* **,Unit test** – a testa o unitate de cod, o clasa, o functie etc. izolate de restul programului, adica fara a atinge alte componente. Uneori un unit test poate testa si 2 unitati, daca ele sunt legate.
* Unit tests nu e suficient pentru a testa aplicatie. Unit tests ne permit sa ne asiguram ca clasele lucreaza bine izolate, dar nu si impreuna.
* Testele sunt bune ca daca am modificat ceva in program, suntem siguri ca nu l-am stricat
* **Coverege e de dorit sa fie in jur de 80%. Totusi, coverege nu e prea de incredere, caci daca am executat o metoda undeva prin teste, nu inseamna neaparat ca anume am verificat metoda data, ci poate pur si simplu am executat-o pe undeva fara a testa ceva din ea.**

**TDD**

* **Test Driven**-**Development** – intai scriem cate un test care cade, apoi scriem codul ca testul dat sa nu cada.
* Deci, scriem un test care cade si pe baza lui sriem codul ca testul sa nu cada.
* **TTD** se bazeaza pe 3 reguli:

1. Nu scriem nici-un rand de production code atata timp cat nu am scris un test care cade
2. Ne oprim sa scriem unit testul exact cand el nu trece.
3. Scriem doar atata cod cat e necesar pentru ca testul sa treaca, apoi scriem iar un test ce cade.

* Se considera ca codul facut cu TDD e mai calitativ si simplu de inteles.
* In plus asa deodata avem testele pentru code.

**@TestInstance(TestInstance.Lifecycle.TIPULALES)**

* Junit creaza cate o noua instanta a clasei pentru fiecare test.
* Totusi, putem modifica asta punand anotatia data deasupra clasei de testare:

@TestInstance(TestInstance.Lifecycle.*PER\_CLASS*)

@TestInstance(TestInstance.Lifecycle.*PER\_CLASS*)  
public class TestClass {  
 private String message = "default";  
 TestClass(){  
 System.*out*.println("Constructorul va fi executat o singua data!");  
 }  
 @Test  
 public void test1(){  
 message = "test1";  
 System.*out*.println(message);  
 }  
 @Test  
 public void test2(){  
 System.*out*.println(message);  
 }  
  
}

* Asta e default:

@TestInstance(TestInstance.Lifecycle.*PER\_METHOD*)

**Proprietatile la teste**

* Isolate – testele nu sunt legate intre ele si returneaza acelasi rezultat indiferent de ordinea executiei lor
* Composable – indiferent de numarul de teste rulate, ele ne vor oferi aceleasi rezultate
* Rapide – testele trebuie sa ruleze repede
* Inspiring – un test care trece trebuie sa ofere incredere
* Writable – costul crearii testului trebuie sa fie mai mic decat costul crearii codului
* Readable – testele trebuie sa fie usor de citit
* Behavioral – comportamentul testelor depinde de comportamentul codului
* Structure-insensitive – modificarea logicii codului nu modifica rezultatele testului(cam ca behavioral)
* Automated – ruleaza singure
* Specific – daca un test cade, motivul trebuie sa fie clar
* Deterministic – daca codul nu se modifica, nici testul nu are cum cadea
* Predictive – daca toate testele trec, codul e bun pentru vanzare

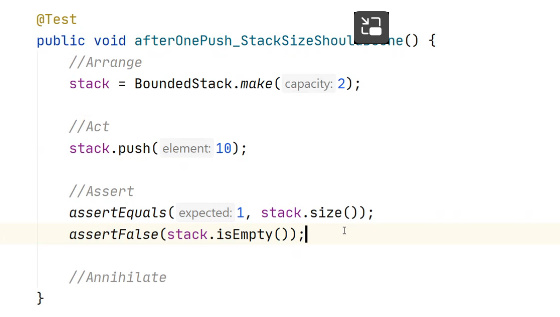
**Unit Test**

* Un test nu este unit test daca:
* comunica cu baza de date
* comunica in retea
* atinge fisierele de sistem
* nu poate rula in acelasi timp cu alte unit tests
* trebuie facute modificari in environment(ca config file)
* Aceste caracteristice sunt pentru **Integration Test**

**Etapele unui test**

1. Arrange – cream obiectul de testat
2. Act – executam metodele de testat
3. Assert – asserturile
4. Anihilate – stergem obiectele testate. Garbage Collection o va face

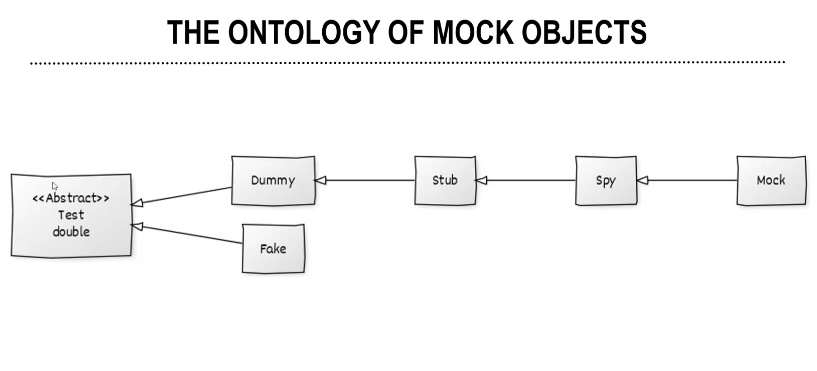
* E recomandat sa lasam un rand liber intre fiecare etapa in test cod.



* Fiecare test trebuie sa testeze o singura chestie doar

**Test Double**

* Le folosim pentru a mima comportamentul vreunei dependente sau resurse necesare
* Gen sa nu comunicam chiar cu o baza de date reale, noi setam rezultate fictive
* Avem urmatoarele implementari pentru test double:



Atentie ca fiecare este un caz particular de altul, gen stub este un fel de dummy, spy este un fel de stub si dummy si tot asa.

**Dummy**

* Dummy – test double(obiect) sau mai bine zis clasa metodele caruia nu fac absolut nimic.
* De ex, sa zicem ca avem clasele Client, Bank si Report.Clasa Bank implementeaza interfata BankInterface. Clasa Report e cea care creaza rapoarte pentru clienti in baza de date, sau le sterge. Clasa Bank e cea care creaza rapoartele, dar clasa Report pur si simplu se asigura ca ele se creaza corect. Deci, asa arata clasa Report:
* public class Report {

public void generateReport(Client client,Bank bank){  
 if(client == null)  
 throw new NullPointerException();  
 bank.generateReportInDataBaseForClient(client);  
  
 }  
 public void removeReport(Client client,Bank bank){  
 if(client == null)  
 throw new NullPointerException();  
 bank.removeReportFromDataBaseForClient(client);  
 }  
  
}

* Si clasa Bank arata asa:
* public class Bank implements BankInterface{  
   private DataSource dataSource;  
   private Action action;  
    
   public Bank(DataSource dataSource, Action action) {  
   this.dataSource = dataSource;  
   this.action = action;  
   }  
    
   public void generateReportInDataBaseForClient(Client client){  
    
   }  
   public void removeReportFromDataBaseForClient(Client clint){}  
    
    
  }
* Acum, sa zicem ca vrem sa testam clasa Report. Vrem sa testam daca se arunca exceptie care trebuie cand oferim un Client null, deci propriu zis nici nu ne trebuie un obiect de tip Bank.Dar, avem problema, caci nu vom putea testa fara a avea un obiect de tip Bank, dar crearea lui implica inca si o baza de date,si un Action,si deci alte beanuri si constructor default nu avem.Am putea da in locul lui un null, dar poate if ar fi asa if(client == null || bank==null} si e problema
* Dummy va insemna sa cream o nou clasa care implementeaza BankInterface dar ale carei metode nu fac absolut nimic, adica cam asa va fi testul:

public class TestClass {  
  
@Test  
public void testIfExceptionIsThrownWhenClientIsNull(){  
 Client client = null;  
 BankInterface bank = new BankDummy();  
 Report report = new Report();  
  
 Assertions.*assertThrows*(NullPointerException.class,()->report.generateReport(client,bank));  
}

* public static class BankDummy implements BankInterface{  
   @Override  
   public void generateReportInDataBaseForClient(Client client) {  
    
   }  
    
   @Override  
   public void removeReportFromDataBaseForClient(Client clint) {  
    
   }  
   }  
    
    
  }
* Daca metodele ar trebuie sa returneze ceva, Dummy returneaza datele default pentru datele primitive, adica 0 pentru int etc. si null pentru obiecte

**Stub**

* Un Stub este un dummy(clasa),metodele sale nu fac nimic, dar daca metodele sale trebuie sa returneze ceva, ele vor returna ceva valori speciale, nu default pentru date primitive sau null
* Sa zicem ca acum am modifica BankInterface sa fie asa:
* public interface BankInterface {  
   boolean generateReportInDataBaseForClient(Client client, LocalDate localDate);  
   boolean removeReportFromDataBaseForClient(Client clint,LocalDate localDate);  
    
  }

Daca data trimisa va fi 1, vom face un report pe toata luna si vom returna true, daca nu e 1, vom face un report pe ziua aceea doar si vom returna false

* clasa Bank e asa deja:
* public class Bank implements BankInterface{  
   private DataSource dataSource;  
   private Action action;  
    
   public Bank(DataSource dataSource, Action action) {  
   this.dataSource = dataSource;  
   this.action = action;  
   }  
    
   public boolean generateReportInDataBaseForClient(Client client, LocalDate localDate){  
   return localDate.getDayOfWeek().getValue() == 1;  
   }  
   public boolean removeReportFromDataBaseForClient(Client clint,LocalDate localDate){  
   return localDate.getDayOfWeek().getValue() == 1;  
   }  
    
    
  }

Iar clasa report e asa:

public class Report {  
 private final LocalDate localDate;  
  
 public Report(LocalDate localDate) {  
 this.localDate = localDate;  
 }  
  
 public String generateReport(Client client, BankInterface bank){  
 if(client == null)  
 throw new NullPointerException();  
 if(bank.generateReportInDataBaseForClient(client, localDate))  
 return "Weekly report";  
 return "Simple Report";  
  
 }  
 public void removeReport(Client client,Bank bank){  
 if(client == null)  
 throw new NullPointerException();  
 bank.removeReportFromDataBaseForClient(client,localDate);  
 }  
  
}

Deci, daca avem un report pe o luna, va returna “Weekly Report”

* Problema e ca clasa Bank necesita dependente, si accesarea lor nu e buna in Unit Test. Iata de ce, cream o noua clasa ce implementeaza BankInterface, dar Stub va returna ceva concret, nu default:
* public class TestClass {  
    
   @Test  
   public void testIfExceptionIsThrownWhenClientIsNull(){  
   Client client = new Client("Test","Test");  
   BankInterface bank = new BankStub();  
   Report report = new Report(LocalDate.*of*(2002,12,30));  
    
   Assertions.*assertEquals*("Weekly report",report.generateReport(client,bank));  
   }  
   public static class BankStub implements BankInterface {  
   @Override  
   public boolean generateReportInDataBaseForClient(Client client, LocalDate localDate) {  
   return localDate.getDayOfWeek().getValue() == 1;  
   }  
    
   @Override  
   public boolean removeReportFromDataBaseForClient(Client clint,LocalDate localDate) {  
   return true;  
   }  
   }  
    
    
  }

Pur si simplu e aceeasi metoda ca in clasa Bank, dar nu mai e nevoie de acele dependente, care si asa nu ne trebuie, dar fara de care nu am putea crea clasa Bank.

**Spy**

* Un Spy este un Stub dar care memoreaza ce metode s-au apelat, de cate ori, cu ce parametri etc.
* Sa zicem ca vrem sa numaram de cate ori se executa metoda

generateReportInDataBaseForClient

de catre Bank class. Tot ce ar trebuie sa facem e sa modificam clasa BankStub ca ea sa aiba o un membru static ce calculeaza asta:

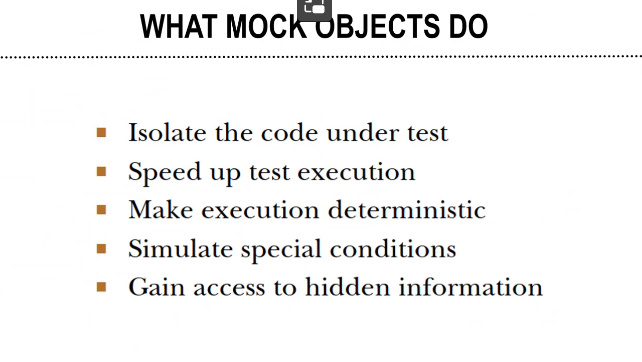
public static class BankSpy implements BankInterface {  
 private static int *count* = 0;  
 @Override  
 public boolean generateReportInDataBaseForClient(Client client, LocalDate localDate) {  
 *count*++;  
 return localDate.getDayOfWeek().getValue() == 1;  
 }  
  
 @Override  
 public boolean removeReportFromDataBaseForClient(Client clint,LocalDate localDate) {  
 return true;  
 }  
 public int getCount(){  
 return *count*;  
 }  
}

Si testul va fi asa:

@Test  
public void testIfExceptionIsThrownWhenClientIsNull(){  
 Client client = new Client("Test","Test");  
 BankSpy bank = new BankSpy();  
 Report report = new Report(LocalDate.*of*(2002,12,30));  
  
 report.generateReport(client,bank);  
 BankSpy bankSpy = bank;  
  
 Assertions.*assertEquals*(1, bank.getCount());  
}

**Mock**

* **Mock** este un Spy, dar el are asteptari, adica stie ce trebuie sa se intample. Cand folosim un spy, testul ia toata informatia din Spy ca sa decida daca a trece sau nu(testul). Cand folosim un mock, testul il intreaba pe mock daca totul a fost cu succes si mockul decide daca a fost sau nu
* **Spre deosebire de stub, el chiar are cod in metode, nu pur si simplu returneaza ceva, si permite setarea de expectation pentru a decide daca testul trece sau nu, dar stub e limitat, si metodele practic sunt goale.**
* public class TestClass {  
    
   @Test  
   public void testIfExceptionIsThrownWhenClientIsNull(){  
   Client client = new Client("Test","Test");  
   BankMock bank = new BankMock();  
   Report report = new Report(LocalDate.*of*(2002,12,30));  
    
   report.generateReport(client,bank);  
    
   bank.expected(1);  
   bank.assertIsSatisfied();  
   }  
   public static class BankMock implements BankInterface {  
   private static int *count* = 0;  
    
   private int expected;  
   public void expected(int value){  
   expected = value;  
   }  
   public void assertIsSatisfied(){  
   Assertions.*assertEquals*(expected,*count*);  
   }  
   @Override  
   public boolean generateReportInDataBaseForClient(Client client, LocalDate localDate) {  
   *count*++;  
   return localDate.getDayOfWeek().getValue() == 1;  
   }  
    
   @Override  
   public boolean removeReportFromDataBaseForClient(Client clint,LocalDate localDate) {  
   return true;  
   }  
   public int getCount(){  
   return *count*;  
   }  
   }  
  }



**Fake**

* este o implementare partiala la o interfata care are neajunsuri pentru a fi folosit in productie, dar ne ajuta in teste
* Fie DAO:



Daca am vrea se obtinem date din baza de date, ar trebui sa cream data source, session factory etc. si asta e cam anevoios.

* Asa ar fi creat un fake pentru ReportRepository:



**More about Unit Tests**

* Clasele ce nu au dependente sunt perfecte pentru unit tests
* Clasele cu colaboratori/dependentele se testeaza cu test double
* Clasele ce comunica de ex cu baza de date chiar, nu pot fi testate cu Unit Tests, ci cu Integration Test si trebuie sa comunice cu baza de date sau In memory DB

**Mockito**

* Mockito ne ajuta sa cream Test Double fara hard code
* pentru a crea un **Dummy**, tot ce trebuie sa facem este:

BankInterface bankInterface = Mockito.*mock*(BankInterface.class);

Tot ce a facut Mockito este sa creeze la runtime o clasa ce implementeaza interfata data si metodele ei nu fac nimic.

Anotatia @Mock face exact asta.

* **Mockito.verify(MockObject).MetodaNoastra()** – daca metoda setata nu e executata niciodata, testul da fail
* Acum apare intrebarea cum sa oferim un corp unei metode. De ex, fie clasa:

public class Computer {  
  
 public int doubleNumbers(int nr1,int nr2){  
 return nr1\*nr2;  
 }  
}

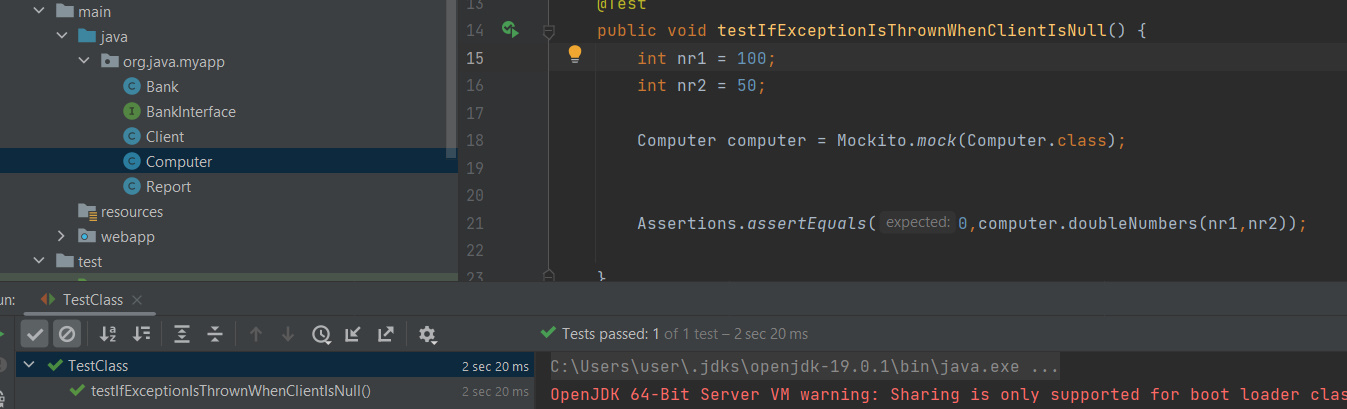
Un mock va returna mereu 0 ca valoare default, dar noi vrem ca mockul sa aiba o metoda creata de noi.Pentru asta folosim

**doAnswer(answer -> { }).when(object).metoda()**

answer este un parametru de tip Answer, ce are metode, ca **answer**.**getArgument(pozitie, tip.class)** care ne returneaza primul argument

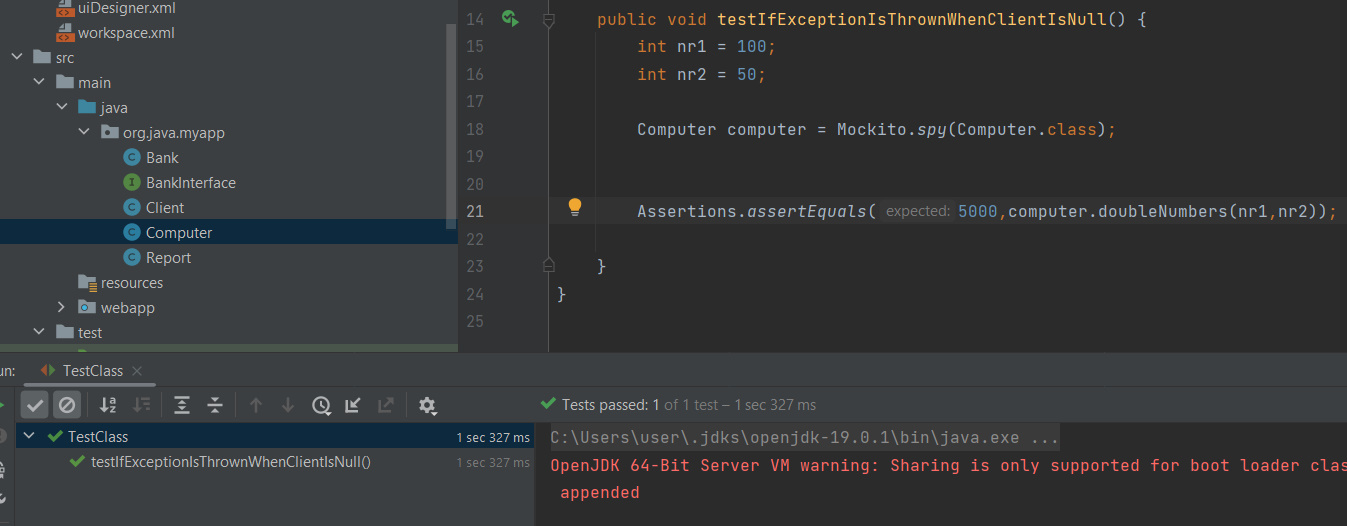
@Test  
public void testIfExceptionIsThrownWhenClientIsNull() {  
 int nr1 = 100;  
 int nr2 = 50;  
  
 Computer computer = Mockito.*mock*(Computer.class);  
 Mockito.*doAnswer*(answer -> {  
 int param1 = answer.getArgument(0);  
 int param2 = answer.getArgument(1,Integer.class);  
 System.*out*.println("Metoda doubleNumber e creata de Mockito");  
  
 return param1\*param2;  
 }).when(computer).doubleNumbers(nr1,nr2);  
 Assertions.*assertEquals*(5000,computer.doubleNumbers(nr1,nr2));  
  
}

Iata ce va face mockito fara instructiunile de mai sus:



deci a returnat automat 0

* Putem crea si un spy asa:



Dar vedem ca nu a returnat 0, ci rezultatul corect.

La spy, mockito putin modifica regulilele generale. Ea tot va crea o clasa noua si va crea metode ce nu fac nimic, dar cand o metoda va returna ceva, ea meru va returna ceea ce se executa in clasa de baza, gen va fi asa:

**int doubleNumber(int a,int b)**

**return super.doubleNumber(a,b);**

**}**

**Method Parametrized test**

public class TestClass {  
  
 @ParameterizedTest()  
 @MethodSource(value = "source")  
 public void test(int nr1, int nr2, int res) {  
 Computer computer = Mockito.*spy*(Computer.class);  
 Assertions.*assertEquals*(res,computer.doubleNumbers(nr1,nr2));  
  
 }  
 public static Stream<Arguments> source(){  
 return Stream.*of*(  
 Arguments.*of*(0,1,0),  
 Arguments.*of*(5,6,30),  
 Arguments.*of*(10,8,80)  
 );  
 }  
}

**More**

* Mockito.mock(clasa.class) – creaza un dummy
* Daca mai folosim when(...).thenReturn(...) – cream un stub, caci setam returnul
* Mockito.spy(clasa.class) – spy
* Daca folosim doAnswer(....) – cream mock, caci dam corpuri concrete metodelor