele care testeaza comportamente anterioare.

Unit Testing

Unit testing-ul reprezinta procedeul de testare a unitatilor elementare din program. In abordarea orientata obiect, acestea reprezinta metodele unei clase. Ideea este **izolarea** fiecarui modul de functionalitate a unui program si asigurarea corectitudinii **individuale**. In afara de unit testing, se mai intalnesc:

- **integration testing** (analizeaza comunicarea intre mai multe module)
- **acceptance testing** (reprezinta testele beneficiarului produsului, la nivelul specificatiei).

Avantaje majore:

- inlesneste **schimbarile**: o data scrisa, suita de teste poate fi rulata la fiecare modificare a codului sursa, pentru a investiga aparitia unor bug-uri la adaugarea de functionalitate (*regression testing*)
- simplifica **integrarea** modulelor, prin asigurarea validitatii **individuale**
- **documenteaza** functionalitatea: testele ofera exemple de utilizare a modulului pentru dezvolatorii nefamiliarizati
- reprezinta o modalitate de **design**: testele pot fi scrise inaintea implementarii, fapt ce garanteaza intelegerea functionalitatii de catre dezvoltator
- incurajeaza proiectarea **modulara**: crearea unor unitati mici ce pot fi testate individual.

JUnit

Reprezinta un framework de unit testing pentru Java, disponibil aici. Documentatia poate fi accesata aici.

Pentru utilizare este necesara prezenta fisierului `junit.jar`. Pentru rulare in linia de comanda, este utila adaugarea caii catre acest fisier la variabila de mediu `CLASSPATH`:

```
setenv CLASSPATH ~/java/lib/junit.jar:.:$CLASSPATH
```

sau

```
export CLASSPATH=~/java/lib/junit.jar:$CLASSPATH
```

Exemplu

Iata un prim exemplu de clasa de test, care verifica (minimal) functionalitatea clasei `ArrayList`:

ListTest.java

```
import java.util.List;
import java.util.ArrayList;

import junit.framework.TestCase;

public class ListTest extends TestCase {
    private List<Integer> emptyList;
    private List<Integer> filledList;

    @Override
    protected void setUp() {
  emptyList = new ArrayList<Integer>();

  filledList = new ArrayList<Integer>();
  filledList.add(1);
  filledList.add(2);
  filledList.add(3);
    }

    public void testContains() {
  assertTrue("filledList must contain 1", filledList.contains(1));
  assertFalse("emptyList must not contain 1", emptyList.contains(1));
    }

    public void testRemoveElement() {
  filledList.remove(new Integer(3)); // .remove(3) would mean position 3, not the
element 3
  assertFalse("filledList must not contain 3", filledList.contains(3));
    }

    public void testAddElement() {
  fail("Not yet implemented");
    }
```

```
}
```

Din cele de mai sus, se observa:

- clasa de test trebuie sa **mosteneasca** `TestCase`, iar numele acesteia se termina, conventional, cu sirul „`Test`"
- **numele** metodelor de test incep cu sirul „`test`"
- pentru **compararea** rezultatului asteptat cu rezultatul curent se folosesc apeluri `assert`. Alte variante des intalnite sunt `assertEquals`, `assertNull`, `assertNotNull` etc. Aceste functii permit precizarea unor mesaje (in variantele cu 3 parametri, ca `assertEquals`) ce vor fi afisate la incalcarea asertiunii cu pricina
- se poate **forta** picarea unui test, folosind metoda `fail`. In cazul functiei `testAddElement`, absenta acestui apel ar conduce la rularea cu succes a testului, fara ca el sa fie implementat!
- metodele `setUp` si `tearDown` se apeleaza inaintea, respectiv dupa **fiecare** test. Se intrebuinteaza pentru initializarea/eliberarea resurselor ce constituie mediul de testare, evitandu-se, totodata, **duplicarea** codului, si respectandu-se principiul de **independenta** a testelor.

Detalii

Implicit, `JUnit` foloseste *reflection* pentru a determina, pe baza numelui, metodele ce reprezinta teste. Programatorul poate preciza, explicit, testele ce se doresc rulate, dar nu vom acoperi acest aspect.

In decursul rularii, sistemul extrage informatii despre **problemele** aparute. Acestea se inscriu in 2 categorii:

- ***failures***: conditii nesatisfacute (*failed assertions*). Acestea sunt de asteptat, provenind din nerespectarea specificatiilor. Este vorba de cazul „obisnuit" de picare a unui test.
- ***errors***: probleme neasteptate, cum sunt exceptiile.

Pentru a adauga, in **Eclipse**, o astfel de clasa de test: click dreapta pe proiect → `New` → `JUnit Test Case`. Eclipse poseda propriul sistem de vizualizare a rezultatelor.

Rulare folosind Ant

`Ant` permite automatizarea activitatii de testare. Fisierul `build.xml` poate contine un target definit astfel:

build.xml

```
<target name="test" depends="compile" description="Run JUnit tests">
    <junit haltonfailure="false" haltonerror="false" printsummary="withOutAndErr">
        <classpath refid="classpath" />
        <batchtest>
            <fileset dir="${src.dir}" includes="**/*Test*.java" />
        </batchtest>
    </junit>
</target>
```

Acesta va cauta metode de test in toate clasele al caror nume contine sirul „`Test`". Pentru afisarea output-ului in format HTML, se fac modificarile de mai jos, unde `tmp.dir` reprezinta o proprietate ce stocheaza calea catre un director temporar:

build.xml

```
<target name="test" depends="compile" description="Run JUnit tests">
    <junit haltonfailure="false" haltonerror="false" printsummary="withOutAndErr">
        <classpath refid="classpath" />
        <batchtest todir="${tmp.dir}">
            <fileset dir="${src.dir}" includes="**/*Test*.java" />
        </batchtest>
        <formatter type="xml" />
```

```
        </junit>
        <junitreport todir="${tmp.dir}">
            <fileset dir="${tmp.dir}" includes="TEST-*.xml" />
            <report format="frames" todir="${tmp.dir}" />
        </junitreport>
    </target>
```

Pentru a vizualiza rezultatele, deschideti fisierul `index.html` din directorul referit de `tmp.dir`.

Stubs

In cadrul proiectelor de mare amploare, poate aparea necesitatea testarii unor componente care nu sunt finalizate. Pentru aceasta se pot folosi interfete numite **Stubs**, care simuleaza functiile de baza ale componentei respective fara sa efectueze insa si teste de integritate a datelor. Astfel de interfete sunt des folosite la dezvoltarea unitatilor unui proiect care depind de componentele simulate.

Mockups

Spre deosebire de Stubs, **Mockups** reprezinta implementarea unor interfete care testeaza aprofundat functiile oferite de interfete. Cu ajutorul mockup-urilor se poate simula functionalitate unui server pentru testarea clientilor. Pentru o utilizare mai facila se recomanda folosirea interfetelor si utilizarea lor in functia de testare. Mockup-urile sunt utile in multe situatii precum:

- componenta nu exista sau este incomplet implementata
- dureaza prea mult rularea componentei reale
- functia reala returneaza valori nedeterministe si se doreste sa se testeze comportarea cu toate valorile limita
- functia reala necesita interactiunea cu utilizatorul

Dependency Injection

Dependecy Injection este procesul prin care sunt furnizate dependinte externe unei componente software. Componenta software nu isi creaza dependintele explicit ci le primeste ca si parametrii la creare.

Daca se urmeaza modelul Dependency Injection, codul este mai usor de testat deoarece va permite injectarea de instante `false/mock-up`.

Clasa `User` implementată mai jos nu respectă principiul DI deoarece nu permite injectarea unei instante a clasei `DbClient` ci isi creeaza propria instanta in constructor:

```
class User {
 int id;
 String username;
 IDbClient client;
 User(int id) {
   this.id = id;
   this.client = new DbClient();
   this.username = client.getUsernameById(id);
 }
}
```

Intrucat `DbClient` este creată intern, atunci când se vor rula testele pentru clasa `User` nu se va putea injecta o implementare mai simplă (o machetă - mock-up) a lui `DbClient`. Astfel toate testele pentru `User` vor depinde de capabilitatea de a crea o conexiune la baza de date, vor consuma bandă în mod inutil si vor incetini procesul de testare.

O implementare imbunatatita:

```
class User {
 int id;
 String username;
 IDbClient dbClient;
 User(IDbClient client, int id) {
   this.id = id;
   this.username = client.getUsernameById(id);
 }
}
```

Acum putem crea obiecte de test cu

```
x = new User( new MockupDbClient(), 5);
```

injectând în locul lui `DbClient` o versiune mai simplă care întoarce acelasi nume de utilizator

```
class MockupDbClient implements IDbClient {
  public String getUsernameById(int id) {
      return "Arthur";
  }
}
```

Principii generale ale Dependency Injection:

- o clasa cere doar obiectele cu care lucreaza in mod direct (nu obiecte care sa creeze obiectele cu care va lucra),
- folosirea unor constructii de tipul `a.getX().getY()` indica o dependinta incorect injectata,
- se injecteaza ca parametri in constructori obiecte a caror durata de viata e similara(sau mai scurta) cu cea a obiectului in care se injecteaza

Pentru testarea automata, exista o serie de containere care sunt capabile sa automatizeze injectarea dependintelor prin delegare. De regula acest lucru se face folosind XML sau definitii de metadate. Spre exemplu, pentru cazul de mai sus daca folosim in framework extern definitia XML va arata similar cu:

```
    <container>
        <environment name="production">
                <dependency interface="IDbClient" class="DbClient"/>
        </environment>
        <environment name="test">
                <dependency interface="IDbClient" class="MockedDbClient"/>
        </environment>
    </container>
```

In interiorul aplicatiei, serviciul de dependency injection va determina ce tip de instanta sa foloseasca in functie de mediul in care este folosit:

```
public class MyApplication {
    public static void main(String[] args) {
        DependencyManager manager = new DependencyManager("test");
        IDbClient client =  manager.create(IDbClient.class);
    }
}
```

Mersul lucrarii:

- Utilizati scheletul de laborator (fisierul TIDPP_Lab_4_skel.rar)
- Importati proiectul in Eclipse (File→Import→General→Existing projects in workspace).

1. (2.5p) **Familiarizarea** cu instrumentul `JUnit`.

    - Porniti de la clasa `Money`. Metoda `add` poate fi testata folosind o clasa simpla de test, `MoneyTest`. Studiati sursele si rulati testul. Care este rezultatul?
    - Modificati rezultatul asteptat (in test) la o alta valoare decat cea asteptata (26). Rulati din nou testul. Care este acum rezultatul?
    - Considerand, in continuare, rezultatul asteptat ca fiind 26, actualizati clasa `MoneyTest` astfel incat sa apeleze metoda `assertEquals` pe cele 2 obiecte. Rulati din nou testul. Care este rezultatul in acest caz? De ce? Faceti modificarile necesare in clasa `Money` astfel incat rezultatul sa fie cel dorit.

2. (5p) ***Black-box testing***. Acest exercitiu urmareste identificarea unor cazuri de test, strict pe baza **specificatiei**, in **absenta** accesului la codul sursa si a cunoasterii modului intern de functionare a sistemului.

    - **Specificatia** (partiala) a programului de testat este:
        - Se considera o clasa Java, numita `Triangle`. Aceasta este instantiata pornind de la **trei** valori intregi, reprezentand lungimile celor trei **laturi** ale unui **triunghi**.
            - Daca argumentele constructorului nu desemneaza laturile unui triunghi valid, acesta va arunca `InvalidTriangleException`.
        - Metodele `isScalene`, `isIsosceles` si `isEquilateral` au drept scop evaluarea starii obiectului `Triangle`.
            - Va reamintim ca un triunghi isoscel (en. *isosceles*) este acel triunghi avand doua laturi egale. Un triunghi echilateral (en. *equilateral*) are toate cele trei laturi egale.
            - Metoda `isIsosceles` intoarce `true` daca triunghiul este isoscel, fara a fi echilateral.
            - Metoda `isScalene` intoarce `true` daca triunghiul nu este nici isoscel, nici echilateral.
    - Creati un **scenariu de testare** pentru aceasta clasa, prin implementarea propriilor cazuri de testare, intr-o clasa `TriangleTest`, ce extinde `TestCase`.
        - **Atentie**: Nu veti crea voi insiva o implementare a clasei `Triangle`, ci veti folosi clasele furnizate in cadrul laboratorului. Veti presupune ca tipul `Triangle` exista deja, expunand interfata:

```
public Triangle(int a, int b, int c) throws InvalidTriangleException
public boolean isScalene()
public boolean isIsosceles()
public boolean isEquilateral()
```

        - Testati ca **exceptia** este aruncata, la pasarea unor argumente necorespunzatoare. **Hint**: metoda `fail`.
        - Construiti teste **specializate**, orientate pe o anumita functionalitate. De exemplu, in cadrul unui test, verificati doar una din cele 3 metode, eventual prin mai multe apeluri, si nu o combinatie a tuturor 3, deoarece cauzele erorilor isi pierd localitatea.
    - Redenumiti, pe rand, **fiecare** din fisierele `ErrorXTriangle.class`, in `Triangle.class` (in total vor fi 4 seturi de teste) si puneti-le pe build path-ul proiectului (folderul lib/).
    - Asigurati-va ca testele identifica toate problemele din toate implementarile clasei `Triangle`. **Hint**: `Error1Triangle` are una dintre metode gresit implementata. Celelalte permit diverse tipuri de triunghiuri invalide.

3. (`2.5p`) **Dependency Injection** Studiati clasa House, Locator-ul asociat si clasa de test, HouseTest.

    - Rescrieti clasa Locator astfel incat sa respecte principiul de Dependency Injection si adaptati clasa HouseTest sa foloseasca noua implementare.