# Academia de Studii Economice București

# Facultatea de Cibernetică, Statistică și Informatică Economică

Expert Independent

Atelier de grafica (Graphics Workshop)



Profesor: Student:

Asist. univ. dr. ALIN ZAMFIROIU EREMIA FLORINA-ROXANA

GRUPA 1064, SERIA A

**Cuprins**

**1. Prezentarea pe scurt a aplicației3**

**2. Definirea și justificarea pattern-urilor implementate** 2**1**

2.1.Pattern-uri creaționale 21

2.1.1. Singleton 21

2.1.2. Simple Factory 22

2.1.3. Builder22

2.2. Pattern-uri comportamentale23

2.3.1. Strategy 23

2.3.2. Chain of Responsibility 24

2.3.3. Observer 24

2.3. Pattern-uri structurale22

2.2.1. Adapter 21

2.2.2. Decorator22

2.2.3. Facade22

**3.** **Definirea și detalierea Test Case-urilor 30**

3.1.Test\_Client30

3.2.Test\_Administrator30

3.3.Test\_Designer31

3.4.Test\_Livrare32

3.5.Test\_Tiparire32

3.6.Test\_Comanda30

3.7.Test\_Scenariu30

3.8.Test\_Afis31

3.9.Test\_Poster32

3.10.Test\_Pliant32

**4.** **Definirea și descrierea Test Suite-urilor33**

4.1.Teste\_Utilizatori\_Platforma30

4.2.Teste\_Operatii\_Platforma30

4.3.Test\_Materiale\_Publicitare31

4.4.Teste\_Platforma31

# Prezentarea pe scurt a aplicației

Platforma online “Expert Independent” are ca obiectiv simplificarea și automatizarea unui proces rezultat din necesitatea obținerii serviciului denumit: achiziționarea de materiale publicitare. Această nevoie se regăsește în principal la persoanele juridice dar nu numai.

Misiunea platformei, pe lângă obiectivul asumat de a automatiza întregul proces de achiziție de materiale publicitare (include și achizitia de design și grafică), este de a oferi prin competiție deschisă servicii de design și grafică tot mai bune, dar și de a crea o comunitate de designeri competitivi. Urmăresc pe termen lung, ca efect al utilizării platformei, să crească accesibilitatea oricărei companii către grafică și design de calitate la cateva clickuri distanță.

Scenariul platformei este următorul: un antreprenor dorește crearea unui flyer sau afiș. Creează cont sau se loghează în platformă, introduce: bugetul și cerințele, fișiere ( precum logo, text, poze) și lansează un concurs cu deadline la care se vor înscrie designerii care au cont în platformă. Bugetul alocat proiectului va fi încasat inițial și aprobat de către platformă Aprobarea este necesară pentru a evita fraude. Designerii nu vor vedea bugetul. Aceștia vor realiza fiecare câte un produs pe care îl vor uploada în platformă alături de un preț considerat de ei necesar pentru implementarea cerinței.

Clientul are la dispoziție 10 zille după expirarea deadline-ului pentru a desemna un câștigător. În cazul în care nu va desemna un câștigător, bugetul încărcat în platformă va fi distribuit folosind următoarea distribuție: 25% rămân în contul clientului, 25% în contul platformei, 50% se împarte între participanți.

În cazul desemnării unui câștigător, bugetul se suplimentează în cazul în care este mai mic decat oferta aleasă, clientul va plăti diferența online și va desemna în continuare următorii 4 câștigători ai locurilor: 2, 3, 4 și 5. Distribuirea bugetului se face folosind un algoritm menit să echilibreze și să motiveze participanții și să ofere puncte de feedback.

Algoritmul de distribuire al sumei proiectului:

a) Desemnarea castigatorilor:

- câștigătorul (furnizorul proiectului) primește 5 stele;

- clientul poate desemna și alți câștigători de stele oferind un număr de stele între 1 și 5 din maximul de 20 pe care le are la dispoziție.

- clientul poate desemna de asemenea pe cei care nu s-au încadrat în cerințele sale fiind considerați

incompetenți.

b) Distribuirea sumei:

- 10% pentru platformă;

- 70% pentru câștigător;

- 20% se împart la numărul de stele oferite (din maximum 20) celorlalți câștigători. În cazul în care nu au fost desemnați și alți câștigători acesti 20% se împart în mod egal între ceilalți participanți care nu au fost desemnați ca fiind incompetenți.

Mai departe, după obținerea graficii, clientul va putea downloada documentul pregătit pentru printare, va putea alege una din tipografiile afiliate platformei căreia îi va da automat comanda de print cu plată online și va putea alege și firma de curierat pentru livrare, tot cu plată online prin intermediul platformei.

# Definirea și justificarea pattern-urilor implementate

# 2.1.Pattern-uri creaționale

# 2.1.1.Singleton

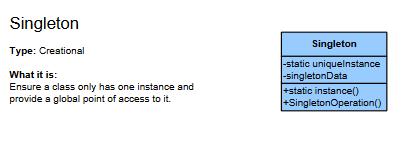
**Definiție**

Singleton-ul denumit și *obiect unicat* crează premizele ca o anumită clasă să fie instanțiată doar o singură dată, permițându-se un acces global la instanța respectivă.

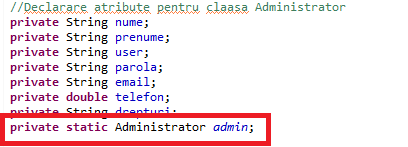
**Context**

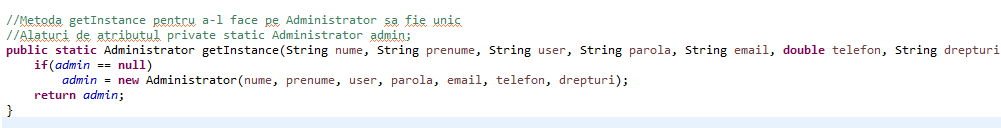
Există situații când un anumit obiect trebuie să existe doar într-un singur exemplar. Aplicând în contextul aplicației mele, administratorul platformei este unic întrucât nu avem nevoie de mai multe persoane să administreze această platformă.

**Implementare**



Ce este specific acetui design pattern**(clasa Administrator)** este faptul că avem un atribut static de tipul clasei a cărei intanțiere este unică și de asemenea implementarea metodei getInstance.





# 2.1.2. Simple Factory

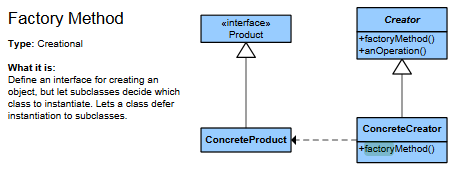
**Definiție**

Simple Factory denumit și *fabrică* este design pattern-ul ce crează mai multe obiecte ce conțin atribute și proprietăți asemănătoare, pe scurt creează obiecte dintr-o familie.

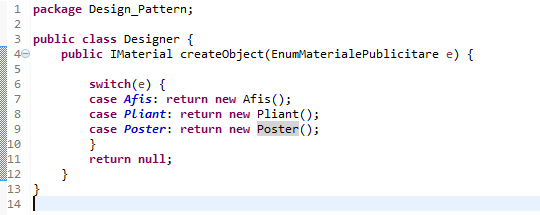
**Context**

Aplicând în contextul aplicației mele, clientul este cel care comandă realizarea schiței unui material publicitar, iar designerul este cel care implementează soluția cerută de către client. Clientul poate cere realizarea unui afiș publicitar, un flyer, unui pliant, un poster, etc.

**Implementare**



Ce este specific acetui design pattern**(clasa Designer)** este faptul că aceeași clasă poate creea mai multe tipuri de material publicitare enumerate mai sus.



Am creat clase pentru fiecare material publicitar(Afiș, Pliant, Poster) care implementează interfața IMaterial ce conține metoda de notificare a momentului când afișul respectiv este realizat de către designerul nostru. De asemenea am creat și un enum pentru a enumera cele trei categorii de materiale publicitare enunțate mai sus.

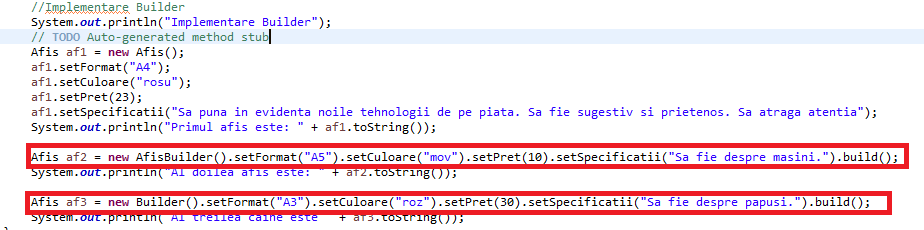
# Builder

**Definiție**

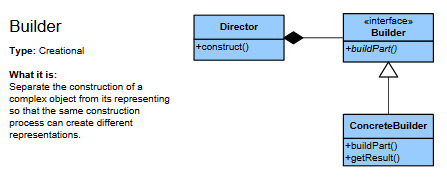
Builder-ul denumit și *constructor* este șablonul care descrie crearea unor obiecte în regim pas cu pas, în condițiile în care procesul de creare este independent de structura internă a obiectelor. Pes curt, builder-ul creează obiecte setând anumite atribute.

**Context**

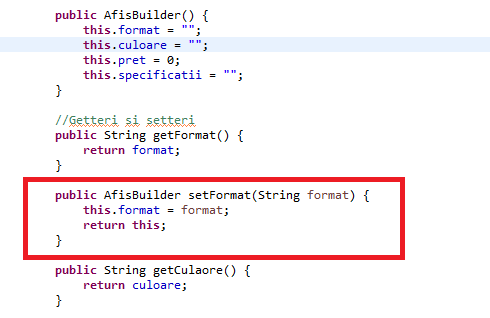
Aplicând în contextul aplicației mele, am ales să implementez builder-ul pentru unul din materialele publicitare și anume pentru **clasa Afiș.** Am creat trei afișe prin schimbarea atributelor obiectului afiș.

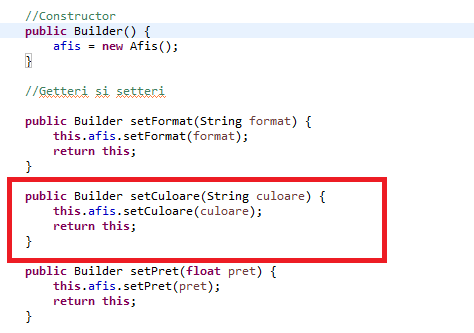


**Implementare**



Ce este specific în implementarea acestui design pattern este reprezentat de implementarea metodei set în **clasel**e **AfisBuilder** și **Builder.**

**Clasa AfișBuilder**

**Clasa Builder**

# 2.2.Pattern-uri comportamentale

# 2.2.1.Strategy

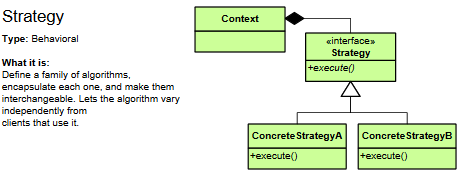
**Definiție**

Strategy-ul denumit și *behavioral(=comportamental)* este cel care modifică comportamentul obiectului. Alegerea la run time a algoritmului/funcției care să fie utilizată pentru procesarea unor date. Algoritmul se poate alege pe baza unor condiții descrise la execuție în funcție de contextul datelor de intrare.

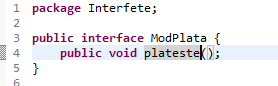
**Context**

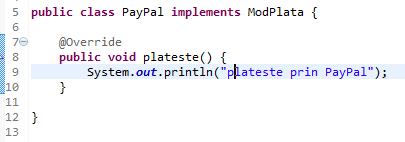
Aplicând în contextul aplicației mele, am ales să implementez strategy-ul pentru a evidenția metodele de plată prin care un client își poate achita comanda. Acesta poate plăti online prin Visa, MasterCard sau Paypal.

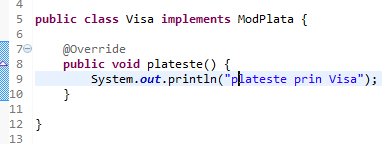
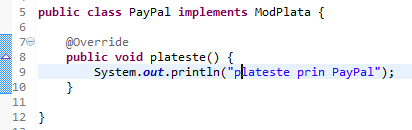
**Implementare**

****

Avem o interfață ce implementează o metodă ce este definită înfiecare din clasele ce o implementează. Metoda respectivă are comportament diferit în funcție de apelatorul metodei.







# 2.2.2.Chain of Responsibility

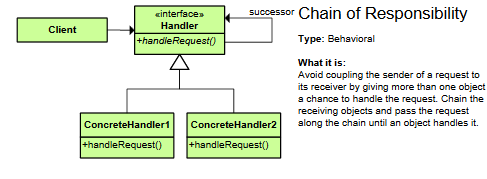
**Definiție**

Tratarea unui eveniment sau a unui obiect se face diferit în funcție de starea acestuia. Gestiunea tuturor cazurilor ar implica o structură complexă care să verifice toate cazurile particulare. Există legături de dependență între cazurile de utilizare: execuția unui caz poate implica ignorarea celorlalte sau tratarea următorului caz.

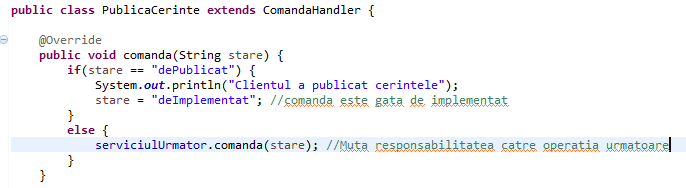
**Context**

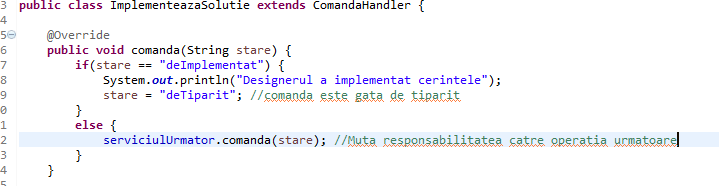
Aplicând în contextul aplicației mele, am ales să implementez chain of responsability pentru a evidenția activitățile ce au loc în interiorul platformei. Mai întâi are loc comanda clientului ce presupune obținerea celei mai bune soluții pentru problema în cauză. După preferințele clientului, acesta poate bemeficia de tipărirea materialelor publicitare, responsabilitatea (materialele publicitare) este preluată de tipografie ce le printează. Un alt serviciu suplimentar este livrarea materialelor la domiciliul clientului. Responsabilitatea este acum la curier care are misiunea de a livra materialele publicitare la destinație.

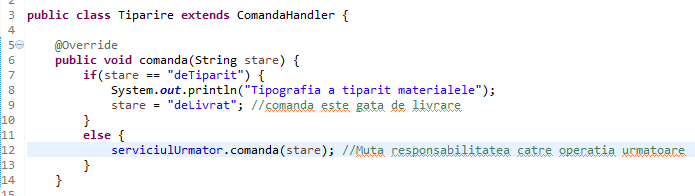
**Implementare**

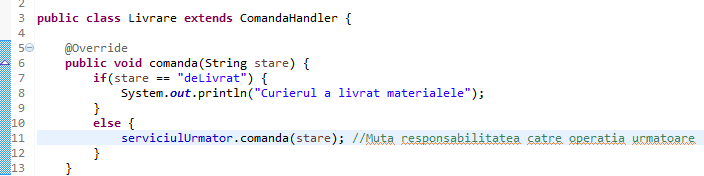


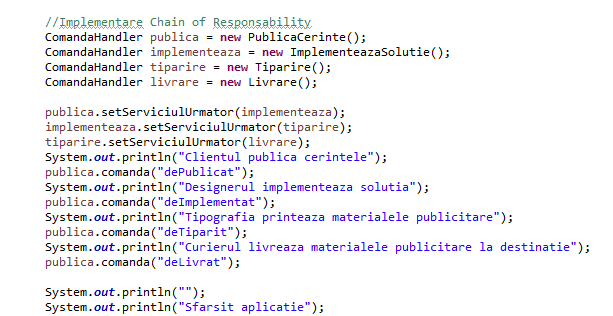
Handlerul acestui design pattern este reprezentat de starea unei comenzi( care poate fi – publicat dacă clientul și-a publicat cerintele in cadrul platformei (moment in care statusul stării trece din starea *dePublicat* devine *deImplementat*). În momentul în care starea este  *deImplementat*, designerul se apucă de implementarea cerințelor. La finalizarea muncii, acesta publică soluțța, iar statusul stării devine *deTipărit*. După aceea, dacă clientul dorește să i se și printeze soluția atunci admin-ul contactează tipgrafia spre tipărirea materialelog publicitare. Acesta este pomentul în care statusul stării devine *deLivrat*. Dacă clientul dorește și livrarea la domiciliu, admin-ul contactează și un curier.











Apelul în clasa **Main.**

# 2.3.3. Observer

**Definiție**

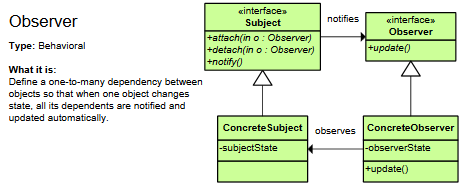
Observer-ul execută o acțiune când are loc un eveniment sau un observabil își schimbă starea. Acesta definește în fapt un hadler pentru diferite evenimente.

* Există componente care trebuie să fie notificate la producerea unui eveniment.
* Gestiunea evenimentelor la nivel de interfață.
* Componentele se abonează/înregistrează la acel eveniment – modificare de stare/acțiune.
* La producerea unui eveniment pot fi notificate mai multe componente.

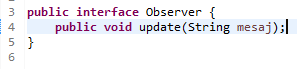
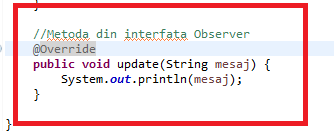
**Context**

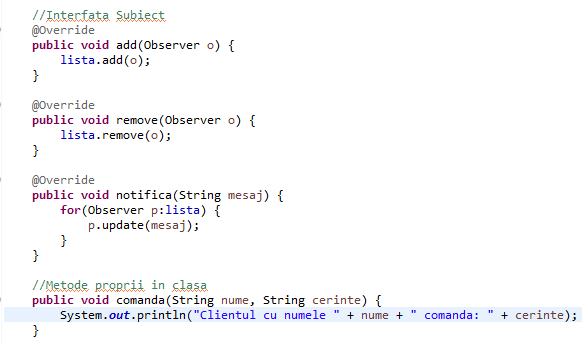
Aplicând în contextul aplicației mele, am ales să implementez observerul în două cazuri. În ambele cazuri observatorul este administratorul platformei. În primul caz subiectul/observabilul este clientul, iar în cel de al doilea caz subiectul este designerul. Admin-ul este cel care îl notifică pe subiect asupra unui eveniment – pe designer dacă clientul a postat cerințele, iar pe client atunci când soluția a fost implementată de către designer.

**Implementare**

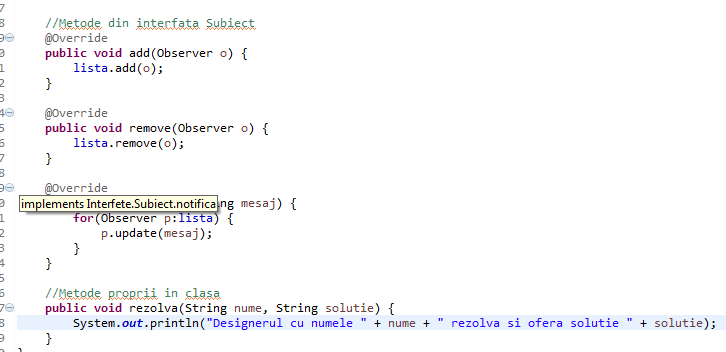


Avem un administrator care observă doi subiecți(client și designer).





**Clasa Client**

**Clasa DesignerFactory**

# 2.3.Pattern-uri structurale

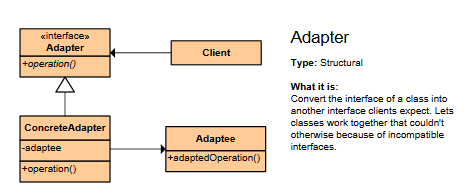
# 2.3.1.Adapter

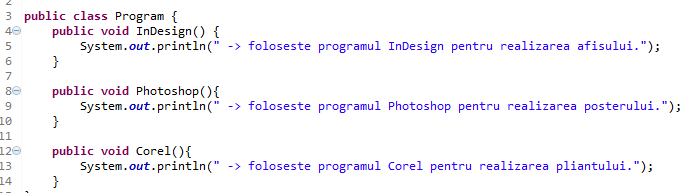
**Definiție**

Utilizarea împreună a unor clase ce nu au o interfață comună. Clasele nu se modifică însă se construiește o interfață ce permite utilizarea lor în alt context. Clasele sunt adaptate la un nou context. Apelurile către interfața clasei sunt mascate de interfața adaptorului. Pe scurt adapter-ul asigură o interfață diferită obiectului.

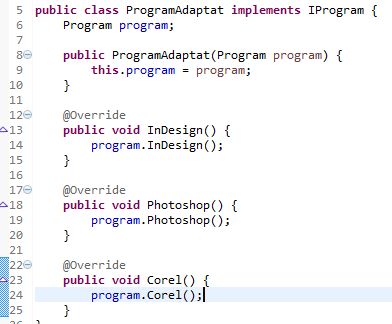
**Context**

Aplicând în contextul aplicației mele, am ales să implementez adapterul pentru a pune în evidență modalitatea de lucru a designerului. El utilizează programe diferite pentru fiecare material publicitar în parte. Spre exemplu pentru realizarea unui afiș va folosi programul InDesign, pentru poster va folosi programul Photoshop, iar pentru pliant va folosi Corel.

**Implementare**



Clasa **ProgramAdaptat** reprezintă handler pentru medotele din interfața IProgram, acestea fiind implementate în clasa **Program**. Clasa **ProgramAdaptat** are în componența sa un **Program**.



# 2.3.2.Decorator

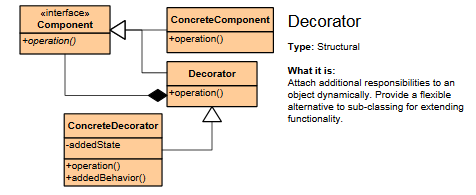
**Definiție**

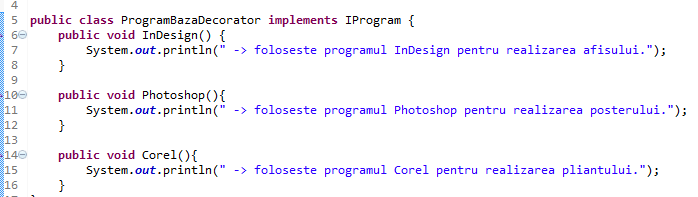
Decorator permite modificarea la run-time a funcționalității obiectului fără a-i schimba definiția. Obiectele sunt create în mod normal și apoi decorate la execuție. Pe scurt, decoratorul este cel care adaugă dinamic funcții noi la comportamentul obiectului. Obiectul poate să fie extins prin aplicarea mai multor decorator.

**Context**

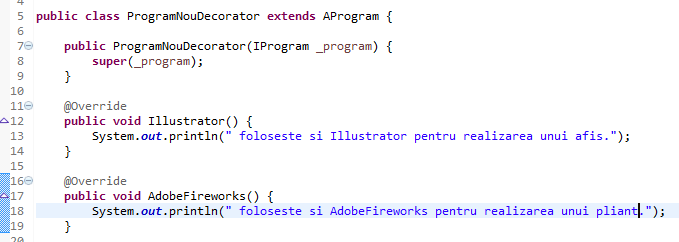
Aplicând în contextul aplicației mele, am ales să implementez decoratorul pentru a pune în evidență faptul că designer-ul dorește să fie mereu în pas cu ce apare nou pe piață în materie de design. Acesta învață să folosească toate programele ce sunt de noutate.

**Implementare**



Programele de bază pe care le folosește designer-ul sunt

Programele noi pe care învață să le folosească sunt



# 2.3.3. Façade

**Definiție**

Decorator simplifică execuția (apelarea) unui scenariu complex. Soluția conține o mulțime de clase, iar execuția unei funcții presupune apeluri multiple de metode aflate în aceste clase. Clasele nu se modifică însă se construiește un layer intermediar ce permite apelul/gestiunea

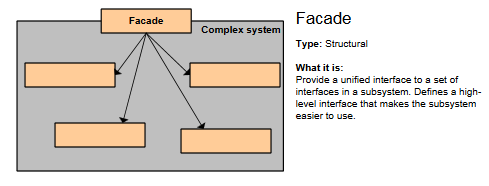
facilă a metodelor din mai multe interfețe. Utilă în situația în care framework-ul crește în

complexitate și nu este posibilă rescrierea lui pentru simplificare. Apelurile către multiplele interfețe sunt mascate de aceasta interfață comună.

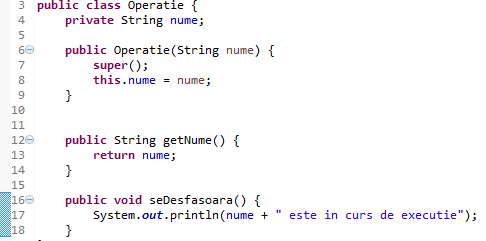
**Context**

Aplicând în contextul aplicației mele, am ales să implementez façade pentru a pune în evidență toate operațiile implicate în procesul de realizare a unui client. Scenariul este în felul următor – clientul comandă(postează cerințele materialului publicitar de care are nevoie), designer-ul implementează cerințele clientului. Apoi soluția urmează să fie tipărită și livrată la domiciliul clientului.

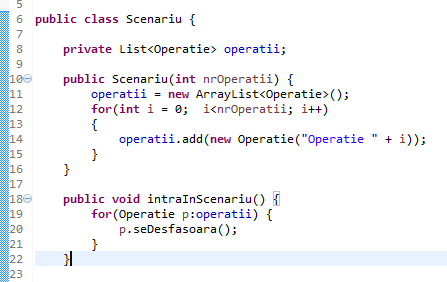
**Implementare**



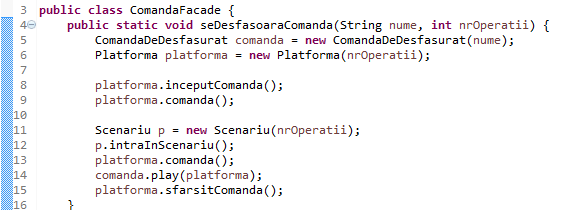
Avem o operație.



Avem un scenariu, format din toate operațiile realizării unei comenzi complete.



Façade-ul nostru urmează să implementeze operațiile din scenariu.

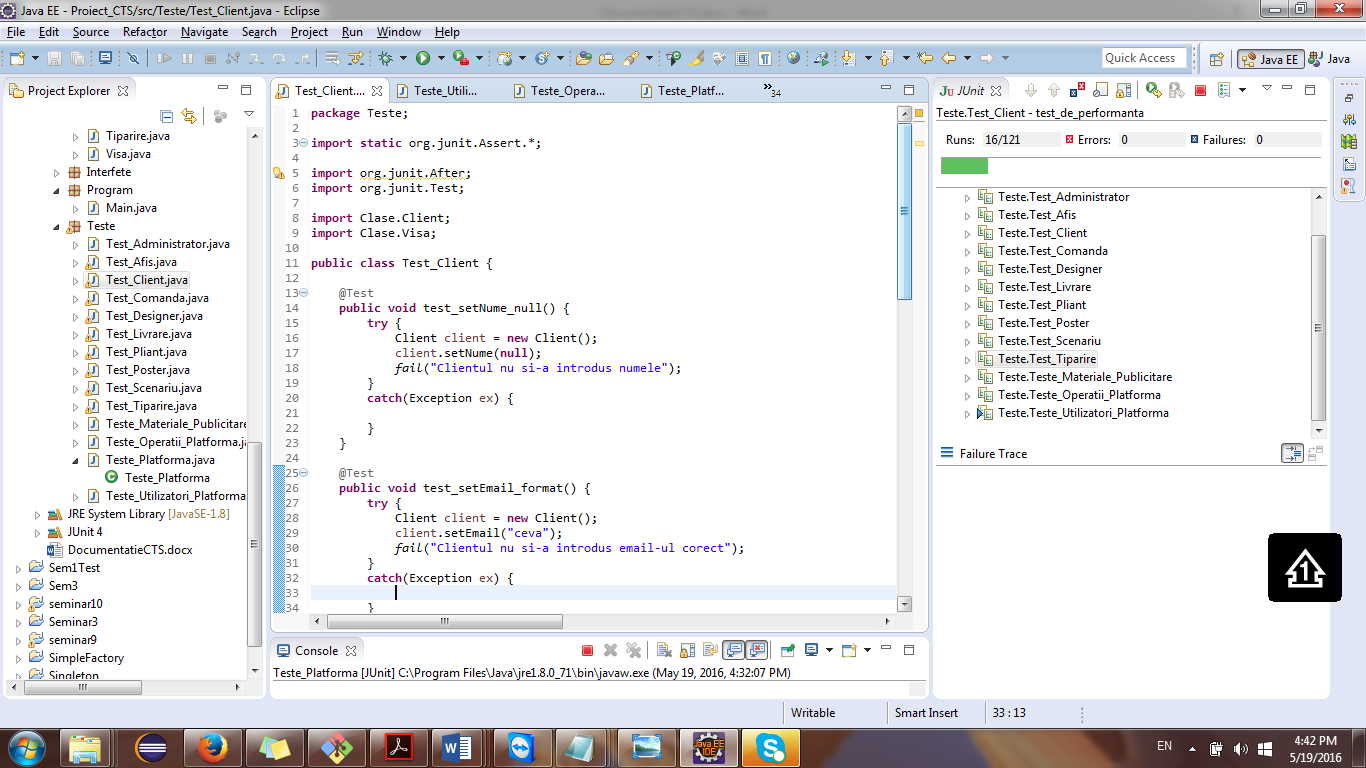
s

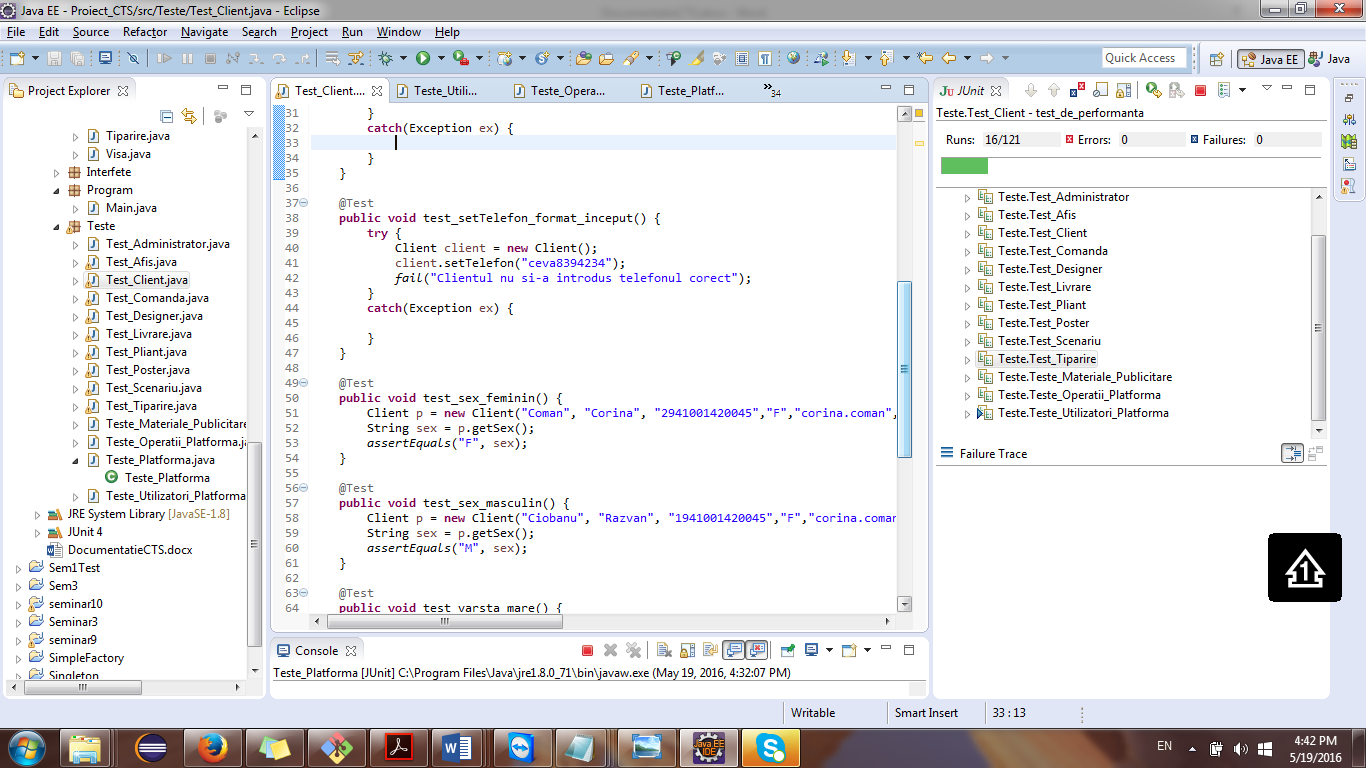
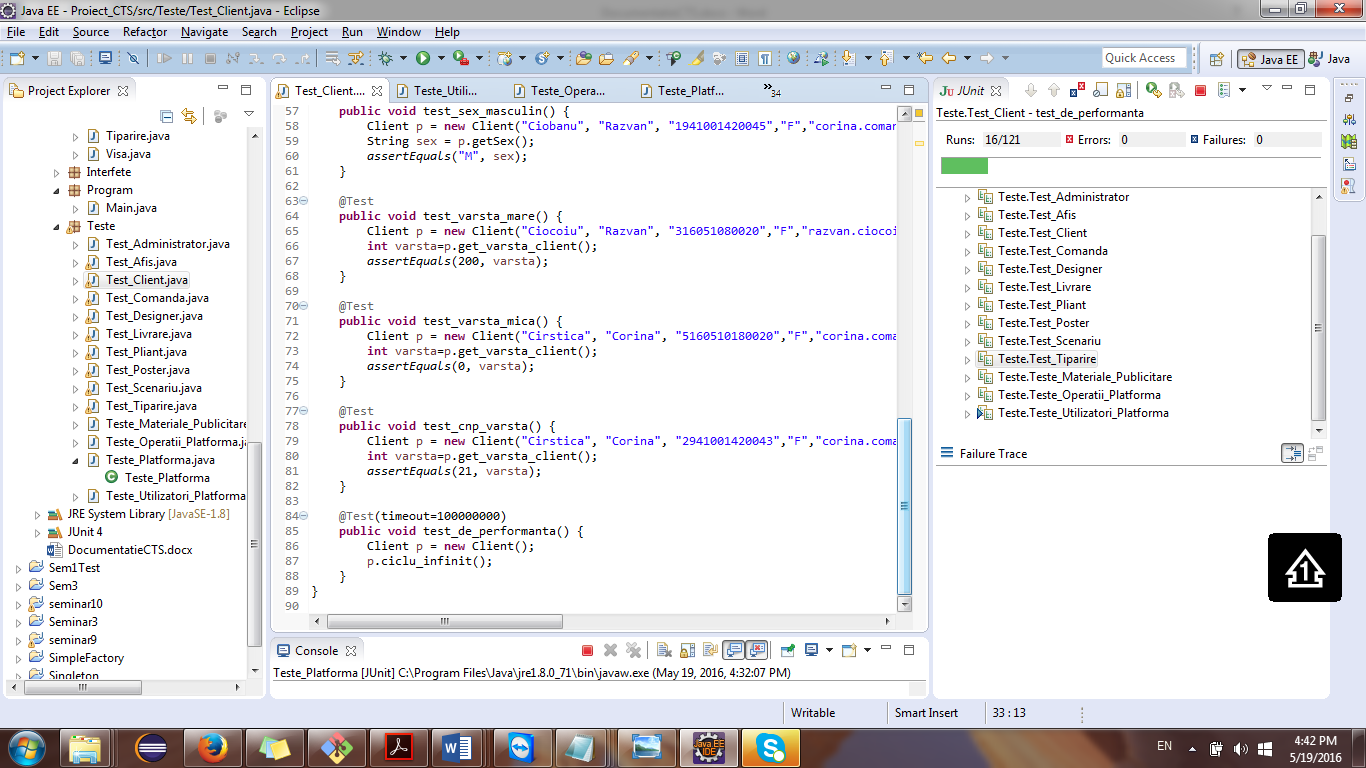
# Definirea și detalierea Test Case-urulor

Unit Testing reprezintă metoda simplă și rapidă de testare a codului sursă de către programatori. Are loc în faza de dezvoltare și este un instrument destinat programatorilor. Un unit test este o secvență de cod scrisă de un programator pentru a evalua o parte din codul sursă testat, de mici dimensiuni – clasă sau metodă. Un unit test evaluează modul de funcționare al unei metode într-un context bine definit. Un unit test este blocul de bază pentru abordarea Test-Driven Development.

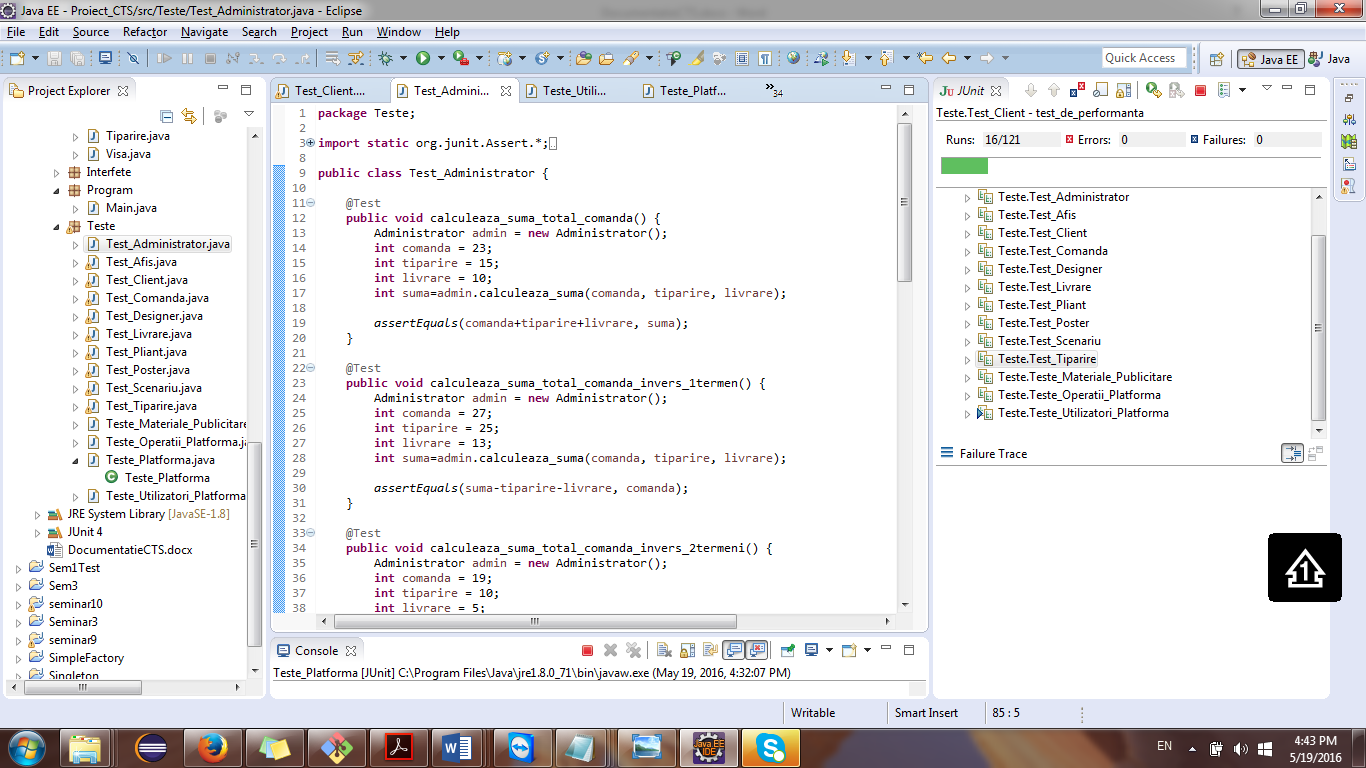
JUnit este un framework de clase ce permite scrierea și execuția de teste pentru diferite metode/clase din cod.

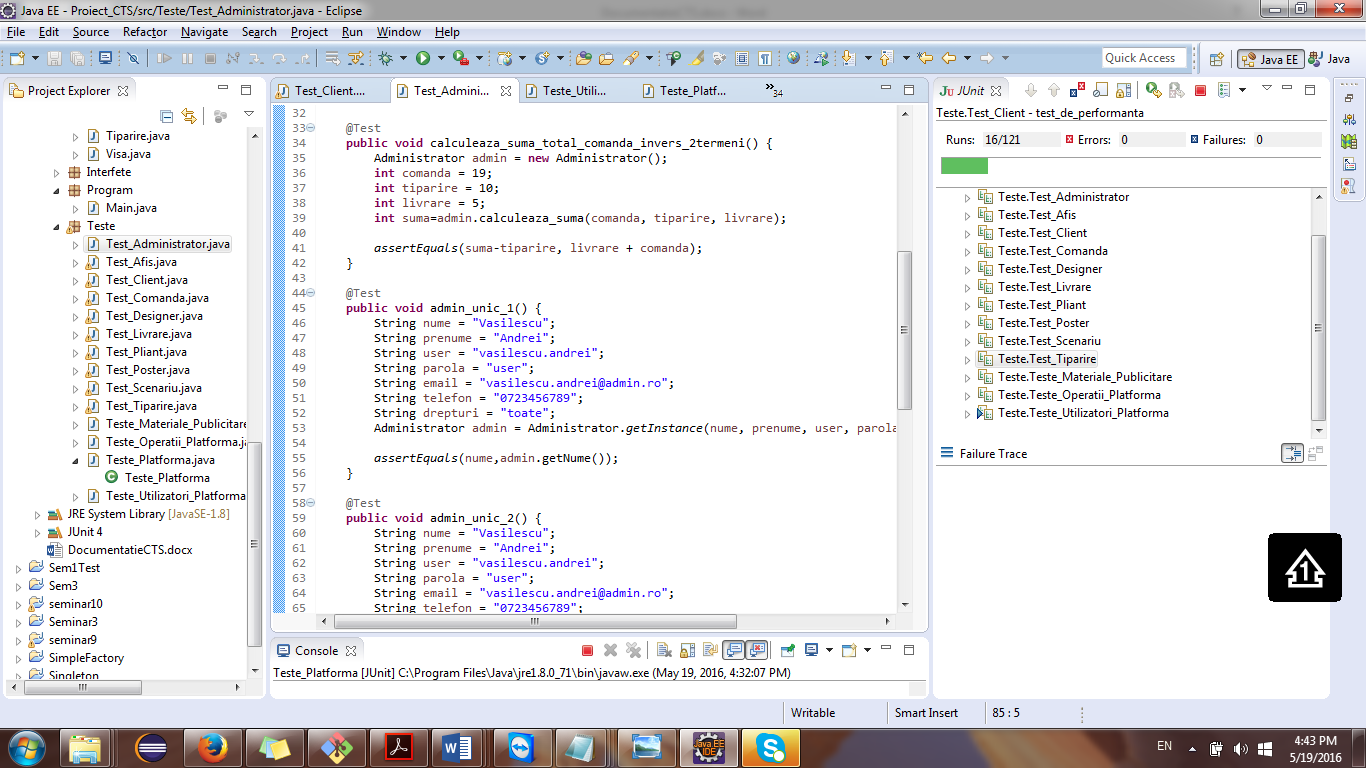
# Test\_Client



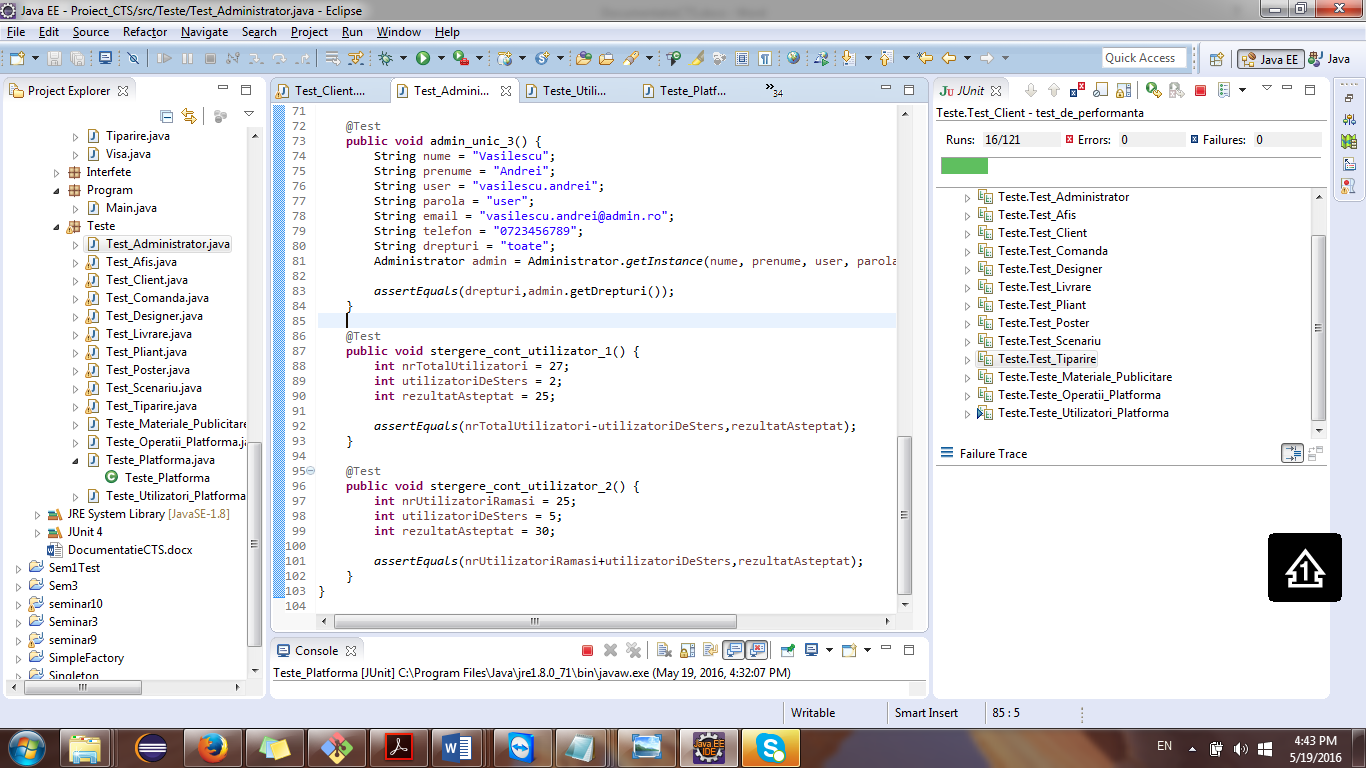


# Test\_Administrator

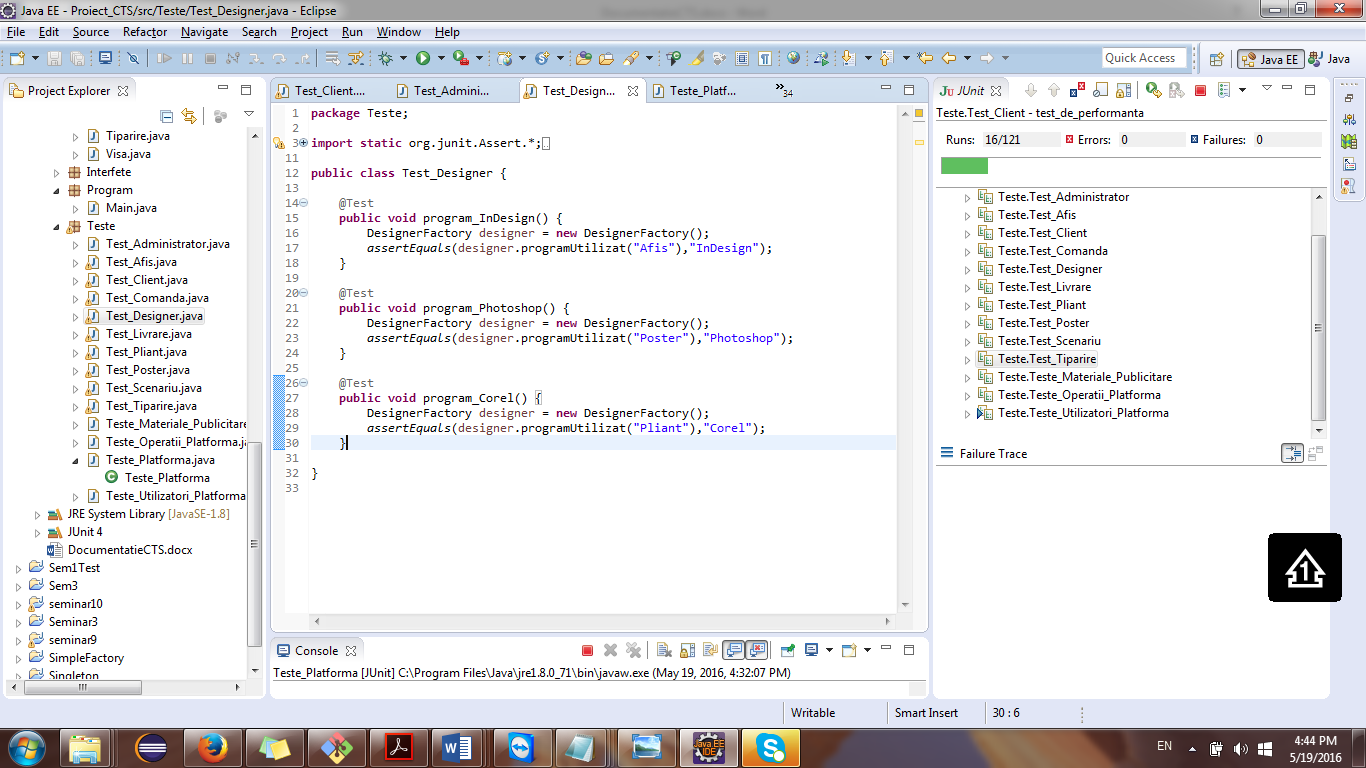




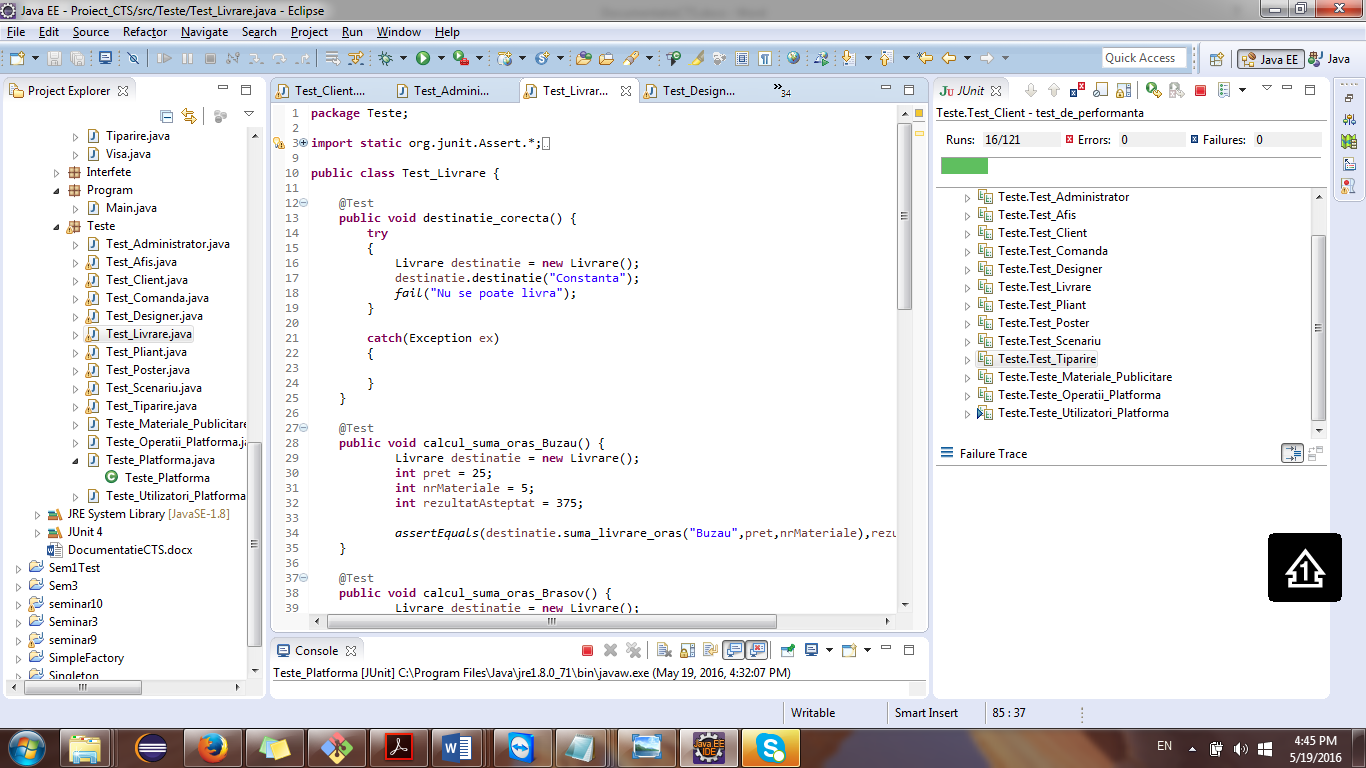


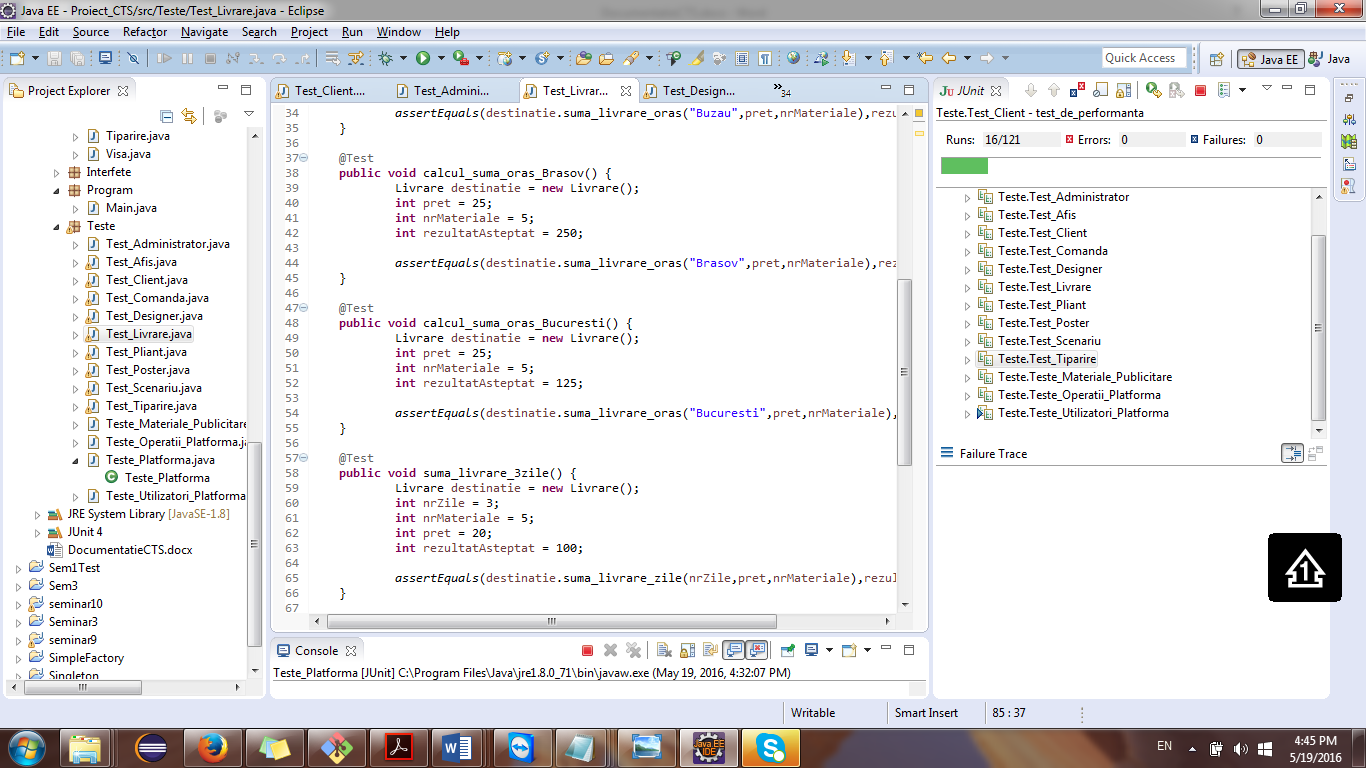


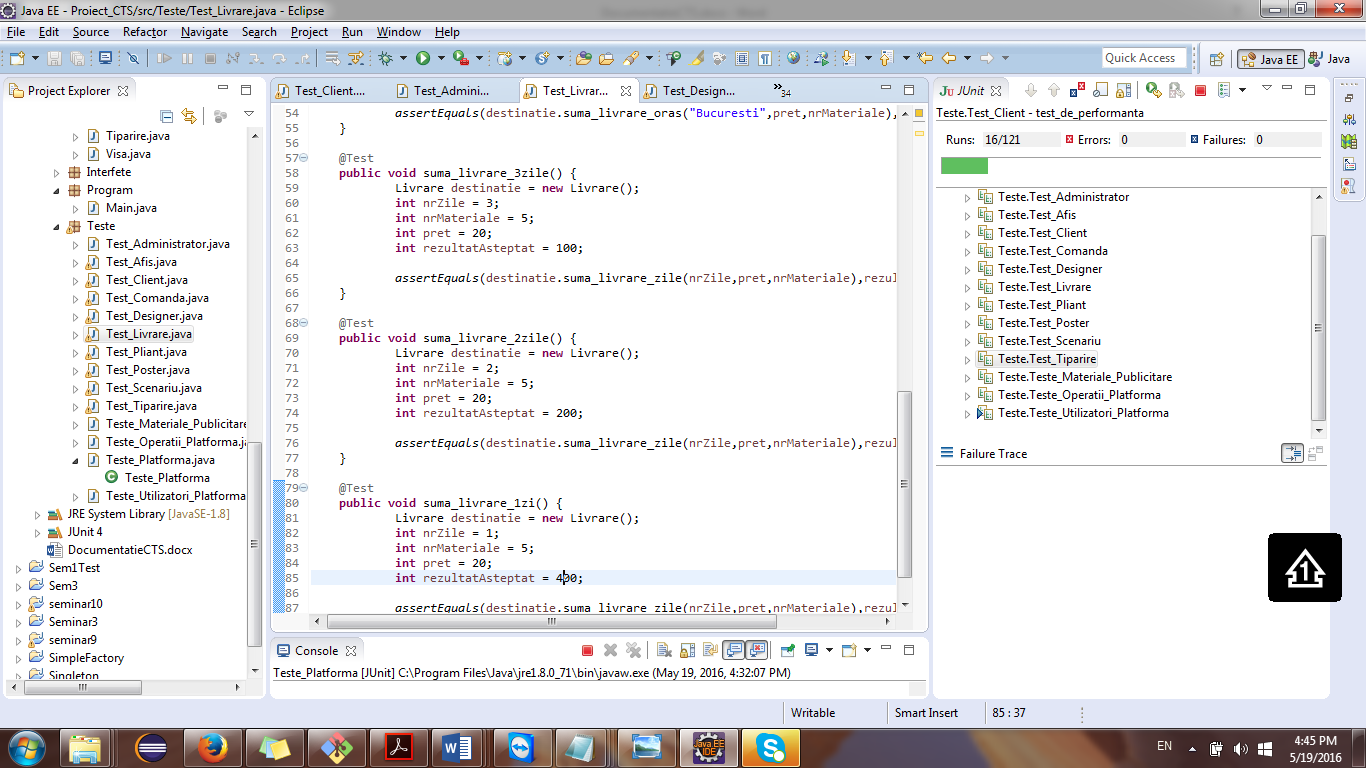
# Test\_Designer



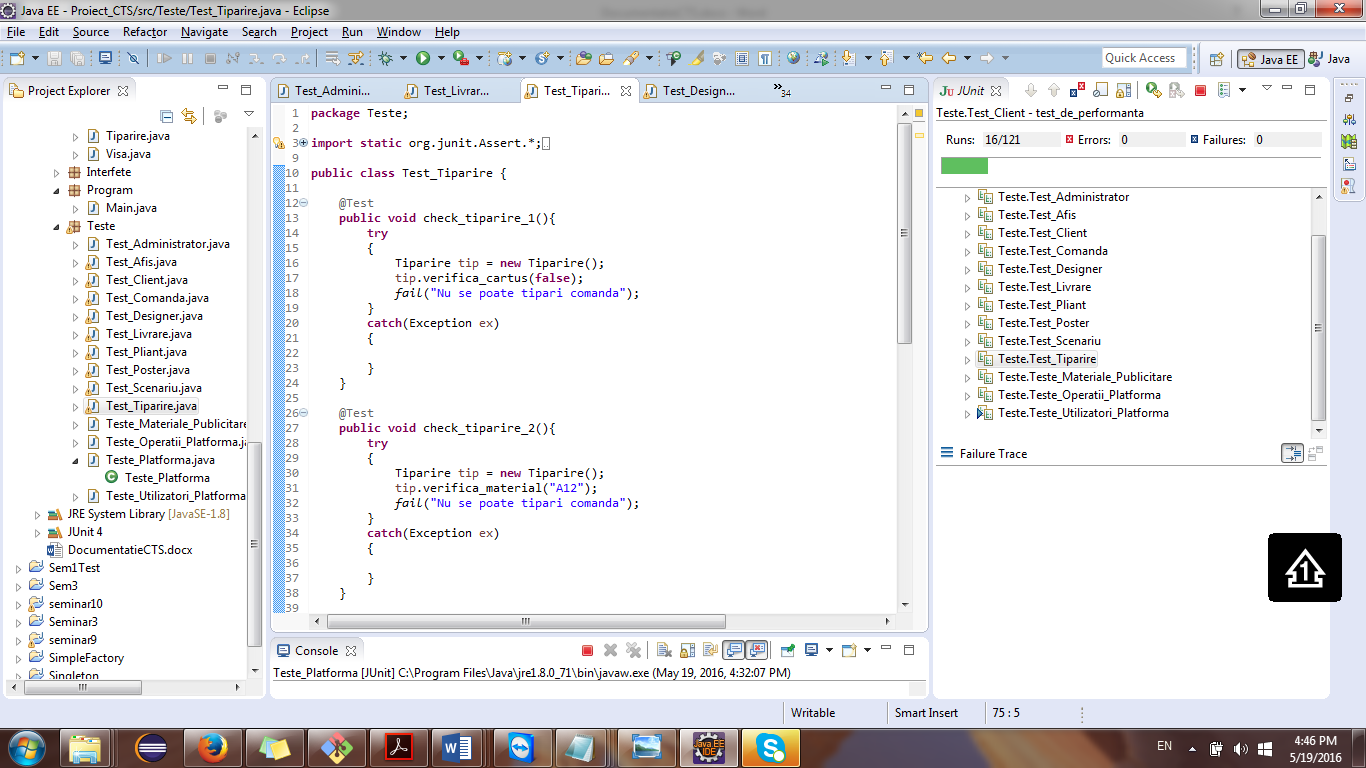
# Test\_Livrare

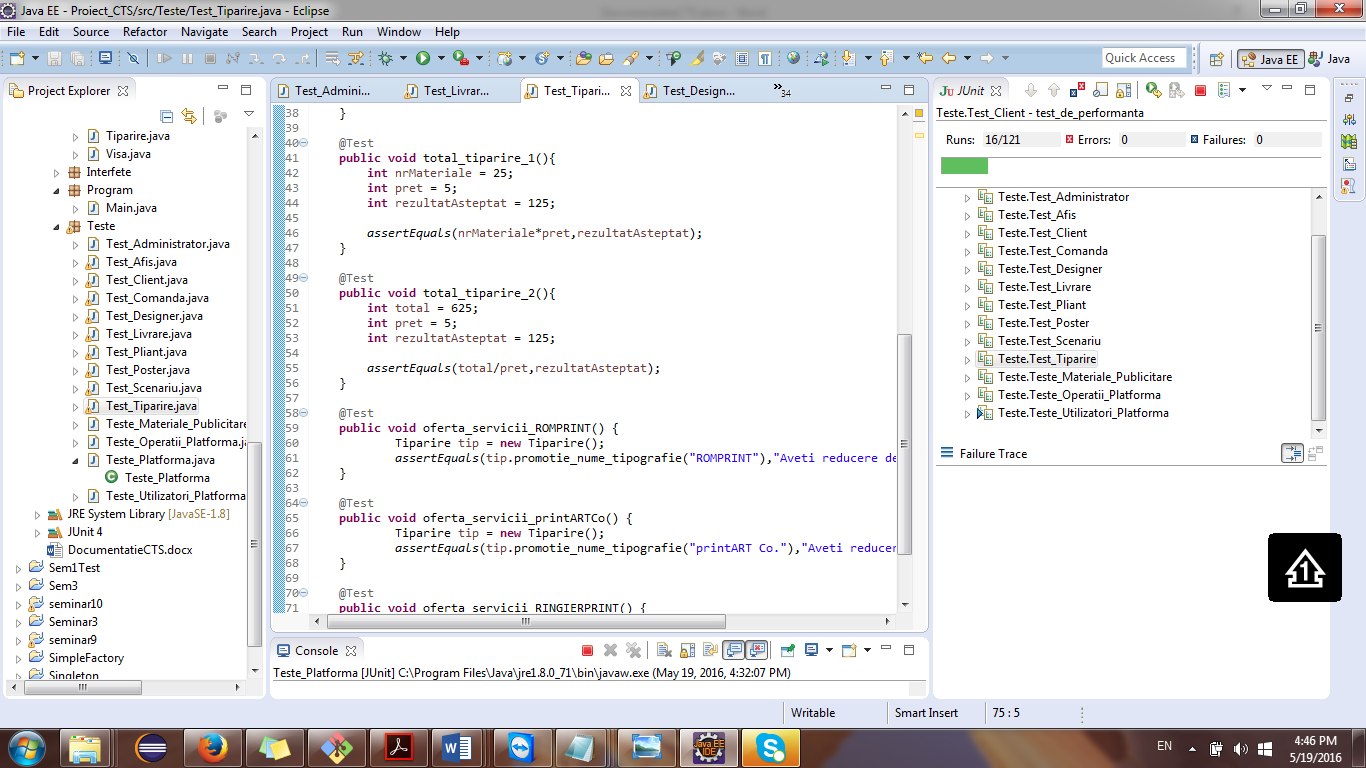




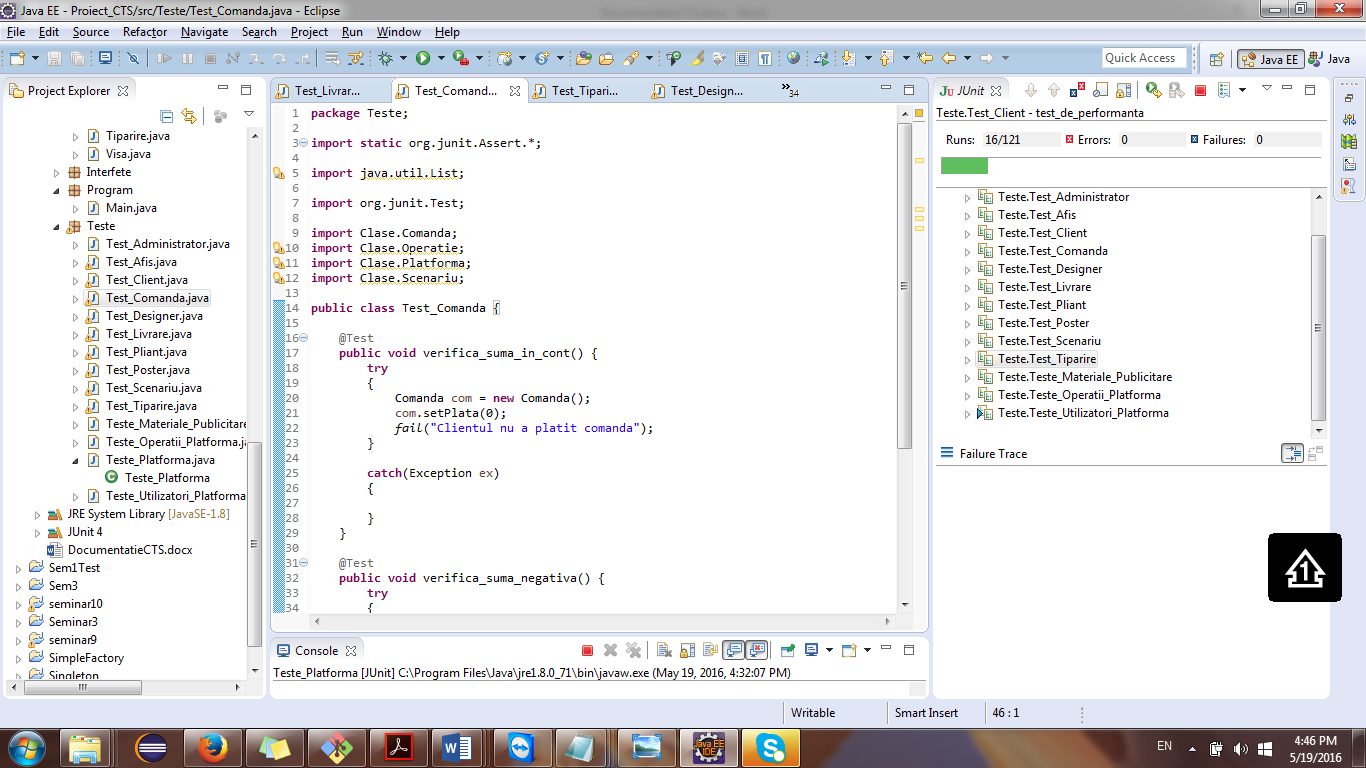


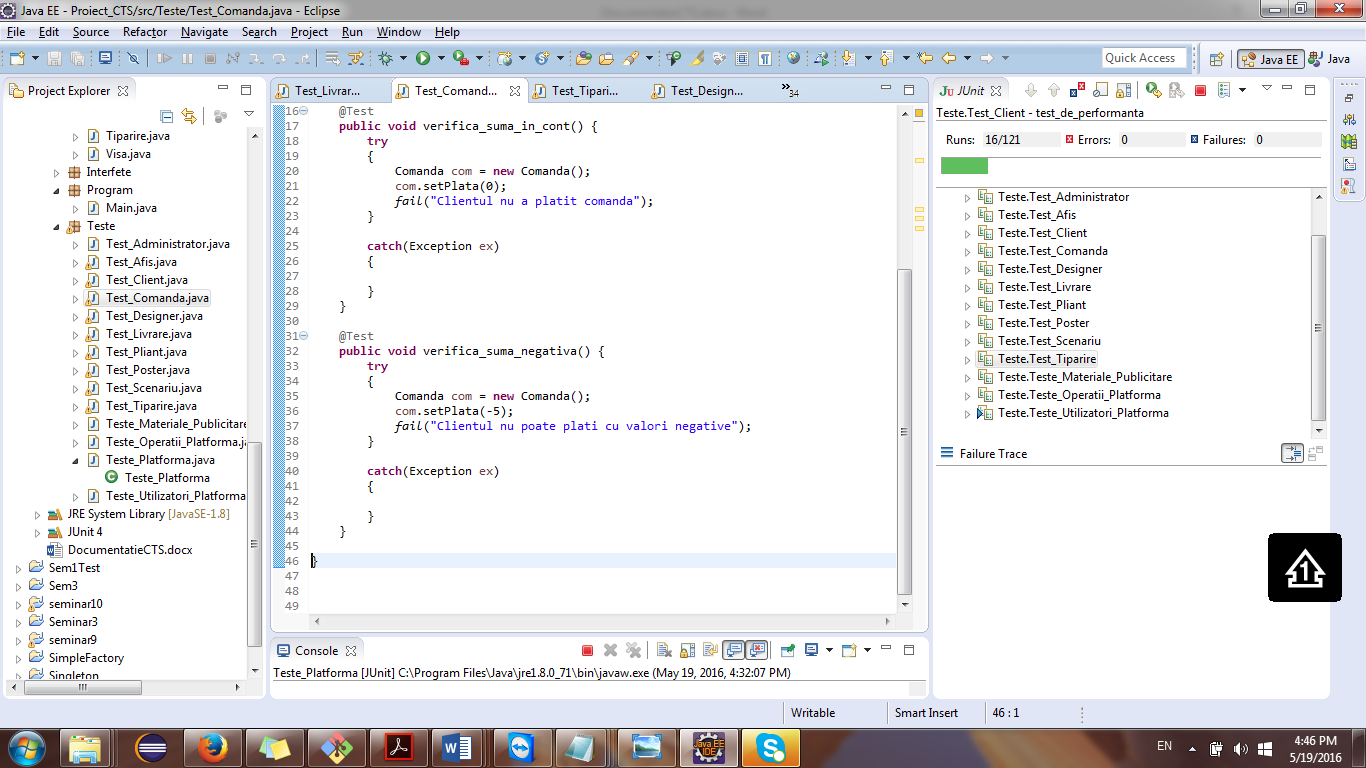
# Test\_Tiparire



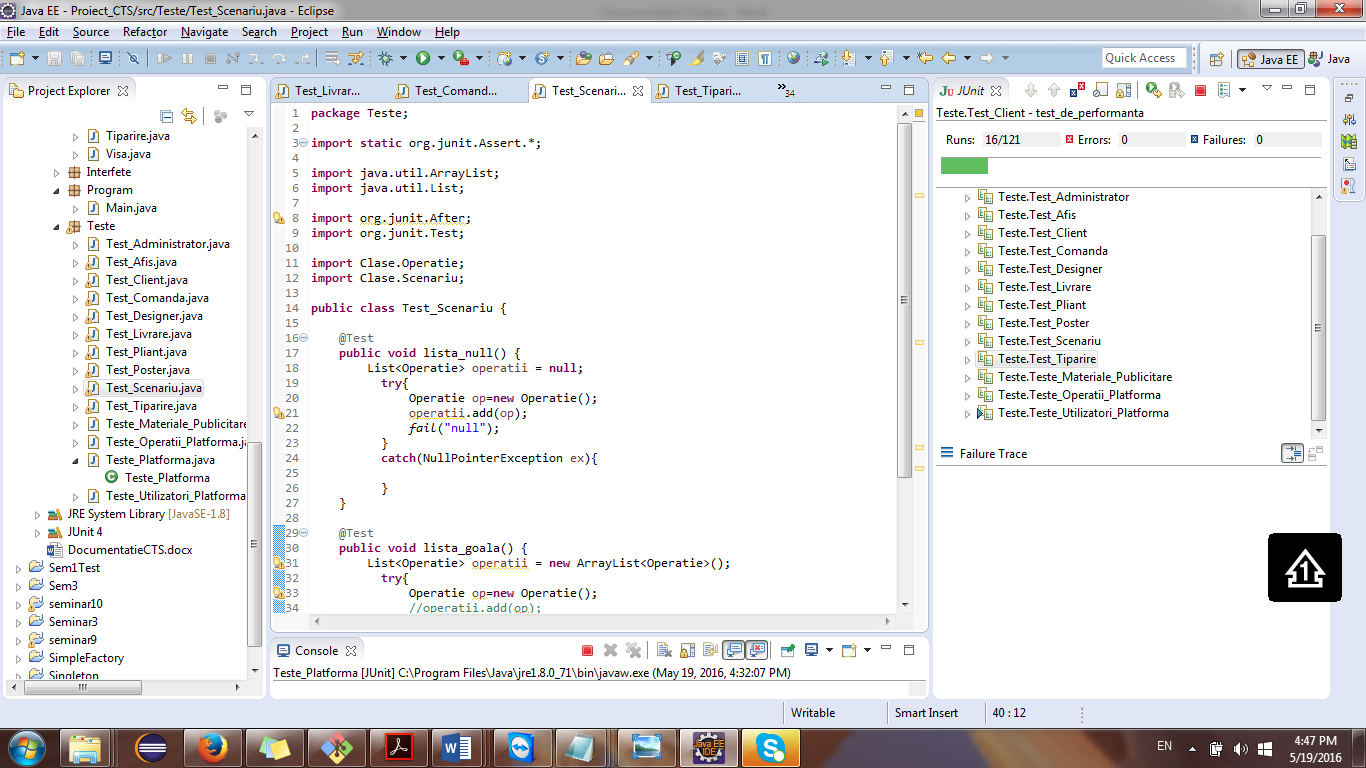


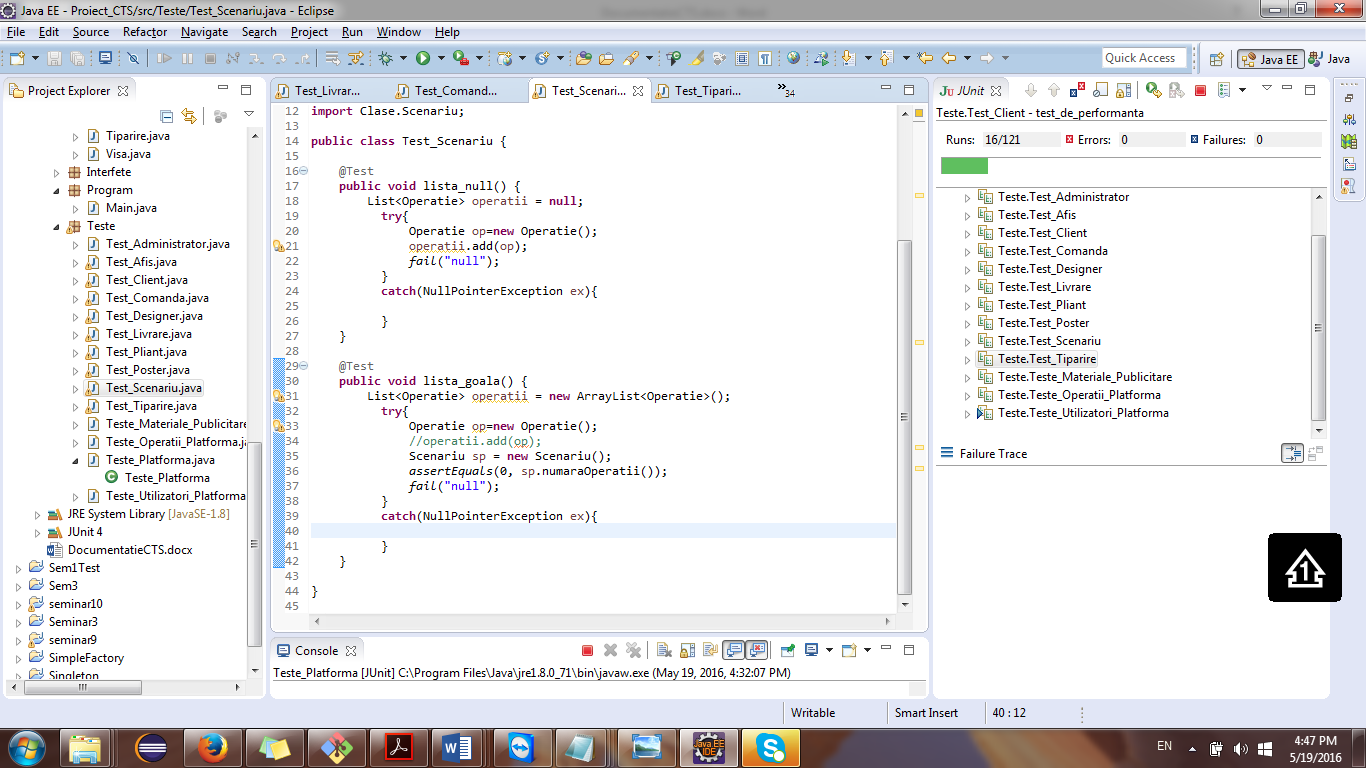
# Test\_Comanda





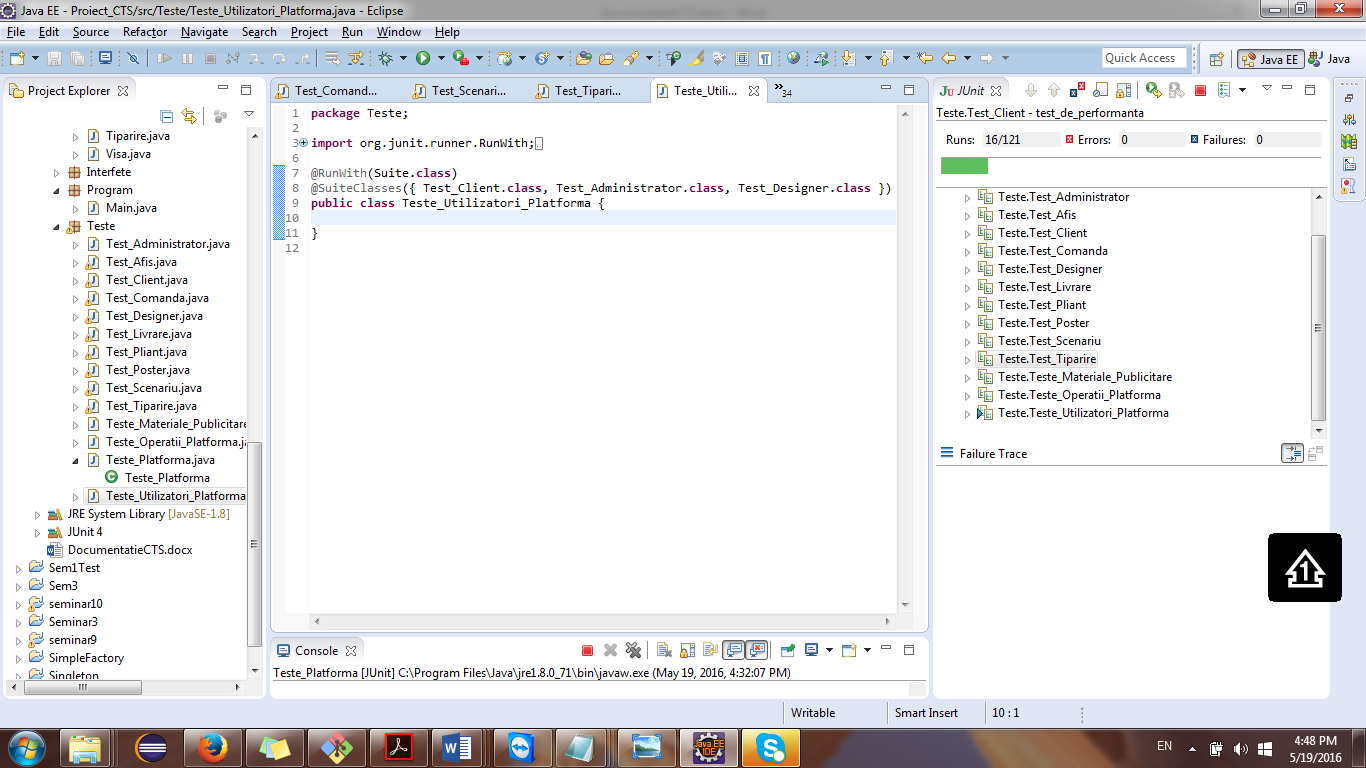
# Test\_Scenariu



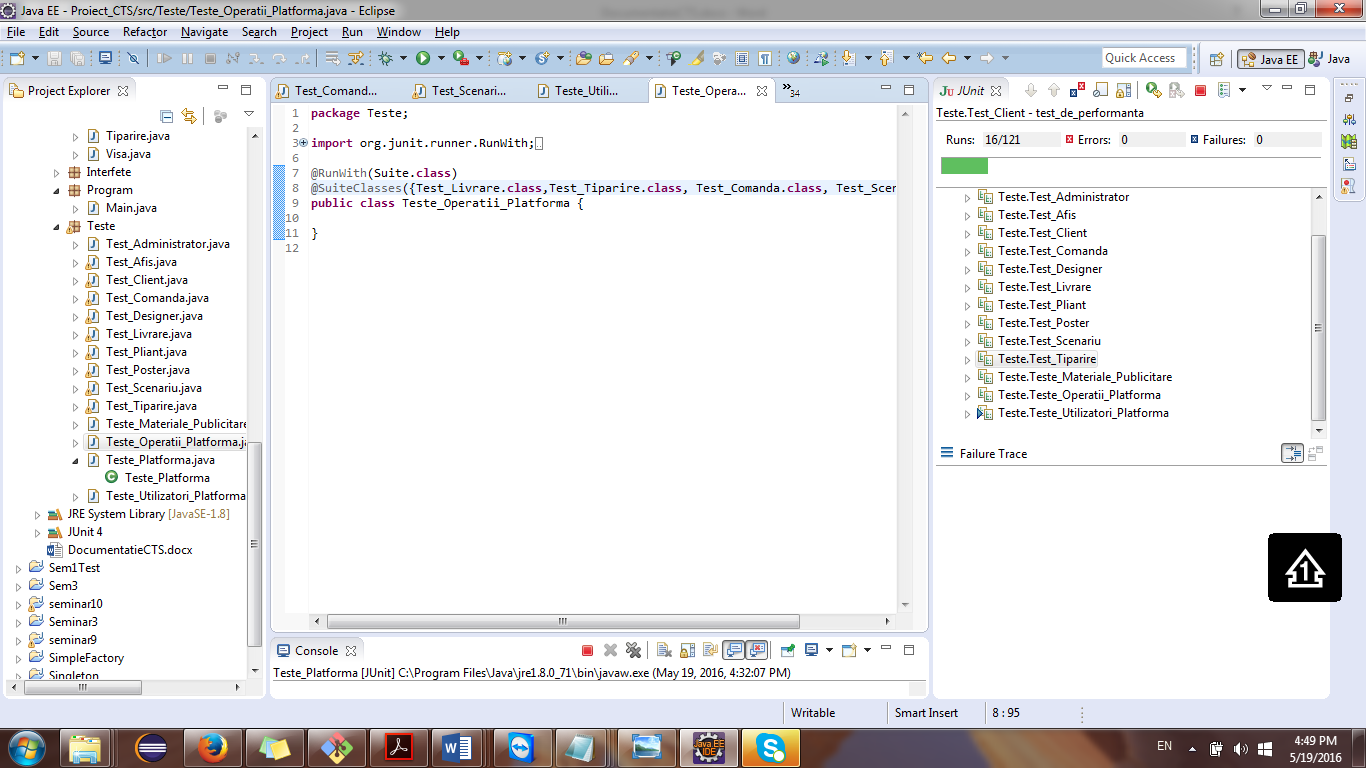


# Definirea Suite Case-urilor

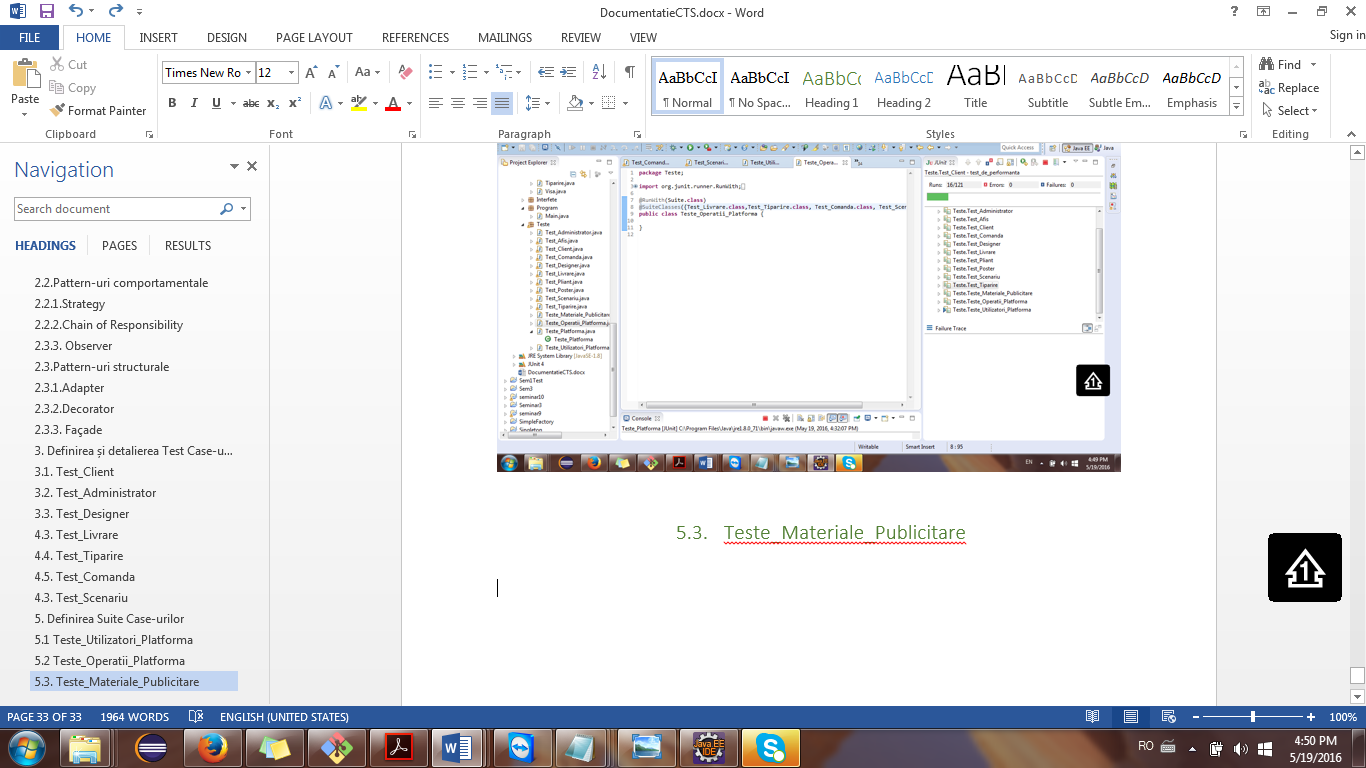
# 5.1 Teste\_Utilizatori\_Platforma



# 5.2 Teste\_Operatii\_Platforma



# Teste\_Materiale\_Publicitare



# Teste\_Platforma

