MINISTERUL EDUCAȚIEI ȘI CERCETĂRII

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICA SI CALCULATOARE
CATEDRA DE CALCULATOARE

SISTEM DISTRIBUIT DE ADMINISTRARE A UNEI LIBRARII VIRTUALE

PROIECT DE DIPLOMA

Autor: **Catalin Cosmin MURESAN**

Coordonator: **S. L. Ing. Cosmina IVAN**

2009

MINISTERUL EDUCAȚIEI ȘI CERCETĂRII

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICA SI CALCULATOARE
CATEDRA DE CALCULATOARE

VIZAT DECAN,

Prof. Dr. Ing. Sergiu NEDEVSCHI

VIZAT SEF CATEDRA**,**

Prof. Dr. Ing. Rodica POTOLEA

Autor : **Catalin Cosmin MURESAN**

SISTEM DISTRIBUIT DE ADMINISTRARE A UNEI LIBRARII VIRTUALE

1. **Enuntul temei:** *Realizarea unei aplicatii web care sa implementeze o librarie virtuala. Se va implementa atat partea de utilizator, care permite vizualizarea site-ului si efectuarea comenzilor de carti on-line, cat si partea de administrare a site-ului.*

2. **Continutul proiectului:** *Introducere, Obiective si specificatia proiectului, Studiu bibliografic, Analiza si proiectare, Implementare, Instalare si testare, Concluzii*

3. **Locul documentatiei:** Universitatea Tehnica din Cluj Napoca

4. **Data emiterii temei:** 1 noiembrie 2008

5. **Data predarii:** 19 iunie 2009

Semnatura autorului _____

Semnatura coordonatorului _____

MINISTERUL EDUCAȚIEI ȘI CERCETĂRII

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICA SI CALCULATOARE
CATEDRA DE CALCULATOARE

Declaratia autorului,

Subsemnatul Catalin Cosmin MURESAN, student al Facultatii de Automatica si Calculatoare, Universitatea Tehnica din Cluj-Napoca, declar ca ideile, analiza, proiectarea, implementarea, rezultatele si concluziile cuprinse in acest proiect de diploma constituie efortul meu propriu, mai putin acele elemente ce nu imi apartin, pe care le indic si recunosc ca atare.

Declar de asemenea ca, dupa stiinta mea, lucrarea in această forma este originala si nu a mai fost niciodata prezentata sau depusa in alte locuri sau alte institutii decat cele indicate in mod expres de mine.

Data: 19 iunie 2009

Autor: **Catalin Cosmin MURESAN**

Numar matricol :

Semnatura:_____

Cuprins

1. Introducere

Comertul electronic a devenit o modalitate eleganta si facila de a efectua anumite tranzactii, utilizand internetul. Fiecare magazin, librarie, editura, echipa de fotbal, canal TV etc care se respecta are si un site care sa le ofere posibilitatea clientilor si navigatorilor obisnuiti, de a vizualiza anumite informatii sau produse si de a efectua anumite tranzactii. Pentru realizarea comertului electronic, este nevoie de proiectarea aplicatiilor distribuite in internet.

Tema aleasa este *"Proiectarea unui sistem distribuit de administrare a unei librarii virtuale* ". Sistemul implementat permite administrarea unui site care contine produse specifice unei librarii (carti), vizualizarea acestuia de catre mai multi clienti si efectuarea de comenzi, de catre utilizatorii autentificati.

Motivatia alegerii temei curente este strans legata de specificul acesteia. Programarea distribuita a devenit, in ultimii ani, cea mai utilizata in aplicatiile practice construite de diferite firme de soft. Invatarea principiilor acesteia, dar si a tehnologiilor moderne pe care le utilizeaza, constituie un avantaj pentru oricine vrea sa isi creeze o cariera in domeniul calculatoarelor. Tema aleasa mi-a permis sa inteleg si sa aprofundez modul in care se proiecteaza si functioneaza un sistem distribuit. Mai mult, implementarea unei asemenea aplicatii mi-a dat ocazia sa invat, la nivel practic, limbaje si tehnologii moderne, precum Java (editia *Core* si *Enterprise*). Nu in ultimul rand, am avut oportunitatea sa studiez si sa utilizez un framework (Struts2) care faciliteaza proiectarea si implementarea aplicatiilor distribuite in internet, prin faptul ca ofera o arhitectura gata construita pentru acestea. In concluzie, prin alegerea temei curente am urmarit trei obiective : insusirea cunostintelor necesare proiectarii unor sisteme distribuite, asimilarea tehnologiilor moderne cele mai frecvent utilizate pentru implementarea acestora, invatarea unor framework-uri care sa faciliteze proiectarea si implementarea sistemelor distribuite, utilizand diverse tehnologii de actualitate.

2. Obiective si specificatia proiectului

Obiectivele proiectului sunt urmatoarele : crearea unei aplicatii care sa indeplineasca un set de cerinte bine definit si care sa corespunda cerintelor actuale din domeniul aplicatiilor web.

Tipurile de cerinte asociate se impart in doua categorii : cerinte functionale (care tin de multitudinea de functionalitati care trebuie sa le aiba sistemul) si cerinte non-functionale (care se refera la calitatea modului in care sunt implementate cerintele). [7]

Exista trei tipuri de useri care pot utiliza aplicatia curenta, fiecare avand anumite drepturi si restrictii : vizitator (V), user inregistrat, dar nelogat (UN), user inregistrat, dar logat (UL), administrator nelogat (AN), administrator logat (AL).

Cerintele functionale ale aplicatiei sunt :

- Inregistrare in baza de date a site-ului (V, UN)
- Logare-in (UN , AN)
- Logare-out (UL, AL)
- Updatare cont (UL)
- Vizualizare categorii cartii (V, UL, UN, AL, AN)
- Vizualizare detalii despre o anumita carte (V, UN, UL, AN, AL)
- Vizualizare carti noi / carti bine vandute (V, UN, UL)
- Vizualizare comentarii validate despre o carte (V, UN, UL)
- Adaugare comentarii despre o carte (V, UN, UL)
- Cautare carte - dupa autor, titlu, editura, categorie (V, UN, UL, AN, AL)
- Adaugare carte in cosul de cumparaturi (V, UN, UL)
- Eliminare carte din cosul de cumparaturi (V, UN, UL)
- Vizualizare cos de cumparaturi (V, UN, UL)
- Updatare numar de carti de un anumit fel din cosul de cumparaturi (V, UN, UL)
- Efectuare comanda (UL)

- Modificare categorie de carti (AL)
- Stergere categorie de carti (AL)
- Adaugare categorie de carti (AL)
- Vizualizare comentarii validate / nevalidate despre o carte (AL)
- Validare / Invalidare comentarii despre o carte (AL)
- Stergere comentarii despre o carte (AL)
- Modificare detalii despre o carte (AL)
- Stergere carte (AL)
- Adaugare carte (AL)
- Vizualizare utilizatori inregistrati (AL)
- Stergere utilizatori inregistrati (AL)
- Vizualizare comenzi efectuate (AL)
- Stergere comenzi efectuate (AL)
- Finalizare comenzi efectuate (AL)

Cerintele non-functionale ale aplicatiei sunt :

- Corectitudinea sistemului (a implementarii cerintelor functionale)
- Validarea datelor de intrare in sistem
- Securitatea, stabilitatea si scalabilitatea sistemului
- Performanta sistemului (eficienta implementarii cerintelor functionale)
- Reutilizarea componentelor sistemului
- Baza de date din spatele site-ului poate fi accesata prin internet concurent, de catre utilizatori multipli, cu diverse nivele de acces (sistemul este distribuit)

3. Studiu bibliografic

3.1 Concepte fundamentale in programarea web

3.1.1 Arhitectura client-server

Un server web preia cererea unui client si ii da acestuia un raspuns. Server-ul poate reprezenta atat masina fizica – calculatorul (hardware) cat si o aplicatie de tip server, care ruleaza pe un calculator (software). Un browser web permite unui client sa ceara o resursa. Server-ul web preia cererea acestuia, cauta resursa si ii returneaza raspunsul clientului (daca server-ul nu gaseste resursa, il anunta pe client ca nu a gasit-o). Resursa gasita poate fi o pagina HTML, o poza, un document PDF etc. Cererea clientului contine numele si adresa (URL-ul) resursei cautate. Pentru a putea comunica, clientii si server-ele utilizeaza HTML si HTTP. Cand un server raspunde la o solicitare, el ii trimite browser-ului un set de instructiuni scrise in HTML (HyperText Markup Language) care ii spun acestuia cum sa afiseze continutul raspunsului. Majoritatea browser-elor web (toate cele noi) cunosc HTML (si versiunile mai noi ale acestuia). Cele mai multe conversatii dintre clienti si servere, in cadrul aplicatiilor web, se bazeaza pe protocolul HTTP, care permite formularea de cereri si raspunsuri. [1]

Cand un server web trimite o pagina HTML unui client, utilizeaza protocolul HTTP (HyperText Transfer Protocol). O conversatie HTTP este formata dintr-o secventa request / response (efectuat de browser / executat de server). Elementele cheie legate de stream-ul request sunt : metoda HTTP corespunzatoare cererii, pagina care va fi accesata (un URL), parametrii form-ului (analogi argumentelor unei metode). Elemente asociate stream-ului response sunt : un cod de status (care indica daca cererea a avut succes sau nu, daca s-a gasit un raspuns), tipul continutului raspunsului (text, poza, HTML etc), continutul efectiv al raspunsului. Un raspuns HTTP poate contine HTML. [1]

O cerere HTTP contine in primul rand numele metodei HTTP, care ii indica server-ului tipul de cerere facuta si modul in care va fi format restul mesajului. Protocolul HTTP are cateva metode, dar cele mai utilizate sunt GET si POST. Restul metodelor (HEAD, TRACE, PUT, DELETE, OPTIONS, CONNECT) sunt folosite mai rar. Metoda GET cere server-ului sa preia o resursa si sa o trimita inapoi (o pagina HTML, un JPEG, un PDF etc). Numarul total de caractere dintr-un GET e limitat (depinde de server). Datele trimise cu GET sunt adaugate la URL, in browser, deci sunt expuse (atentie sa nu se puna o parola sau o informatie confidentiala intr-un GET). Prin metoda POST se poate solicita server-ului ceva (o resursa) si in acelasi timp se pot trimite date acestuia. Aceasta este utilizata de browser pentru cereri mai complexe adresate server-ului (de exemplu, daca un user a completat o fereastra mai mare cu date si vrea ca acele date sa fie puse de server intr-o baza de date). Datele trimise inapoi server-ului sunt cunoscute ca “message body” si pot avea dimensiuni mai mari. [1]

Un raspuns HTTP contine un header (antet) si un body (corp). Continutul header-ului il informeaza pe browser de protocolul utilizat si daca cererea poate fi onorata cu succes, dar cuprinde si tipul continutului inclus in corpul raspunsului (body). Corpul raspunsului contine ceea ce browser-ul va afisa pe ecran, raspunsul efectiv la cererea facuta. [1]

URL-ul (Uniforme Resource Locator) este utilizat pentru localizarea resurselor cautate. Acesta este format din : protocolul utilizat pentru comunicare, numele unic al server-ului cautat (se mapeaza la o adresa IP unica, de forma : “xxx.yyy.zzz.aaa” ; in loc de numele server-ului se poate pune adresa IP specifica acestuia) , portul (e optional : un server poate avea mai multe porturi ; o aplicatie server e identificata de un port ; daca nu se specifica nici un port in URL, se ia portul 80 din oficiu deoarece acesta este portul default pentru server-ele web), calea (catre locatia de pe server, unde se afla resursa - se utilizeaza sintaxa UNIX pentru a descrie ierarhia directoarelor de pe server-ul web), resursa (numele continutului cerut - o pagina HTML, un servlet, o imagine, video etc ; daca partea aceasta optionala a URL-ului este exclusa, majoritatea server-elor vor cauta, din oficiu, in index.html). [1]

Un port TCP este un numar pe 16 biti care identifica in mod unic un program soft de pe un server hardware (server-ul software HTTP ruleaza pe portul 80, server-ul TELNET pe portul 23, FTP pe portul 21, SMTP pe portul 25). Poate avea valori intre 0 si 65535. Porturile TCP de la 0 la 1023 sunt rezervate pentru servicii speciale, deci nu se vor utiliza pentru aplicatiile server proprii. Fara numere de port-uri, un server nu ar sti la ce aplicatie vrea sa se conecteze un client. [1]

Server-ele web pot returna pagini web statice. O pagina statica este o pagina care “sta” intr-un director. Server-ul o gaseste si o reda clientului, la cererea acestuia. Mai jos este prezentat un prim model al arhitecturii client-server. Clientul ii cere server-ului o pagina statica HTML, pe care acesta din urma o poate returna direct. [9]

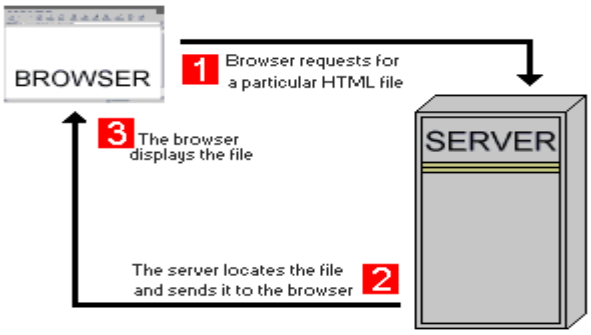


Figura 3.1 : Model I de arhitectura client-server

Daca insa se doreste o pagina dinamica (de exemplu se vrea afisarea orei curente pe pagina), nu e de ajuns aplicatia de tip server care returneaza doar pagini statice. Aceasta trimite cererea mai departe unei alte aplicatii server ajutatoare, apoi preia raspunsul acesteia si il trimite la client. Clientul nu stie ca altcineva a facut munca, si nu aplicatia server pe care a apelat-o. [1]

Exista doua lucruri pe care server-ele web nu le pot face singure : redarea continuturilor, a paginilor dinamice (stiu sa redea doar pagini statice ; o aplicatie ajutatoare, separata, cu care server-ul web poate comunica, stie sa construiasca pagini dinamice, just-in-time - care nu au existat inainte de cererea clientului) si salvarea datelor pe server. Pentru a procesa datele introduse de user de la tastatura intr-un form (a le salva intr-un fisier sau intr-o baza de date sau chiar a le folosi in pagina raspuns returnata clientului), e nevoie de o aplicatie ajutatoare. Termenul non-Java pentru o aplicatie web ajutatoare, de tip server, este un program de tip CGI (Common Gateway Interface) – scris utilizand scripturi Perl sau chiar alte limbaje (C, Python, PHP). [1]

Servlet-urile (componente Java) si CGI joaca rolul unor aplicatii ajutatoare pentru server-ul web. Avantajele servlet-urilor fata de CGI sunt urmatoarele : eficienta (pentru fiecare apel al CGI, server-ul web lanseaza un nou proces dar la apelurile servlet-urilor, JVM creeaza doar un nou thread => consum mai mic de resurse), usurinta proiectarii (dezvoltarea unui CGI in Perl sau C reclama respectarea unui mare numar de restrictii), putere (servlet-urile Java pot comunica usor cu server-ul web, in timp ce CGI-urile nu prea), portabilitate (Java este un limbaj perfect portabil), solutii ieftine (Java, Apache sunt solutii free). E neplacut la servlet-uri faptul ca paginile HTML sunt construite prin apeluri ale metodei *println*. JSP-urile rezolva aceasta problema, caci permit scrierea codului Java intr-o pagina HTML, in loc de a pune HTML in codul Java. [1]

In al doilea model client-server, clientul ii cere server-ului o pagina dinamica HTML, iar acesta o returneaza cu ajutorul unei aplicatii ajutatoare (script CGI sau servlet). [9]

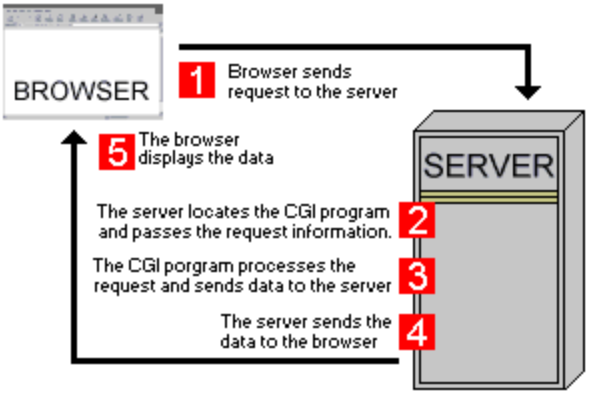


Figura 3.2 : Model II de arhitectura client-server

3.1.2 Definitia si rolul container-ului web

Servlet-urile sunt controlate de catre o aplicatie Java, numita container (Tomcat, de exemplu). Acesta gestioneaza resursele si ciclul de viata al servlet-urilor. Cand server-ul web (Apache, de exemplu) primeste o cerere pentru apelul unui servlet, acesta inmaneaza cererea nu servlet-ului, ci container-ului in care acesta este rulat. Container-ul este cel care apeleaza metodele servlet-ului. [1]

Daca nu ar exista servlet-uri si containere, am putea scrie programe Java care sa gestioneze cererile dinamice care vin la o aplicatie web de tip server (Apache, de exemplu), daca am configura server-ul astfel incat sa se poate invoca din el aplicatia Java si daca am scrie codul J2SE, cu socket-uri. [1]

Container-ul ofera in plus urmatoarele facilitati [1] :

- Suport pentru o comunicare facila a servlet-urilor cu server-ul web => Nu e nevoie sa se construiasca obiecte de tip *ServetSocket*, sa se asculte pe un port, nici sa se creeze stream-uri. Container-ul stie protocolul dintre server-ul web si el insusi, deci servlet-ul nu trebuie sa gestioneze API-ul intermediar dintre, sa spunem, server-ul web Apache si codul aplicatiei web proprii. Trebuie ca programatorul sa se concentreze pe logica si codul scris in servlet – partea logica a aplicatiei, in loc sa scrie cod

pentru asigurarea securitatii, a multithreading-ului si a comunicarii in retea.

- Gestionarea ciclului de viata al servlet-urilor => Container-ul are ca sarcini incarcarea claselor, instantierea si initializarea servlet-urilor, invocarea metodelor acestora, facerea instantelor (la servlet-uri) eligibile pentru garbage-collector.

- Suport pentru multithreading => Container-ul creeaza automat un nou thread Java pentru fiecare cerere pe care o primeste. Cand servlet-ul a terminat de rulat metoda HTTP service corespunzatoare cererii clientului, thread-ul asociat moare.

- Support pentru securitate => Se poate utiliza fisierul XML Deployment Descriptor pentru a configura sau modifica parametrii de securitate ai aplicatiei si aceasta fara a scrie si a recompila cod Java.

- Suport pentru JSP-uri => Container-ul este cel care transforma JSP-urile in cod Java.

3.1.3 Functionalitatile si limitarile Servlet API

Exista doua caracteristici esentiale ale protocolului HTTP : acesta este *stateless* (dupa ce server-ul trimite un raspuns unui client, uita cine este si, la urmatoarea cerere pe care o face acesta, nu il mai recunoaste) si *text-based* (in form-ul care face cererea HTTP, toate datele sunt reprezentate ca String-uri, pe baza de text => e necesara o conversie de la String-uri la tipurile de date Java si invers). [2]

Specificatia Servlet defineste o aplicatie web ca pe o colectie de servlet-uri (pentru a satisface cererile clientilor), pagini HTML, clase si alte resurse. Acestea sunt impachetate impreuna intr-o structura de directoare specifica si arhivate cu o arhiva WAR. Un fisier WAR este o versiune specializata a unui fisier Java JAR. Arhiva WAR este pusa pe un anumit container web care va prelua si va controla ciclul de viata si executia servlet-urilor. Pe langa realizarea comunicarii client-server, servlet-urile pun la dispozitie mecanismul de sesiune, prin care coreleaza cererile unui client astfel incat se poate mentine o stare de conversatie neintrerupta de-a lungul mai multor cereri ale aceluiasi client, cu alte cuvinte pentru o sesiune intreaga in care clientul este logat. In afara de mecanismul de sesiune, Servlet API nu ofera alte functionalitati de nivel inalt ale aplicatiei. Este doar un mecanism de nivel scazut care asigura si incapsuleaza detaliile comunicarii dintre clienti si servere si care ofera baze solide pentru construirea aplicatiilor web robuste. [2]

.

Functionalitatile de nivel inalt, necesare, care nu sunt puse la dispozitie de Servlet API, sunt [2] :

- Legarea parametrilor cererii de tipurile de date Java => Fiind un protocol *text-based*, HTTP trebuie sa-si reprezinte parametrii cererii intr-o codificare text. Cand acestia ajung in aplicatia web, trebuie sa fie convertiti la tipurile de date native Java. API-ul Servlet nu realizeaza acest lucru, caci parametrii preluati de la obiectele servlet asociate cererii sunt reprezentati tot ca si String-uri. Conversia manuala a acestor date la tipuri de date Java nu e dificila, dar poate consuma timp si poate facilita aparitia erorilor.

- Validarea datelor care intra in sistem => Exista doua nivele de validare. In primul rand, String-ul trebuie sa fie o reprezentare valida a tipului Java de date la care se doreste conversia lui. Apoi, datele trebuie sa fie valide din punct de vedere logic, adica trebuie sa fie in setul de valori acceptabile.

- Apelurile efectuate la logica de business si layer-ul de date ale aplicatiei => Intr-o aplicatie cu un design bun, logica de business si layer-ul de date ale aplicatiei nu ar trebui sa tina cont de faptul ca sunt apelate dintr-o aplicatie web sau desktop. E necesara o generalizare a cererilor efectuate catre datele si business-ul aplicatiei (fiecare procesare a unei cereri consta dintr-o secventa de operatii care trebuie efectuate si reprezinta o actiune – intr-un framework orientat pe actiuni).

- Realizarea unor vizualizari cat mai user-friendly si mai complexe pentru raspunsurile date clientului (utilizand HTML, CSS, AJAX etc).

- Internationalizarea aplicatiei => Permite construirea unei aplicatii web care poate identifica localizarea fiecarui user si poate oferi limbajul specific localizarii determinate anterior.

Toate aceste task-uri comune tuturor aplicatiilor web, care nu sunt oferite de Servlet API, sunt implementate automat de framework-uri, care ofera solutii reutilizabile pentru acestea. [2]

3.1.4 Conceptul de framework. Utilitatea unui framework

Un framework este un soft structural care care automatizeaza task-urile comune aplicatiilor web si construieste o solutie arhitecturala care poate fi mostenita usor de aplicatiile implementate pe el. Un framework ofera mecanisme pentru solutii avantajoase si, in majoritatea cazurilor, automate, pentru task-urile comune aplicatiilor web, scutind developerii de eforturi in plus in acest sens. Astfel, un bun framework va pune la dispozitie mecanisme interne pentru conversia datelor din reprezentarea String HTTP la tipurile de date Java, pentru validarea datelor, pentru separarea layer-ului de business si date ale aplicatiei de layer-ele de web ale acesteia, pentru internationalizarea si interfata aplicatiilor. Totodata, un framework ofera o solutie arhitecturala comuna pentru aplicatiile web. Daca arhitectura e buna si e greu de refuzat, aplicatia web creata utilizand framework-ul respectiv o poate mosteni fara probleme. [2]

Astfel se generalizeaza task-urile comune care trebuie indeplinite pentru un anumit domeniu si se ofera o platforma pe care aplicatiile domeniului corespunzator pot fi construite mai repede. Nu e neaparata nevoie sa se utilizeze un framework la construirea unei aplicatii web, insa daca se alege aceasta varianta, se va pierde timp pentru rezolvarea task-urilor specificate anterior si pentru construirea unei arhitecturi viabile. [2]

3.2 Fundamentarea teoretica a sistemelor soft utilizate

3.2.1 Sisteme software pentru partea de server

Pe partea de server, am utilizat platforma Java, impreuna cu Java Core, si tehnologia J2EE (prin framework-ul Struts2). Intrucat framework-ul Struts2 este construit pe baza tehnologiilor Java Servlet si Java Server Pages (din J2EE), vor fi descrise si acestea din urma.

3.2.1.1 Limbajul Java

Tehnologia Java faciliteaza dezvoltarea aplicatiilor pentru comunicarea dintre diverse sisteme, reprezentand atat un limaj de programare cat si un set de platforme specializate. Caracteristicile principale ale limbajului Java sunt urmatoarele : este independent de platforma, orientat obiect, concurent, distribuit sigur, performant. [5]

Tehnologia Java s-a dezvoltat de-a lungul timpului, ajungand sa includa urmatoarele platforme : Java 2 Platform, Standard Edition (J2SE – permite dezvoltarea aplicatiilor Core Java si Desktop Java) ; Java 2 Platform, Enterprise Edition (J2EE - permite si simplifica dezvoltarea aplicatiilor care ruleaza in internet, a aplicatiilor distribuite si comerciale) ; Java 2 Platform, Micro Edition (J2ME – set de tehnologii pentru dispozitive precum telefoane mobile, imprimante etc) ; Java Card (adapteaza platforma Java pentru dispozitive si carduri inteligente). In figura de mai jos sunt prezentate elementele de baza ale limbajului Java : [13]

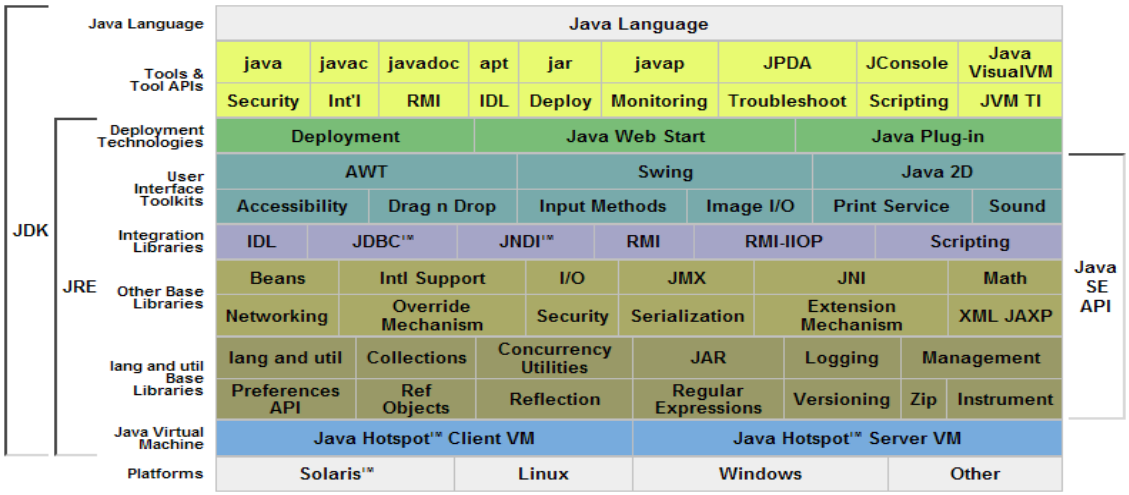


Figura 3.3 : Elemente de baza ale platformei Java

3.2.1.2 Tehnologia Java Servlet

Servlet-urile sunt componente software pe partea de server, scrise in Java, care extind funtionalitatea unui server. Ele nu afiseaza o interfata grafica catre utilizator, ci returneaza doar rezultate catre client, de obicei sub forma unui document HTML. Sunt clase Java care se conformeaza unei interfete specifice care poate fi apelata de server. [1]

Servlet-urile furnizeaza un cadru pentru crearea de aplicatii care implementeaza paradigma cerere / raspuns. Cand un navigator trimite o cerere catre server, acesta o trimite mai departe unui servlet. Servlet-ul proceseaza cererea (eventual accesand o baza de date) si

construieste un raspuns convenabil (de obicei in HTML) care este trimis server-ului, care, la randul lui, il returneaza clientului. [1]

3.2.1.3 Tehnologia JSP

Paginile JSP (Java Server Pages) sunt componente utilizate pe partea de server, care combina HTML, XML, servlet-uri si JavaBeans pentru a crea aplicatii web distribuite. Sintaxa scrierii permite separarea partii dinamice de partea statica a unei astfel de pagini web. Partea statica se scrie ca un fisier normal HTML sau XML, care este invocat de browser. Secventele dinamice sunt distribuite in acelasi fisier, intercalate cu elementele din partea statica. Separarea secventelor dinamice se face cu niste tag-uri speciale – elemente de scripting. Aceasta tehnologie este o extensie la tehnologia Java Servlet. In loc sa fie scris cod Java care sa genereze pagini web, se vor crea pagini web care contin elemente dinamice. Cand se primeste o cerere de pagina JSP, se creeaza un servlet din acea pagina si acesta e executat, iar rezultatul este trimis ca raspuns la cererea primita. Un avantaj important al JSP-urilor este faptul ca se separa continutul HTML static de continutul dinamic. In cazul servlet-urilor, orice modificare referitoare la design-ul paginii web implica recompilarea intregului servlet. La JSP-uri, partea de generare a continutului dinamic e pastrata separat de cea statica prin utilizarea componentelor JavaBeans externe. Deci se permite verificarea la compilare a HTML-ului sau XML-ului si nu mai e necesara scrierea “de mana” a servlet-urilor care sa genereze paginile web. Pretul platit e timpul necesar transformarii JSP-ului in servlet care nu este mare deoarece tranformarea se intampla numai la modificarea fisierului JSP care l-a generat. [1]

3.2.1.4 Framework-ul Struts2

3.2.1.4.1 Privire de ansamblu asupra framework-ului

Struts2 nu este doar o imbunatatire a framework-ului Struts1, ci un nou framework, bazat pe Open Symphony WebWork. Ca si Struts1, Struts2 este orientat pe actiuni si implementeaza design pattern-ul MVC (Model View Controller). O aplicatie care respecta MVC se poate imparti in trei componente care trebuie bine separate, dar, in acelasi timp trebuie sa comunice foarte bine intre ele : model - implementat de actiunile din Struts2, view - reprezentat de rezultatele framework-ului, controller - reprezentat de un *FilterDispatcher* . Struts2 are la baza tehnologia servlet-urilor, dar nu cere user-ului sa utilizeze API-ul Servlet la construirea aplicatiilor web. [2]

3.2.1.4.2 Arhitectura framework-ului

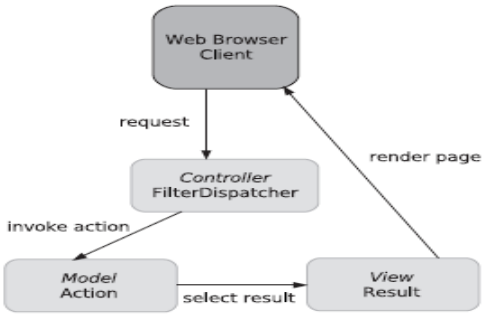


Figura 3.4 : Arhitectura framework-ului Struts2

In urmatoarele paragrafe vor fi descrise, detaliat, componentele arhitecturii de mai sus.

3.2.1.4.2.1 Controller-FilterDispatcher

Varianta MVC utilizata in Struts2 este numita deseori front-controller MVC. Rolul unui controller, in cadrul unei aplicatii web, este de a mapa cererile care vin de la client, pe actiuni Struts2. Intr-o aplicatie web, cererile HTTP pot fi vazute ca si comenzi pe care userii le initiaza din browser. Unul din task-urile fundamentale ale aplicatiilor web este rutarea acestor cereri catre actiunile specifice implementate in partea de model a aplicatiei. [2]

Rolul controller-ului, in Struts2, este implementat de un *FilterDispatcher*. Acest obiect este un filtru-servlet care analizeaza cererile care vin de la client, pentru a determina actiunile Struts2 care le vor prelua. Framework-ul realizeaza toata munca controller-ului pentru programator. Developer-ul trebuie doar sa informeze framework-ul despre modul in care URL-urile cererilor se mapeaza pe actiuni (pe ce actiune se mapeaza un anume URL). Acest lucru se poate realiza fie in fisiere Java de configurare, fie utilizand adnotari. [2]

3.2.1.4.2.2 Model - Action

Modelul reprezinta starea interna a aplicatiei. Aceasta este compusa din logica de business a aplicatiei si din modelul de date din spatele acesteia. De exemplu, daca se doreste implementarea operatiei de logare, pentru o aplicatie, in procesul de autentificare vor fi implicate atat logica de business cat si modelul de date corespunzator acesteia. Logica de business va oferi o modalitate de autentificare (va prelua username-ul si parola si va verifica daca acestea exista in baza de date). Astfel, logica de business si modelul de date se combina pentru a forma doua stari : "autentificat", "neautentificat". [2]

Modelul este implementat de actiunile din Struts2. Acestea trateaza fiecare cerere care vine de la client si care e trimisa mai departe de *FilterDispatcher* la actiunea corespunzatoare. O actiune are urmatoarele sarcini :

- Incapsuleaza logica de business a aplicatiei => Utilizeaza metoda *execute* in acest scop. Codul din interiorul metodei ar trebui sa se refere numai la logica "muncii" asociate cu cererea corespunzatoare.

- Ofera o modalitate de a retine date locale si de a le transmite apoi mai departe => Vor fi disponibile in timpul executiei logicii de business a operatiei curente. O actiune declara si implementeaza proprietati JavaBeans pentru fiecare data pe care doreste sa o transfere. Parametrii din form-ul asociat cererii vor fi transferati proprietatilor actiunilor, ale caror nume se potrivesc (transferul e realizat automat de framework, prin apelul setter-ilor definiti in actiune). Mai mult, proprietatile JavaBeans din actiune pot expune date care le contin rezultatelor care vor fi afisate.

- Returneaza String-uri care selecteaza rezultatele care vor fi afisate clientului.

Actiunile pot fi organizate in pachete. Fiecare pachet are patru atribute, dintre care numai primul trebuie neaparat specificat : numele pachetului, spatiul de nume pentru toate actiunile din pachet (genereaza URL-ul la care toate actiunile din pachet sunt mapate ; se poate transmite acelasi spatiu de nume la mai mult de un pachet ; la invocarea unei actiuni, daca aceasta nu este gasita in spatiul de nume specificat, va fi cautata in spatiul de nume implicit ; daca nu se specifica un spatiu de nume, actiunile din pachet fac parte din spatiul de nume implicit), *extends* (specifica pachetul parinte ale carui componente sunt mostenite de pachetul curent ; pachetul curent poate suprascrie membrii pachetului parinte ; majoritatea *intelligent-defaults* (tipuri de result, tipuri de interceptori), care pot fi folosite direct de programator, se afla in pachetul struts-default, definit in fisierul *struts-default.xml*), *abstract* (daca e setat la valoarea true, pachetul curent va fi folosit pentru a defini doar componente care se pot mosteni, nu si actiuni). [2]

Actiunile Struts2 nu trebuie sa implementeze interfata *Action*. E suficient ca acestea sa defineasca metoda *execute*, care returneaza un String. Totusi, implementarea interfetei *Action* costa putin si are anumite beneficii. Ea ofera cateva constante String care pot fi folosite pentru a selecta raspunsul dorit : ERROR ("error"), INPUT ("input"), LOGIN ("login"), NONE ("none"), SUCCESS ("success"). [2]

Exista si actiuni de tip “pass-through” (nu se specifica o clasa care sa implementeze actiunea, deoarece nu exista). Acestea sunt utilizate la rutarea cererilor catre alte pagini JSP. In cazul in care nu se specifica o clasa care sa implementeze actiunea curenta, se pune la dispozitie o implementare implicita a actiunii care este mostenita. Aceasta are o metoda "goala", *execute*, care nu face altceva decat sa returneze constanta SUCCESS a interfetei *Action*. [2]

Totusi, exista o alternativa mai buna decat implementarea interfetei *Action*. O actiune poate extinde clasa *ActionSupport* care ofera o implementare implicita pentru interfata *Action* si pentru alte cateva interfete care permit validarea datelor si localizarea mesajelor de eroare. Validarea datelor se poate realiza prin suprascrierea metodei *validate* si e controlata de un interceptor pus la dispozitie de framework. Mesajele de eroare pot fi localizate si in fisiere *.properties* - simple fisiere text (fiecare linie contine o cheie si o valoare, care vor fi preluate prin metode mostenite de clasa *ActionSupport* de la interfata *TextProvider*). Se poate asigura si internationalizarea aplicatiei, deoarece *ActionSupport* implementeaza si o alta interfata, *LocaleProvider*, care are metoda *getLocale* pentru a localiza user-ul curent. Totusi, mecanismele de validare si internationalizare oferite de *ActionSupport* sunt limitate si primitive. Struts2 ofera si mecanisme mai avansate pentru validarea si internationalizarea aplicatiilor web. [2]

3.2.1.4.2.3 View-Result

3.2.1.4.2.3.1 Componenta View

Interfata grafica ofera user-ului o reprezentare a starii aplicatiei. Aceasta are ca scop translaterea starii aplicatiei intr-o reprezentare vizuala cu care user-ul poate interactiona. API-ul de tag-uri Struts2 ofera posibilitatea de a crea pagini web robuste si dinamice. Struts2 pune la dispozitie mai multe tehnologii pentru partea de View : JSP, *Velocity* sau *FreeMarker*. API-ul de tag-uri Struts2 a fost implementat in toate cele trei tehnologii. Tag-urile puse la dispozitie de framework se impart in patru categorii [2] :

- Tag-uri de date => Permit preluarea datelor de pe *ValueStack* sau plaseaza obiectele

si variabilele pe *ValueStack* (*property*, *set*, *push* etc).

- Tag-uri de control => Controleaza fluxul executiei paginii, manipuleaza si parcurg datele existente (*iterator*, *if-else*).

- Tag-uri pentru alte functionalitati (*include*, *URL*, *i18*, *text*, *param*).

- Tag-uri UI => Ajuta la crearea componentelor vizuale care interactioneaza cu user-ul. La baza fiecarei componente UI din Struts2 se afla o forma HTML. Totusi, componentele UI din acest framework sunt mult mai mult decat tag-uri care afiseaza un element HTML. Ele indeplinesc si alte functionalitati, precum : genereaza markup-ul HTML (elementul HTML corespunzator tag-ului UI), leaga campurile formelor de intrare de proprietatile asociate de pe *ValueStack* (transferul de date se poate realiza in doua directii : in primul rand, datele de pe *ValueStack* pot ajunge in form-ul de intrare => prepopulare ; in al doilea rand, datele din form-urile de intrare pot ajunge inapoi in framework si pot fi transferate automat pe *ValueStack*, mapandu-se pe proprietatile Java asociate), detecteaza erorile care sunt asociate cu ele si afiseaza mesajele de eroare care pot apare in cazul operatiilor de conversie de tipuri (text - tip de data Java sau invers) si de validare de date, se pot integra bine in mecanismul de internationalizare a aplicatiei.

Arhitectura componentelor UI are mai multe elemente precum [2] :

- Tag-uri => Pot fi implementate in orice tehnologie. Framework-ul ofera implementari ale API-ului de tag-uri pentru JSP, *Velocity* si *FreeMarker*.

- Template-uri => Fiecare tag are un template *FreeMarker* care interpreteaza si reda markup-ul sau. Acesta arata ca un fisier text normal.

- Teme => Fiecare tag dispune de cateva versiuni ale template-ului asociat si fiecare dintre aceste versiuni apartine unei teme diferite. Exista patru teme puse la dispozitie de framework, pentru a reda componentele UI : simple, xhtml, css-xhtml, ajax. Initial, toate tag-urile vor fi afisate utilizand tema xhtml, dar aceasta proprietate se poate modifica. Mai mult, se pot crea noi teme, prin modificarea celor deja existente.

3.2.1.4.2.3.2 Componenta result

Framework-ul pune la dispozitie cateva tipuri de rezultate care pot fi utilizate in aplicatiile web : *dispatcher* (utilizat pentru JSP-uri sau alte resurse ale aplicatiilor web, precum servlet-uri), *redirect* (browser-ul efectueaza o cerere catre un alt URL), *redirectAction* (cererea e redirectata de browser catre o alta actiune). Toate aceste tipuri de rezultate sunt definite in *struts-default.xml*. Pentru a le putea utiliza, trebuie extins acest pachet [2] :

- RequestDispatcher (*dispatcher*) => Va fi utilizat acest rezultat cand se doreste afisarea unei pagini JSP ca rezultat al executiei unei actiuni sau forwardarea rezultatului catre un server, care are sarcina de a returna user-ului un raspuns. Acest tip este default pentru tag-ul *result*, deci nu trebuie specificat.

- ServletRedirectResult (*redirect*) => Browser-ul va efectua o noua cerere catre URL-ul specificat prin operatia de redirectare. Noul URL catre care e indreptata redirectarea inlocuieste vechiul URL din browser. Redirectarea e utila atunci cand se doreste ca user-ul sa nu ajunga la URL-ul anterior apasand butonul Reload. La efectuarea unei redirectari, starea cererii curente si datele asociate cu aceasta se pierd. Daca e nevoie sa se transfere date la o a doua resursa, se recomanda utilizarea RequestDispatcher. Totusi, exista tehnici, dar ineficiente, pentru a transfera date asociate cu cererea initiala, la resursa catre care se efectueaza redirectarea. Prima optiune este salvarea datelor ca parametri *queryString* care pot fi populati dinamic cu valori de pe *ValueStack*, prin OGNL. A doua optiune este salvarea datelor in map-ul asociat cu sesiunea curenta (dezavantaj : functioneaza numai cand a doua resursa apartine aceleiasi aplicatii web).

- ServletActionRedirectResult (*redirectAction*) => Realizeaza acelasi lucru ca si operatia de redirectare clasica, cu o diferenta : poate intelege numele logice ale actiunilor Struts2 utilizate. Nu trebuie utilizate URL-urile reale in declararea resurselor catre care se efectueaza redirectarea. Acest tip de rezultat are urmatoarele avantaje : permite declaratii mai robuste si invariante la anumite modificari ale URL-urilor, de exemplu la schimbarea extensiilor de la .action la .go ; ofera posibilitatea ca parametrii cererii initiale sa fie transmisi actiunii catre care se efectueaza redirectarea, prin tag-ul *param* (se evita astfel utilizarea manuala a *queryString*-urilor).

Exista si tipuri de rezultate care suporta tehnologii alternative pe partea de view : VelocityResult (*velocity* - pentru *Velocity*), FreemarkerResult (*freemarker* - pentru *FreeMarker*). Rezultatele pot fi configurate in doua moduri : local (pentru fiecare actiune) sau global (pentru toate actiunile pachetului in care a fost configurat rezultatul). Un rezultat definit local va suprascrie un alt rezultat definit global. Se cauta un rezultat global numai cand framework-ul nu gaseste printre rezultatele locale, unul care sa corespunda String-ului returnat de metoda *execute* a actiunii curente. [2]

Tag-ul folosit pentru partea de view este *result*, care are mai multi parametri : *type* (care are ca valoare implicita *dispatcher*), *name* (are ca valoare implicita SUCCESS). [2]

3.2.1.4.3 Modul de lucru al framework-ului

Framework-ul are mai mult decat componentele MVC prezentate anterior. Struts2 ofera o implementare mai curata a MVC, cu ajutorul unor componente arhitecturale care participa la procesarea fiecarei cereri : interceptori, OGNL, *ValueStack*. [2]

3.2.1.4.3.1 Modul de procesare a unei cereri

Modul de procesare a unei cereri, utilizand aceste elemente, este redat in figura de mai jos [2] :

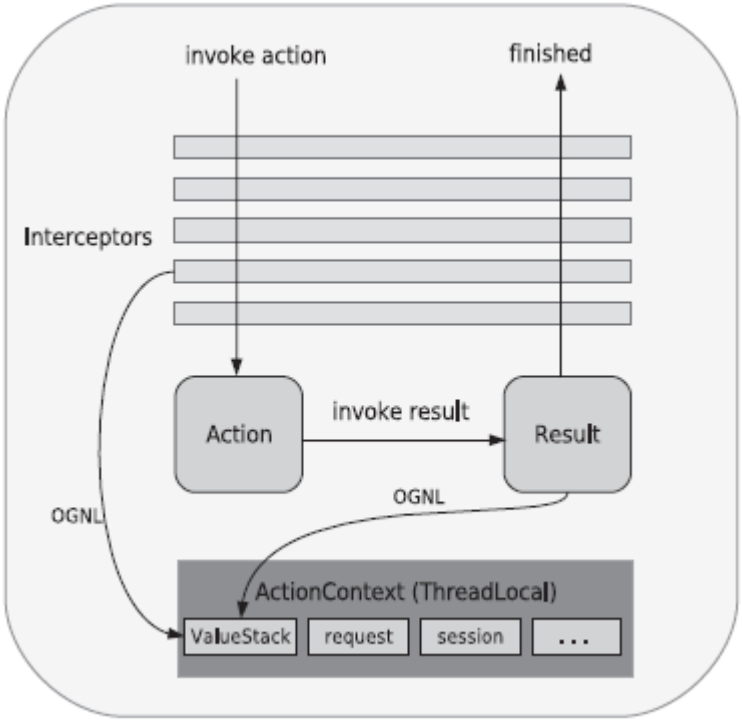


Figura 3.5 : Modul de procesare a unei cereri in Struts2

Figura de mai sus respecta principiile design-ului arhitectural MVC. Aceasta arata ce se intampla in momentul in care o actiune este invocata de controller, dupa ce aceasta a fost selectata. Mai jos sunt explicate toate elementele componente ale acestei scheme.

3.2.1.4.3.2 Interceptori

Interceptorii sunt componente soft care executa anumite task-uri comune unui anumit set de actiuni. Fiecare actiune are o stiva asociata de interceptori. Acestia sunt invocati atat inainte cat si dupa executia actiunii. De remarcat este faptul ca interceptorii permit task-urilor comune mai multor actiuni sa fie definite in componente reutilizabile, separate de codul care implementeaza task-ul actiunilor. Unele task-uri implementate de interceptori sunt intelese ca task-uri de preprocesare (efectuate inaintea executiei actiunii), iar altele de postprocesare (efectuate dupa executia actiunii). Exemple de task-uri de preprocesare ar fi transferul de date, de la parametrii cererii la proprietatile specifice unei actiuni (transfer realizat de catre interceptorul params), validari asupra datelor si conversii de tipuri. [2]

Asadar, in loc sa existe un controller simplu care invoca o actiune direct, au fost create componente care stau intre controller si actiuni. In Struts2, nici o actiune nu este invocata solitar. Procesul de invocare a unei actiuni include executia unei stive de interceptori inainte si dupa executia propriu-zisa a actiunii curente. In loc sa se invoce direct metoda *execute* a unei actiuni, framework-ul creeaza un obiect numit *ActionInvocation* care incapsuleaza actiunea si toti interceptorii configurati sa ruleze inainte si dupa executia acesteia, precum si rezultatele returnate de actiune. [2]

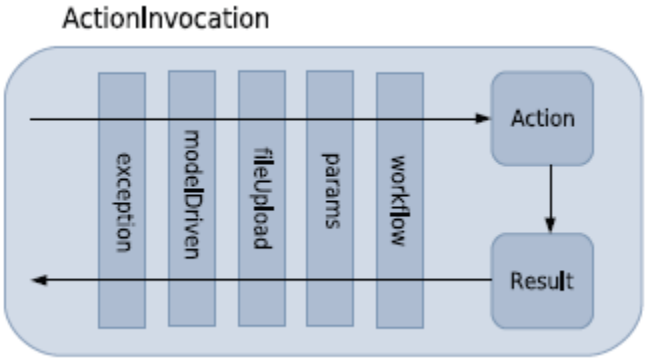


Figura 3.6 : Structura obiectului ActionInvocation

Conform figurii de mai sus, invocarea unei actiuni trebuie sa parcurga mai intai stiva de interceptori asociata cu aceasta. In schema de mai sus e prezentata o versiune simplificata a *defaultStack*, care include task-uri precum uploading de fisiere si transfer de parametri corespunzatori cererii curente catre actiune. Apoi, actiunea se va executa si va returna un String de control cu ajutorul caruia se va selecta rezultatul corespunzator. Dupa executia rezultatului, fiecare interceptor asociat actiunii, in ordine inversa, are sansa de a efectua task-uri de postprocesare. Interceptorii au acces la actiune si la valorile asociate cu aceasta. Unul dintre aspectele functionale puternice ale interceptorilor este capacitatea acestora de a altera fluxul invocarii unei actiuni. Uneori, interceptorii pot determina ca actiunea curenta sa nu se execute, intrerupand fluxul asociat cu invocarea acesteia si returnand un String de control (de exemplu, interceptorul *workflow* face doua lucruri : in primul rand invoca metoda *validate* a actiunii curente, in cazul in care aceasta a implementat interfata *Validateable* ; apoi verifica prezenta mesajelor de eroare asociate cu actiunea curenta ; daca exista erori, returneaza un String de control si intrerupe fluxul de executie ; deci actiunea nu va mai fi invocata si nici macar urmatorul interceptor din stiva nu va mai fi executat ; prin returnarea unui String de control, interceptorul curent preda controlul fluxului de executie interceptorilor anteriori, care au ocazia sa execute task-urile de postprocesare ; in final, rezultatul care se va potrivi cu String-ul de control returnat va fi afisat ; in cazul in care interceptorul *workflow* a gasit mesaje de erori asociate cu actiunea curenta, String-ul de control va fi *input* si se va mapa pe pagina anterioara, pe care s-au introdus date invalide). [2]

Cateva din beneficiile interceptorilor sunt : reutilizarea codului, a software-ului (logica ce se vrea a fi reutilizata poate fi separata, izolata intr-un interceptor si in acest mod poate fi aplicata tuturor claselor ; singurul lucru care trebuie facut pentru a reutiliza interceptorii este mostenirea *defaultStack*, care se afla in pachetul *struts-default*), configurarea ordinii si numarului interceptorilor (*defaultStack* ofera un set de interceptori aranjati intr-o ordine anume, pentru a satisface nevoile functionale comune majoritatii cererilor ; totusi, developerii pot rearanja setul de interceptori corespunzator nevoilor lor, pot sterge sau include interceptori, pot crea interceptori noi pe care sa ii includa in setul de interceptori pusi la dispozitie de *defaultStack*). [2]

Procesarea unei cereri, in Struts2, are mai multe etape. *ActionInvocation* incapsuleaza toate detaliile de procesare asociate cu executia unei actiuni particulare. Cand framework-ul primeste o cerere, decide mai intai pe ce actiune se mapeaza URL-ul cererii. O instanta a unei actiuni (tocmai create) este adaugata unei noi instante a *ActionInvocation*, care se creeaza tot atunci. Apoi framework-ul consulta arhitectura declarativa a aplicatiei (definita prin fisiere XML sau adnotari Java), pentru a vedea ce interceptori trebuie executati si in ce ordine.

Referintele acestor interceptori sunt adaugate obiectului *ActionInvocation*, care contine si alte referinte, printre care setul de rezultate disponibile actiunii. Dupa ce *ActionInvocation* a fost creat si populat cu toata informatia de care are nevoie, poate incepe procesul de invocare a actiunii. *ActionInvocation* contine metoda *invoke*, care e apelata de framework pentru a incepe executia actiunii curente. In momentul apelului, *ActionInvocation* incepe procesul de invocare executand primul interceptor de pe stiva. Metoda *invoke* nu se mapeaza intotdeauna pe primul interceptor de pe stiva. Este responsabilitatea *ActionInvocation* sa tina evidenta procesului de invocare si sa predea controlul executiei urmatorului interceptor de pe stiva. *ActionInvocation* realizeaza acest lucru utlizand metoda *intercept* asociata cu interceptorul respectiv. Executia interceptorilor urmatori si, in final, a actiunii, se realizeaza prin apeluri recursive ale metodei *invoke* corespunzatoare *ActionInvocation*. De fiecare data cand are loc un apel *invoke*, *ActionInvocation* isi consulta starea si executa urmatorul interceptor de pe stiva. Cand toti interceptorii sunt invocati, metoda *invoke* va determina executia actiunii propriu-zise. De remarcat ca metoda *intercept* asociata cu fiecare interceptor ia ca parametru o instanta *ActionInvocation*. In timpul procesarii efectuate, interceptorul va apela metoda *invoke* in conjunctie cu acest obiect, pentru a continua procesul recursiv de apel al interceptorilor. [2]

Un interceptor are asociate trei etape, in ciclul sau de executie [2] :

- Efectueaza preprocesari, deci poate fi utilizat pentru a pregati, filtra, altera sau manipula datele pe care le poate accesa (inclusiv actiunea care apartine cererii curente).

- Apeleaza *invoke* sau opreste fluxul de invocare a a actiunilor (daca un interceptor stabileste ca procesarea unei cereri nu trebuie sa continue, poate returna un String de control, in loc sa apeleze metoda *invoke* in conjunctie cu instanta *ActionInvocation*. In acest fel, poate opri executia si poate determina el insusi rezultatul care va fi afisat).

- Chiar dupa apelul metodei *invoke* si dupa ce aceasta returneaza un String de control, oricare dintre interceptorii care asteapta un rezultat al acesteia pot decide sa modifice obiectele si datele la care are acces, ca parte a operatiei de postprocesare. Atentie, in acest moment rezultatul este deja afisat (cu alte cuvinte, pagina de raspuns a fost trimisa deja clientului), deci chiar daca interceptorul poate implementa o anumita logica de postprocesare, utilizand String-ul returnat de metoda *invoke*, el nu mai poate opri sau modifica raspunsul dat clientului.

Mai jos este dat un exemplu care prezinta metoda *intercept* a interceptorului *TimeInterceptor*, inclus in pachetul *struts-default*, pentru a ilustra mai bine cele trei etape din ciclul de executie al unui interceptor [2] :

```
public String intercept(ActionInvocation invocation) throws Exception {
   long startTime = System.currentTimeMillis();      (1)
   String result = invocation.invoke();      (2)
   long executionTime = System.currentTimeMillis() - startTime;
      ... log the time ...                                          (3)
   return result;
}
```

Exista mai multe categorii de interceptori, in functie de task-ul pe care il executa [2] :

- Interceptori de utilitate : *Timer* => inregistreaza durata unei executii.

- Interceptori pentru transferul datelor : *params* => transfera parametrii cererii la proprietatile asociate din actiunea curenta, care se afla pe *ValueStack* ; *static -params* => transfera parametrii actiunii, definiti in fisierul XML, la specificarea actiunii, cu tag-ul *param*, la proprietatile corespunzatoare acesteia, care se afla pe *ValueStack* ; *servlet-config* => ofera o modalitate de a injecta diferite obiecte din API-ul Servlet, in actiunile utilizate ; *fileUpload* => ofera facilitati de uploading.

- Interceptori pentru controlul fluxului invocarii unei actiuni : *workflow*, *validation* => lucreaza impreuna cu *workflow* pentru a asigura validarea datelor.

- Alte tipuri de interceptori : *exception* => ar trebui declarat primul in orice stiva costum definita de utilizator ; prinde exceptiile si le mapeaza, conform tipurilor, paginilor de eroare definite ; pozitia sa in topul stivei garanteaza faptul ca va fi capabil sa prinda toate exceptiile care pot fi generate in timpul tuturor fazelor de invocare a actiunii curente, deoarece este ultimul care are sansa de a executa task-uri de postprocesare.

Se pot asocia stive costum de interceptori unui set de actiuni, se pot seta si suprascrie parametri pentru interceptori, se pot defini interceptori proprii, care sa fie integrati in framework alaturi de ceilalti interceptori pusi la dispozitie de acesta. [2]

3.2.1.4.3.3 ActionContext, ValueStack si OGNL

Cand framework-ul Struts2 primeste o cerere, acesta creeaza imediat un *ActionContext*, o *ValueStack* si un obiect corespunzator actiunii curente, care este plasat pe *ValueStack*. [2]

ActionContext reprezinta contextul pentru executia unei actiuni, adica un container simplu pentru toate datele si resursele care caracterizeaza executia unei actiuni (atat *ValueStack* cat si obiectele pe care framework-ul le foloseste intern, precum *request*, *session*, *application*, *parameters* – mostenite din API-ul Servlet). *Parameters* reprezinta setul parametrilor asociati cu cererea curenta (cei trecuti in form-ul de intrare), *request* este setul atributelor asociate cu domeniul de vizibilitate al cererii curente, *session* este setul de atribute asociate cu sesiunea curenta, *application* reprezinta setul atributelor asociate cu aplicatia curenta (sunt accesibile tuturor cererilor care vin catre aplicatia curenta) iar *attr* returneaza prima aparitie a unui atribut

in ordinea dimensiunii scopurilor (*page*, *request*, *session*, *application*). Structura *ActionContext* este prezentata mai jos [2] :

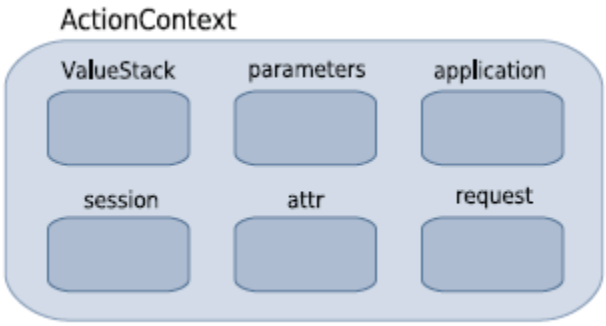


Figura 3.7 : Structura ActionContext in Struts2

Struts2 utilizeaza *ValueStack* ca pe o locatie in care stocheaza toate datele de care va fi nevoie in timpul procesarii unei cereri. Datele sunt mutate pe *ValueStack* pentru a fi pregatite pentru procesarea cererii curente, sunt manipulate acolo in timpul executiei actiunii corespunzatoare si sunt citite de acolo cand rezultatele afiseaza paginile de raspuns trimise clientului. Pe *ValueStack* nu sunt stocate si alte tipuri de obiecte specifice framework-ului Struts2 si specifice API-ului Servlet. Din moment ce Struts2 proceseaza fiecare cerere intr-un nou thread, *ValueStack* este accesibila pe toata durata cererii respective (la fel si *ActionContext*). Cu toare acestea, este considerata o practica rea obtinerea continutului *ActionContext* in mod direct. Framework-ul ofera multe modalitati elegante de a accesa date, fara a utiliza, in mod explicit, *ActionContext* sau *ValueStack* (prin OGNL). [2]

OGNL este o tehnologie integrata in Struts2 care are doua mari facilitati [2] :

- In primul rand, ofera un limbaj de expresii utilizat pentru a simplifica accesul la datele de pe *ValueStack* si din *ActionContext*. Pune la dispozitie o sintaxa simpla pentru a lega tag-urile Struts2 de proprietatile Java asociate, pentru a transfera datele in interiorul si in afara framework-ului. Deci OGNL ajuta la mutarea datelor corespunzatoare parametrilor cererii la proprietatile JabaBeans asociate, din actiuni, dar si la transferul datelor de la proprietatile JavaBeans ale actiunilor, care se afla pe *ValueStack*, la paginile HTML care sunt afisate clientului, ca raspuns. *ValueStack* este vazut ca un obiect virtual atunci cand e accesat prin OGNL. Acest obiect virtual contine toate proprietatile obiectelor pe care le cuprinde. Daca o proprietate apare de mai multe ori, va fi aleasa cea care apare cel mai sus in stiva, iar copiile acesteia, situate mai jos, sunt ascunse. Privind la schema care prezinta modul de functionare a framework-ului, observam ca atat interceptorii cat si rezultatele pot accesa valori de pe *ValueStack*, utilizand OGNL. Limajul de expresii OGNL ofera o gama larga de constructii, de la cele mai simple (care reprezinta numele parametrilor din formele de intrare), pana la cele mai complicate (precum cele utilizate la accesarea colectiilor Java sau cele care reprezinta literali, operatori sau ofera modalitati de a apela metode Java sau de a accesa campuri ale claselor Java). Se pot utiliza expresii OGNL pentru a accesa orice set de obiecte, nu numai cele de pe *ValueStack*. Astfel, pot fi accesate orice obiecte din *ActionContext*. OGNL trebuie sa aleaga unul din obiectele din *ActionContext*, pentru a-l utiliza ca obiect-radacina si acesta este, implicit, *ValueStack*. Datorita *intelligent defaults*, acesta nu va trebui specificat explicit cand

se vor accesa obiecte de pe el, utilizand expresii OGNL. In cazul in care se doreste accesarea altor obiecte si nu a celor de pe *ValueStack*, expresiile OGNL trebuie sa inceapa cu o sintaxa speciala care specifica obiectul care va fi accesat, considerat ca radacina (sintaxa #session[‘user’] returneaza obiectul user care este stocat in obiectul care reprezinta sesiunea curenta).

- Convertorii de tipuri OGNL sunt utilizati pentru realizarea conversiilor de la formatul String al HTML la tipurile de date Java si invers. Framework-ul Struts2 ofera convertori gata construiti pentru toate tipurile uzuale Java (String, boolean / Boolean, char / Character, int / Integer, float / Float, long / Long, double / Double, Date, array, List, Map).

Asadar OGNL indeplineste aceleasi roluri functionale la ambele capete ale procesarii cererii : limbajul sau de expresii cauta proprietatea asociata cu numele campului formei de intrare iar convertorii de tip asigura translatarea de la tipul String corespunzator HTML, la tipul Java asociat. La fel se intampla si cand se doreste afisarea unei proprietati de pe *ValueStack*, doar ca de aceasta data convertorii asigura translatarea de la tipul Java asociat proprietatii la tipul String corespunzator paginii HTML care va fi afisata. [2]

3.2.1.4.3.4 Fisiere de configurare

Exista doua fisiere de configurare de baza in framework-ul Struts2 [2] :

- Fisierul central de configurare, pentru toate aplicatiile web, este *web.xml* (deployment descriptor). Acesta contine definitii pentru toate servelt-urile, filtrele-servlet si alte componente ale API-ului Servlet din aplicatia web curenta. In exemplul de mai jos, cele mai importante elemente ale fisierului sunt *filter* si *filter-mapping*, care seteaza Filter-Dispatcher-ul Struts2 (care va examina toate cererile care vin de la clienti, caci URL-ul lui este “/*”). Parametrul de initializare *actionPackages* este necesar daca in aplicatia web se utilizeaza adnotari :

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
```

```
 http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

<display-name>S2 Example Application - Chapter 1 - Hello World
 </display-name>

<filter>
  <filter-name>struts2</filter-name>
  <filter-class>org.apache.struts2.dispatcher.FilterDispatcher
   </filter-class>
  <init-param>
     <param-name>actionPackages</param-name>
     <param-value>manning</param-value>
  </init-param>
</filter>

<filter-mapping>
   <filter-name>struts2</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>

<servlet>
   <servlet-name>anotherServlet</servlet-name>
   <servlet-class>manning.servlet.AnotherServlet</servlet-class>
</servlet>

<servlet-mapping>
 <servlet-name>anotherServlet</servlet-name>
 <url-pattern>/anotherServlet</url-pattern>
</servlet-mapping>

<welcome-file-list>
   <welcome-file>index.html</welcome-file>
</welcome-file-list>

</web-app>
```

FilterDispatcher: Struts 2 begins here

Tell Struts where to find annotations

A servlet outside of Struts

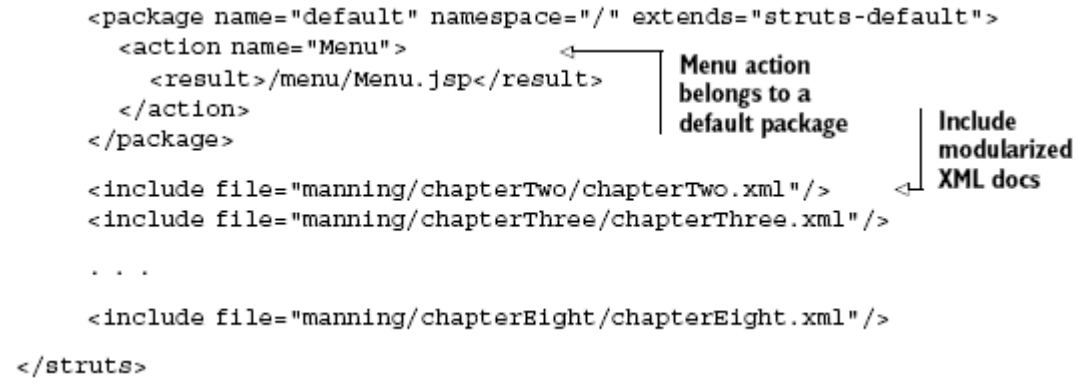
- Fisierul de intrare pentru declararea arhitecturii unei aplicatii web, in Struts2, este *struts.xml*. Multi developeri utilizeaza acest document radacina pentru a include alte documente XML (fiecare din aceste documente declara o parte din arhitectura aplicatiei), pentru a realiza modularizarea acestora. Tot in cadrul fisierului *struts.xml* se pot defini si actiuni globale, in pachetul "*default*". Un exemplu de astfel de fisier este dat mai jos :

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
   "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
   "http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>

  <constant name="struts.devMode" value="true" />
```

1 Use constants to tweak Struts properties




```
<package name="default" namespace="/" extends="struts-default">
  <action name="Menu">
    <result>/menu/Menu.jsp</result>
  </action>
</package>

<include file="manning/chapterTwo/chapterTwo.xml"/>
<include file="manning/chapterThree/chapterThree.xml"/>

. . .

<include file="manning/chapterEight/chapterEight.xml"/>

</struts>
```

3.2.1.4.4 Comparatie intre Struts1 si Struts2

Exista mai multe aspecte prin care Struts2 difera radical de Struts1. Acestea sunt ilustrate in tabelul de mai jos [8] :

Tabel 3.1

Struts1	Struts2
Toate clasele care reprezinta actiuni trebuie sa extinda clasa abstracta *Action*. E incurajata programarea utilizand clase abstracte in locul interfetelor.	O actiune Struts2 poate implementa interfata *Action* sau alte interfete, pentru a beneficia de anumite facilitati puse la dispozitie de framework. Mai mult, Struts2 pune la dispozitie si o clasa *ActionSupport* care implementeaza interfetele cele mai des utilizate. Cu toate acestea, nu este obligatorie implementarea unei interfete sau extinderea unei clase, de catre o actiune Struts2.
Actiunile Struts1 sunt singleton-uri. O singura instanta a unei clase va gestiona toate cererile pentru actiunea corespunzatoare. Aceasta strategie impune restrictii asupra lucrurilor care pot fi realizate cu actiunile Struts1. In acelasi timp, resursele actiunilor trebuie sa fie thread-safe sau sincronizate.	Se creeaza o noua instanta a unei actiuni pentru fiecare cerere efectuata, deci nu trebuie asigurata vreo conditite de sincronizare. De remarcat este ca generarea de obiecte noi, pentru fiecare cerere, nu incetineste performantele framework-ului si nu ingreuneaza task-urile garbage collector-ului.
Actiunile Struts1 au dependinte fata de Servlet API, din moment ce obiectele de tip *HttpServletRequest* si *HttpServletResponse* sunt transmise ca parametri metodei *execute*, cand actiunea este invocata. Prin urmare, testarea actiunilor, in Struts1, e mai dificila decat in Struts2, deoarece actiunile Struts1 depind de Servlet API.	Actiunile Struts2 nu sunt legate de un container. Ele pot accesa obiectele *request* si *response*, daca acest lucru se cere. Totusi, celelalte elemente arhitecturale ale framework-ului reduc sau elimina nevoia de a accesa direct obiectele *HttpServletRequest* si *HttpServletResponse*. Testarea actiunilor Struts2 este mai facila, deoarece acestea nu au dependinte fata de Servlet API.
Struts1 foloseste un obiect de tip *ActionForm* pentru a prelua datele de intrare introduse de utilizator. Ca toate actiunile, clasele	Struts2 foloseste proprietatile actiunilor pentru a prelua datele de intrare, eliminand nevoia unui al doilea obiect care sa le gestioneze.

ActionForms trebuie sa extinda o clasa de baza. Din moment ce alte clase JavaBeans nu pot fi utilizate ca *ActionForms*, developerii sunt nevoiti sa defineasca clase redundante pentru a prelua datele de intrare.	
Struts1 utilizeaza JSTL si JSTL-EL, care ofera functionalitati de baza pentru accesarea obiectelor Java, dar destul de slabe in cazul manipulari colectiilor.	Struts2 poate utiliza JSTL, dar suporta si un limbaj de expresii mai puternic si mai flexibil, OGNL.
Struts1 foloseste mecanismul standard JSP pentru a lega obiectele Java de contextul paginii care trebuie afisata, pentru a fi accesate.	Struts2 foloseste componenta *ValueStack*, astfel incat taglib-urile pot accesa valori fara a cupla partea de view la obiectele pe care le afiseaza. Se permite astfel reutilizarea view-urilor in conjunctie cu mai multe actiuni care pot avea aceleasi nume de proprietati, dar tipuri diferite.
Proprietatile *ActionForm*-ului Struts1 sunt, de obicei, String-uri. Struts1 utilizeaza mecanisme Commons-BeanUtils pentru realizarea conversiei tipurilor. Convertorii Struts1 sunt configurabili per clasa, si nu per instanta.	Struts2 utilizeaza OGNL pentru conversia tipurilor. Framework-ul include convertori automati pentru tipurile primitive si clasele Wrapper asociate acestora, dar si pentru alte cateva tipuri de baza.
Struts1 ofera mecanisme primitive de validare a datelor.	Struts2 suporta mecanisme de validare mai eficiente si mai puternice decat Struts1, caci permite validarea inlantuita, pentru subproprietati ale obiectelor.
Struts1 nu ofera o implementare curata a MVC-ului. Nu exista mecanisme de separare a componentelor comune mai multor actiuni de operatiile specifice unei singure actiuni.	Struts2 utilizeaza interceptori, elemente arhitecturale care nu exista in Struts1, oferind in acest fel o implementare mai curata a MVC-ului (separa logica specifica unei actiuni concrete de logicile aditionale – specifice mai multor actiuni).

3.2.1.4.5 Avantajele si dezavantajele framework-ului

Din prezentarea framework-ului Struts2 se pot deduce usor avantajele pe care acesta le ofera [2] :

- Permite modularizarea componentelor aplicatiei, care implementeaza intr-un mod mai curat design pattern-ul MVC. Componentele sunt independente una de alta dar, in acelasi timp, comunica bine intre ele, fiind reutilizabile si in alte aplicatii.

- Design-ul aplicatiilor este imbunatatit deoarece toate clasele Struts2 se bazeaza pe implementarea interfetelor.

- Prin componentele arhitecturale de care dispune (*ValueStack*, OGNL, interceptori), Struts2 pune la dispozitie mecanisme extrem de puternice si facile de legare a

parametrilor cererii de tipurile de date Java (transmiterea si conversia automata a acestora) ale proprietatilor actiunilor, dar si de transmitere a acestora in paginile de raspuns afisate clientului.

- Ofera diverse mecanisme (de baza sau complexe) pentru validarea datelor si pentru internationalizarea aplicatiilor.

- Permite realizarea unor interfete user-friendly si eficiente, deoarece ofera suport pentru XHTML, CSS, AJAX si permite integrarea de plugin-uri pentru decorarea paginilor (*Tiles, SiteMesh, JFreeChart*).

- Faciliteaza integrarea de noi plugin-uri (prin adaugarea de JAR-uri in aplicatie, fara sa fie necesare alte configurari manuale).

- Permite integrarea Spring, JPA si JSF => se poate utiliza impreuna cu alte framework-uri si tehnologii, pentru crearea de aplicatii web puternice.

- Aplicatiile pot fi testate usor, mai ales ca actiunile nu sunt dependente de Servlet API.

Dezavantajele framework-ului Struts2 :

- Complexitatea arhitecturii, de unde rezulta ca este nevoie de o perioada mai mare de invatare si acomodare, atat cu functionalitatea componentelor framework-ului si comunicarea dintre ele, cat si cu alte tehnologii pe care acesta le poate integra automat.

3.2.1.4.6 Adaugarea Spring la Struts2

S-a ales alternativa adaugarii Spring la Struts2 pentru a profita de tehnica utilizata de acest framework in gestionarea crearii obiectelor. In loc de a scrie cod pentru a crea obiecte sau pentru a acapara anumite resurse, se va folosi metadata pentru a declara dependintele pe care un anumit obiect le are. Containerul Spring citeste metadata, localizata de obicei in fisierul *applicationContext.xml*, pentru a invata dependintele obiectelor pe care le gestioneaza. Spring ofera cateva metode de a injecta aceste dependinte in obiectele care le utilizeaza. O metoda foarte utilizata este *autowiring*, prin care obiectul curent expune o metoda setter pentru a prelua un alt obiect pe care-l utilizeaza (obiect declarat in fisierul *applicationContext.xml* si injectat de catre containerul Spring, prin metoda setter corespunzatoare). Astfel se elimina cuplarea de grad inalt dintre obiecte, introdusa prin crearea acestora utilizand operatorul *new*. Avantajele acestei tehnici de creare a obiectelor sunt vizibile mai ales la testare. Clasele si obiectele care sunt cuplate putin pot fi testate mai usor decat cele care sunt mai strans cuplate. [2]

3.2.2 Sisteme software pentru partea de client

3.2.2.1 Design-ul paginilor web cu ajutorul CSS

Browser-ele web sunt aplicatii software care permit utilizatorilor sa afiseze text, imagini sau alte tipuri de informatii, localizate de obicei in pagini web stocate pe servere sau in reteaua

locala. Textele si imaginile din paginile web pot contine legaturi catre alte pagini web. De asemenea, utilizatorul poate sa interactioneze cu aceste pagini prin intermediul browser-elor, trimitand inapoi la server anumite informatii. Browser-ele web sunt cele mai adesea folosite ca si clienti HTTP. Desi ele sunt utilizate cu predilectie pentru accesarea world wide web-ului, pot fi de asemenea folosite si pentru accesarea informatiilor din retelele locale sau a sistemului de fisiere. [10]

CSS (Cascading Style Sheets) este un standard dezvoltat pentru a controla modul de afisare a elementelor dintr-o pagina web. Ofera suport pentru toate aspectele de vizualizare ale unei pagini si permite definirea modului in care vor fi afisate diverse elemente HTML, precum si definirea de selectori si clase pentru elemente. CSS ofera un nivel mai inalt pentru control, decat stilul mai vechi care presupunea inserarea elementelor de formatare in codul paginilor web. Un alt avantaj al utilizarii CSS este acela ca centralizeaza definitiile selectorilor claselor si a altor elemente de formatare, micsorand dimensiunea paginilor web, prin eliminarea codului de formatare. Fisierul CSS va fi transferat o singura data pentru un client, iar apoi toate celelalte pagini care-l folosesc il au disponibil in *cache*. Astfel se reduce traficul in retea, la vizualizarea unor pagini de pe server. In plus, datorita centralizarii elementelor de formatare, este mult mai usor de intretinut si modificat design-ul unui site. Modificand intr-un singur loc, obtinem schimbari in intreg site-ul. [6]

3.2.3 Sisteme software pentru partea de persistenta a datelor

3.2.3.1. Baze de date relationale si tehnologia MySQL

Un model de date este un ansamblu de reguli pentru organizarea datelor, impreuna cu un set de operatii permise asupra acestor date. O baza de date este o colectie de date organizate intr-o structura descrisa printr-un model conceptual. O baza de date relationala este compusa dintr-o multime finita de relatii, fiecare relatie reprezentand un tip de entitate sau o asociere dintre doua sau mai multe tipuri de entitati. [4]

Sistemul de gestiune al bazelor de date (SGBD) este intregul ansamblu software care trateaza toate cererile de acces la baza de date. SGBD cuprinde doua facilitati importante pentru proiectarea si exploatarea bazelor de date : facilitati de descriere a datelor (sunt materializate, in esenta, prin limbajul de descriere a datelor – specifica structurile de date permise in cadrul unui model de date ; structurile permise pot fi specificate in doua moduri complementare : prin precizarea obiectelor permise si a relatiilor permise dintre ele folosind reguli generice de definire, prin specificarea obiectelor si relatiilor nepermise care sunt exclude prin definirea unor restrictii numite constrangeri), facilitati de manipulare a datelor (se refera, in principal, la limbajele de manipulare a datelor, care constituie interfata dintre SGBD si utilizatori ; acestea permit operatii de interogare sau modificare a bazelor de date). [4]

MySQL este un sistem de gestiune al bazelor de date relational, produs de compania suedeza MySQL AB si disponibil sub licenta GNU General Public License (GPL). Este cel mai popular SGBD open-source la ora actuala, fiind o componenta cheie a stivei LAMP (Linux, Apache, MySQL, PHP). Desi este folosit foarte des impreuna cu limbajul de programare PHP, cu MySQL se pot construi aplicatii in orice limbaj. [11]

3.2.3.2 JPA

JPA (Java Persistence API) este un API pus la dispozitie de limbajul Java, care le permite developerilor sa acceseze si sa managerieze baze de date relationale, in aplicatiile scrise de acestia, utilizand una din platformele : Java 2 Platform, Standard Edition ; Java 2 Platform, Enterprise Edition. [3]

Exista trei mari concepte, cand vorbim despre JPA, si anume : API-ul specific (definit in pachetul javax.persistence), JPQL (Java Persistence Query Language) si medatata obiect / data relationala. [3]

O entitate de persistenta este o clasa Java care se mapeaza pe un tabel din baza de date relationala. Instantele entitatilor corespund liniilor individuale dintr-un anumit tabel. Entitatile pot avea relatii cu alte entitati, iar relatiile dintre ele sunt ilustrate prin metadata obiect / data relationala. Metadata poate fi specificata intr-un fisier XML separat, sau direct in fisierul sursa Java, utilizand adnotari. [3]

JPQL este un limbaj utilizat pentru a efectua interogari si modificari pe entitatile de persistenta care se mapeaza pe tabelele din baza de date relationala. JPQL opereaza pe instante ale entitatilor si nu pe tabelele definite in baza de date relationala. [3]

4. Analiza si proiectare

4.1 Arhitectura sistemului

Arhitectura modulului de comunicare este o arhitectura pe trei nivele, specifica aplicatiilor web moderne. O aplicatie web pe trei nivele include un browser web pe primul nivel, un server web pe al doilea nivel si o baza de date pe ultimul nivel. In figura de mai jos este ilustrat acest tip de arhitectura [14] :

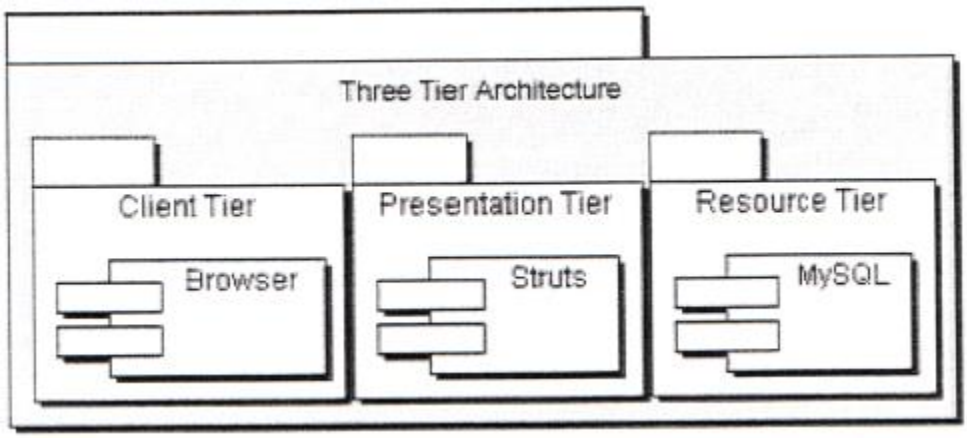


Figura 4.1 : Arhitectura pe trei nivele a unui sistem distribuit

Avantajul principal al acestei arhitecturi este faptul ca permite un design flexibil, prin care se separa logica aplicatiei de datele acesteia si de modul de afisare a informatiilor. Daca se va dori, ulterior, sa se modifice modul in care este implementat un nivel al arhitecturii, celelalte nivele nu vor fi afectate de aceste modificari. De exemplu, se poate schimba tehnologia utilizata pentru partea de persistenta a datelor, fara sa se efectueze modificari in nivelul care contine logica aplicatiei. Totodata, dezvoltarea aplicatiilor pe trei nivele este avantajoasa, deoarece permite mai multor dezvoltatori specializati pe tehnologii diferite sa lucreze in paralel la cele trei nivele, fara sa utilizeze aceleasi surse si fara sa fie nevoie sa cunoasca tehnologiile utilizate la implementarea celorlalte nivele. [14]

Primul nivel al aplicatiei este reprezentat de browser-ul web care se afla pe masina clientului. Prin intermediul acestuia, clientul poate comunica, utilizand HTTP, cu server-ul care contine aplicatia propriu-zisa.

Cel de-al doilea nivel este constituit din componenta server a aplicatiei. Aplicatia este incarcata intr-un container de servlet-uri (Tomcat, de exemplu), de unde este accesata apoi de catre clientii web. Aplicatia este implementata utilizand framework-ul Struts2. Toate cererile primite de la clienti sunt preluate de catre un controller predefinit in Struts2 si se mapeaza pe anumite actiuni, care returneaza raspunsuri. Dupa ce se executa codul acestor actiuni, framework-ul incarca paginile JSP corespunzatoare raspunsurilor la cererile primite. Acestea vor fi translatate in servleti, vor fi pastrate pe server si vor fi afisate clientului.

Cel de-al treilea nivel este dat de baza de date din spatele aplicatiei, impreuna cu sistemul de gestiune corespunzator, utilizat pentru a efectua interogari si operatii de modificare asupra acesteia.

4.2 Arhitectura bazei de date din spatele aplicatiei

Sistemul de gestiune al bazelor de date utilizat de catre aplicatie este MySQL. In procesul de construire a tabelelor bazei de date, am utilizat tehnica entitate-atribute-relatie. Am identificat intai entitatile principale pentru care se vor stoca informatii in baza de date. Apoi am determinat relatiile dintre aceste entitati, pentru ca, in final, sa stabilesc atributele (proprietatile) fiecarei entitati in parte.

Baza de date asociata sistemului implementat se numeste *library*. Diagrama bazei de date este prezentata mai jos :

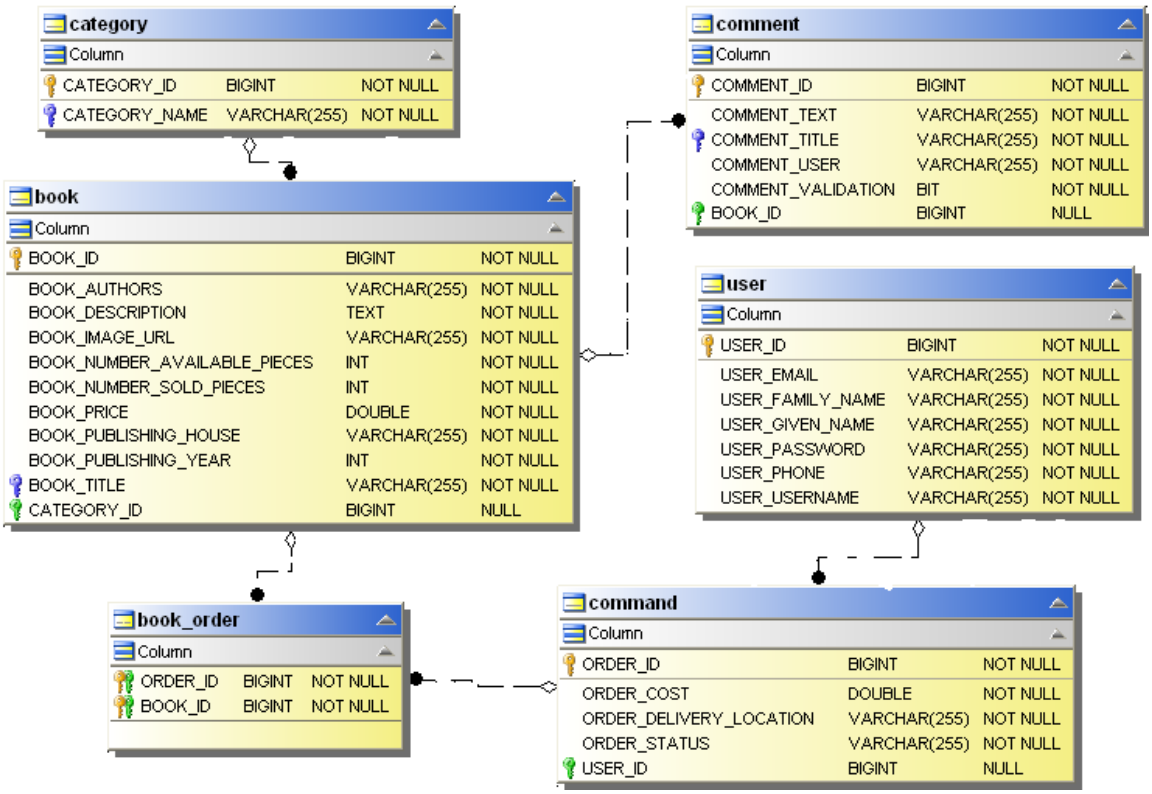


Figura 4.2 : Structura bazei de date a sistemului web

Categoriile de carti sunt pastrate in tabelul *category*. Fiecare categorie este unic identificata de un id (*category_id*), care este cheie primara in tabelul asociat, si are un nume (*category_name*) care este unic.

Cartile se regasesc in tabelul *book*. Fiecare carte este unic identificata de un id (*book_id*), care este cheie primara in tabelul asociat. Orice carte are un titlu (*book_title*) care este unic, unul sau mai multi autori (*book_authors*), o descriere (*book_description*), o poza (*book_image_url*) si un pret (*book_price*). O carte are asociat si numele editurii la care a fost publicata (*book_publishing_house*), precum si anul aparitiei pe piata (*book_publishing_year*). Pentru fiecare carte se cunosc numarul de bucati disponibile pentru vanzare

(*book_number_available_pieces*), numarul de bucati deja vandute (*book_number_sold_pieces*), dar si categoria din care face parte. Categoria unei carti este identificata, in tabelul *book*, printr-un id (*category_id*) care este cheie straina, propagandu-se din tabelul *category*.

Comentariile asociate cartilor sunt pastrate separat, in tabelul *comment*. Fiecare comentariu este unic identificat de un id (*comment_id*), care este cheie primara in tabelul asociat. Se cunosc atat continutul comentariului (*comment_text*), titlul acestuia (*comment_title*), cat si numele user-ului care l-a postat (*comment_user*). De asemenea, fiecare comentariu este asociat unei carti. Cartea e identificata, in tabelul *comment*, printr-un id (*book_id*), care este cheie straina, propagandu-se din tabelul *book*. Un comentariu nu poate fi vizualizat de catre ceilalti useri pana cand acesta nu este validat de administratorul sistemului, evitandu-se astfel postarea unor comentarii neadecvate. Pentru realizarea acestui lucru, s-a introdus si campul *comment_validation*.

Vizitatorii site-ului aplicatiei curente se pot inregistra in baza de date, in tabelul *user*, daca doresc sa efectueze comenzi on-line. Fiecare user este unic identificat de un id (*user_id*) si are un nume de familie (*user_family_name*), un prenume (*user_given_name*), un numar de telefon (*user_phone*) si o adresa de e-mail (*user_email*). Pentru a se loga, fiecare user are nevoie de un username (*user_username*) si de o parola (*user_password*). Username-ul si parola sunt unice, pentru fiecare user.

Comenzile efectuate sunt retinute in tabelul *command*. Fiecare comanda este unic identificata de un id (*order_id*). Se mai cunosc costul comenzii (*order_cost*), destinatia la care trebuie sa ajunga aceasta (*order_delivery_location*) si statusul comenzii (*order_status*). O comanda poate fi inregistrata (daca un user a efectuat-o online, nu a platit-o, dar nu a expirat timpul alocat platii), finalizata (daca a fost platita catre user in timp util) sau anulata (daca user-ul nu a efectuat plata pana la o anumita data). Pentru o anumita comanda, este necesar sa se cunoasca si persoana care a efectuat-o. Aceasta e identificata, in tabelul *command*, printr-un id al user-ului curent (*user_id*), care este cheie straina, propagandu-se din tabelul *user*.

Se observa faptul ca, pentru fiecare comanda, trebuie pastrat si continutul acesteia, si anume cartile comandate. O comanda poate contine mai multe carti, iar o carte poate apartine mai multor comenzi. Asadar, intre o entitate de tip *book* si o entitate de tip *command* exista o relatie de m la n. Relatia dintre cele doua entitati este modelata de tabelul *book_order*. Acesta contine o cheie primara dubla, formata din id-ul unei carti (*book_id*) si id-ul unei comenzi (*order_id*). Luate separat, acestea sunt chei straine propagate din tabelul *book*, respectiv din tabelul *command*. Fiecare linie din tabel modeleaza o relatie intre o entitate de tip *book* si o entitate de tip *order*, specificand faptul ca o carte cu id-ul *book_id* apartine unei comenzi identificate prin id-ul *order_id*.

Se vor sublinia, in continuare, cateva particularitati ale bazei de date prezentate anterior. Pentru a evita aparitia erorilor si confuziilor, nu se admite introducerea valorilor nule in baza de date. De asemenea, se remarca faptul ca nu pot exista doua categorii de carti cu acelasi nume , doua carti cu acelasi titlu sau doua comentarii cu acelasi titlu, in baza de date. Totodata, nu pot exista doi useri care sa aiba acelasi username sau aceeasi parola. Cheile primare, in tabelele *user*, *command*, *book*, *category* si *comment* sunt generate automat, de catre sistemul de gestiune al bazei de date. Campul *book_image_url*, din tabelul *book*, nu contine efectiv o imagine, ci un url catre o anumita poza.

5. Implementare

5.1 Cazuri de utilizare

Cazurile de utilizare reprezinta grafic functionalitatile pe care trebuie sa le indeplineasca un sistem informatic. Sunt formate din doua categorii de entitati (actori si cazuri de utilizare) si relatiile dintre acestea. Relatiile dintre doua cazuri de utilizare pot fi de *utilizare* (un caz de utilizare foloseste functionalitatea altui caz de utilizare) sau de *extindere* (un caz de utilizare extinde functionalitatea altui caz de utilizare). Relatiile dintre doi actori pot fi de *generalizare* (un actor poate interactiona cu sistemul in toate modurile in care interactioneaza un alt actor) sau de *dependenta* (pentru a interactiona cu sistemul, un actor depinde de un alt actor). Relatiile intre actori si cazuri de utilizare sunt de *asociere-comunicare* (bidirectionale). Cazurile de utilizare ajuta la crearea manualelor de utilizare si permit comunicarea cu persoane care nu au cunostinte de informatica. [15]

Sistemul descris are trei actori : un vizitator (poate vizualiza date de pe site), un user (poate vizualiza date de pe site si poate efectua comenzi – este, de fapt, un vizitator inregistrat in baza de date a sistemului si logat) si un administrator (poate vizualiza date de pe site si poate modifica anumite informatii). Mai jos sunt prezentate diagramele use-case corespunzatoare acestor actori :

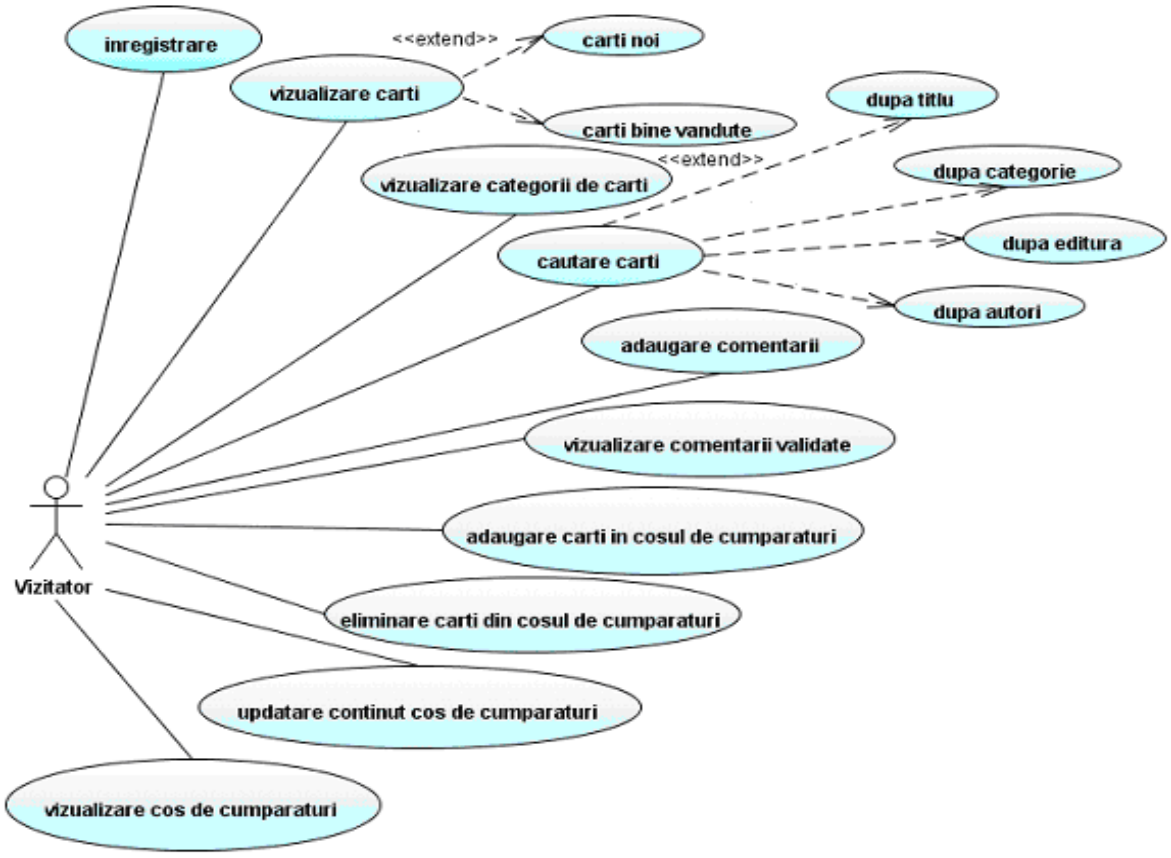


Figura 5.1 : Diagrama use-case a actorului Vizitator

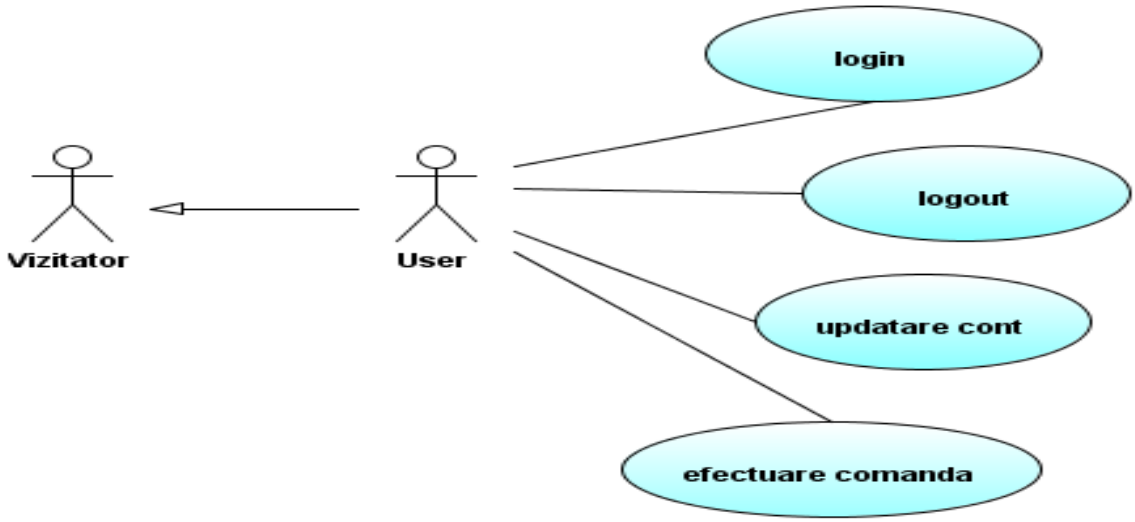


Figura 5.2 : Diagrama use-case a actorului User

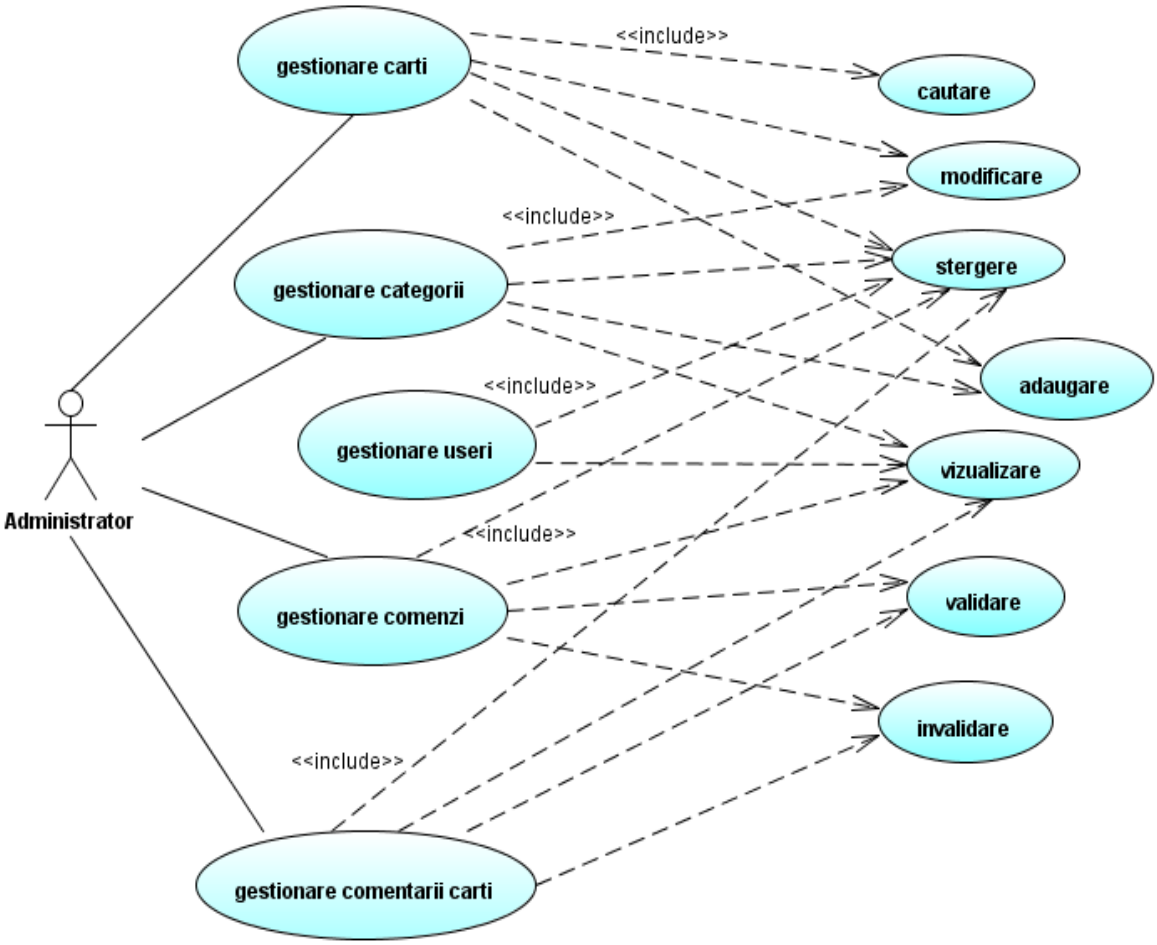


Figura 5.3 : Diagrama use-case a actorului Administrator

5.2 Implementarea sistemului

Pentru a implementa sistemul web, am utilizat framework-ul Struts2, limbajul Java si anumite tehnologii (CSS, JSP, JPA) si framework-uri (Spring) care pot fi integrate in Struts2. In urmatoarele pagini, se va prezenta modul in care au fost construite clasele si pachetele care implementeaza aplicatia web.

Fiind construit in Struts2, sistemul web implementeaza sablonul MVC. Aplicatia este structurata in trei componente de baza : modelul (retine datele aplicatiei, implementeaza partea de business si logica sistemului), controller-ul (mapeaza cererile clientilor pe actiunile corespunzatoare – este implementat automat de catre framework) si vizualizarile (JSP – uri care afiseaza rezultatele cererilor clientilor – constituie interfata grafica cu utilizatorul).

5.2.1 Modelul aplicatiei

Partea de model este constituita din pachetele prezentate in figura de mai jos :

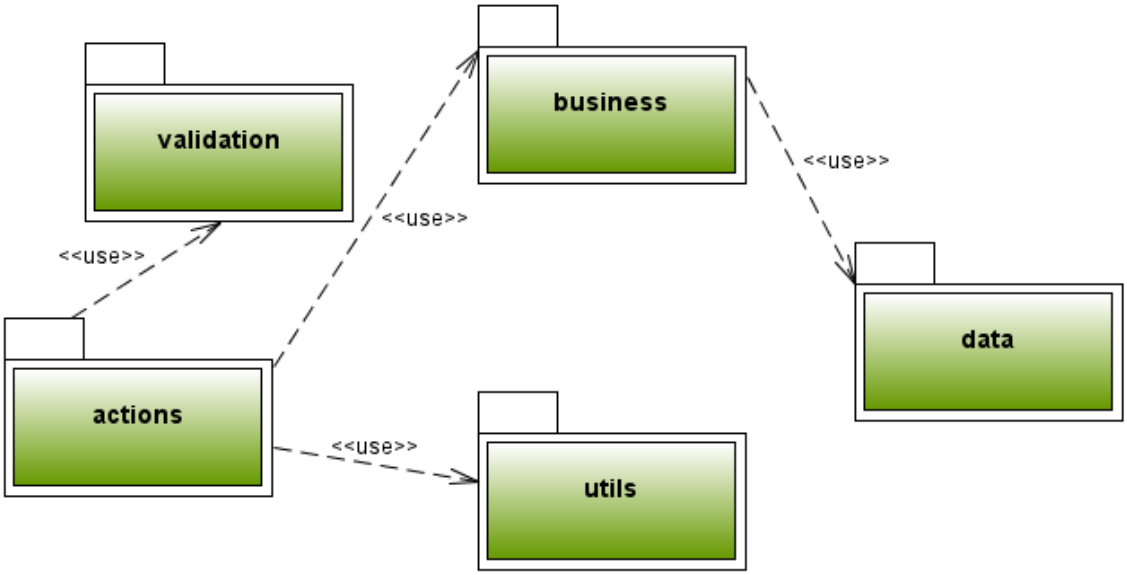


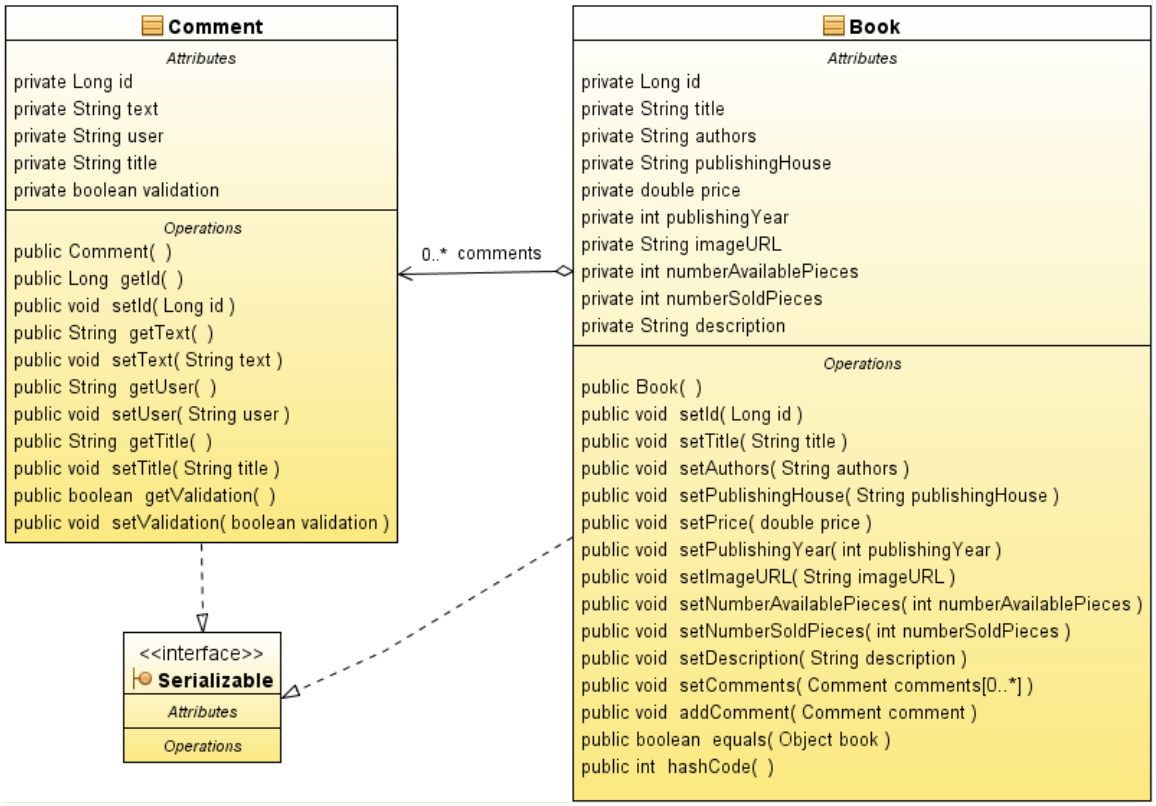
Figura 5.4 : Diagrama de pachete a sistemului web

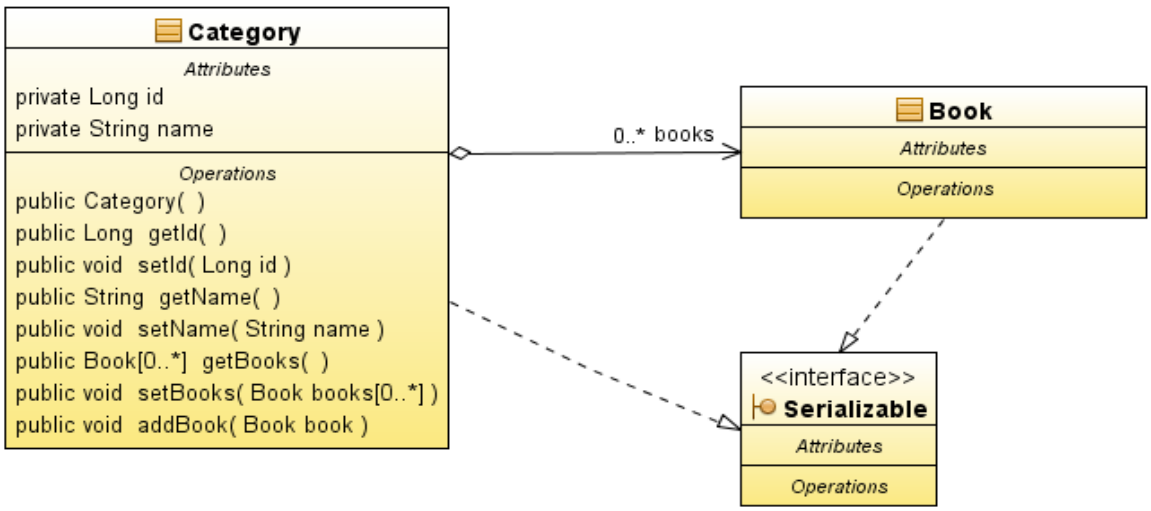
Din diagrama de pachete de mai sus, se deduce faptul ca pachetul principal al modelului este *actions*. Acest pachet utilizeaza, direct sau indirect, clasele si interfetele din celelalte pachete. Pachetul *actions* are doua subpachete, *user* si *admin*, care contin clasele corespunzatoare actiunilor celor trei actori ai sistemului. Fiecare cerere a unui actor se mapeaza pe o actiune din pachetul *actions*. Actiunea respectiva, utilizand si clasele celorlalte pachete ajutatoare, returneaza un raspuns clientului. Acest raspuns este afisat de pagina JSP asociata actiunii. Pachetul *data* contine clasele care construiesc baza de date a aplicatiei curente, impreuna cu constrangerile si relatiile dintre tabele. Pachetul *business* cuprinde clase si interfete care realizeaza operatii pe baza de date, accesand clasele din pachetul *data*. Pachetul *validation* are o clasa care verifica corectitudinea datelor introduse in sistem, iar pachetul *utils* contine

cateva elemente care completeaza functionalitatea sistemului. In continuare, se vor prezenta, detaliat, fiecare dintre componentele enumerate mai sus.

5.2.1.1 Modulul de date

Modulul de date al aplicatiei este constituit din clasele care apartin pachetului *data* : *Category*, *Book*, *Comment*, *User*, *Command*. Mai jos sunt ilustrate diagramele de clasa ale acestora si relatiile dintre ele :





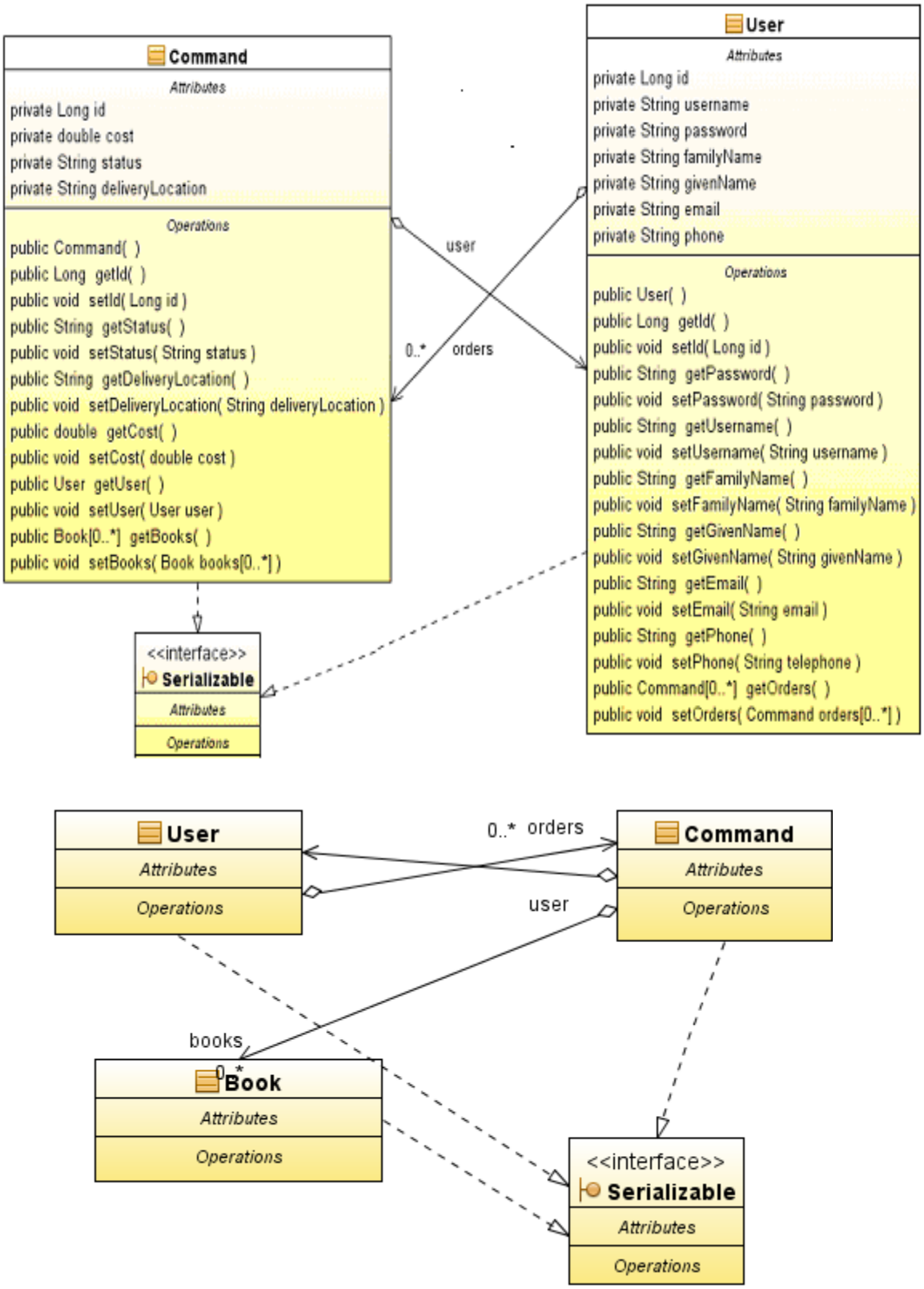


Figura 5.5 : Diagrame de clasa ale modulului de date

In primul rand, aceste clase sunt bean-uri (clase Java care respecta un anumit standard de definire). Toate au cate un constructor fara argumente. Proprietatile lor nu pot fi accesate direct (sunt definite ca *private*), ci doar prin intermediul getters si setters, care sunt definiti pentru fiecare proprietate.

Fiecare dintre aceste bean-uri se mapeaza pe cate un tabel in baza de date a aplicatiei. Sunt utilizate adnotari pentru a realiza maparile, dar si pentru a defini constrangerile bazei de date si relatiile dintre tabele. Adnotarea *@Entity* precizeaza faptul ca bean-ul curent se va mapa pe un tabel din baza de date. Definirea ei este obligatorie. Adnotarea *@Table(name = "nume_tabel")* specifica numele tabelului pe care se mapeaza clasa Java curenta. Se poate evita definirea ei, insa, in acest caz, tabelul din baza de date va avea acelasi nume precum bean-ul asociat lui.

Se observa ca toate cele cinci bean-uri definite implementeaza interfata *Serializable*. Cu alte cuvinte, obiectele corespunzatoare acestor clase Java sunt serializable, adica pot fi tranformate in fluxuri de octeti, pentru a fi persistate pe disc (intr- o baza de date sau un fisier) sau pentru a fi transmise, prin retea, altor aplicatii soft. Serializabilitatea este o conditie necesara pentru toate obiectele care trebuie sa aiba proprietatile enuntate anterior. Intrucat, in aplicatia curenta, bean-urile definite sunt automat persistate in baza de date, ele trebuie sa fie serializabile.

Starea persistenta a acestor entitati este data de campurile sau proprietatile persistente asociate. Pentru o entitate, toate campurile care nu sunt marcate cu adnotarea *@Transient* sunt considerate persistente. Exista doua moduri de a implementa persistenta pentru o anumita entitate : campuri persistente si proprietati persistente. Pentru a defini o entitate care utilizeaza campuri persistente, trebuie ca pentru fiecare variabila instanta a clasei Java sa fie definite adnotari care sa le mapeze pe cate o coloana a tabelului reprezentat de bean-ul curent. Daca se doreste construirea unui bean folosind metoda proprietatilor persistente, adnotarile trebuie aplicate fiecarei perechi de setter-getter definite in cadrul entitatii. De remarcat ca aceste doua solutii sunt echivalente (au acelasi efect). Totodata, nu se pot aplica adnotari atat campurilor cat si perechilor getter-setter in cadrul aceleeasi entitati. Daca persistenta pentru un asemenea bean nu este definita explicit, de catre programator, compilatorul realizeaza o mapare implicita, cand aplicatia este pusa pe server, la *deploy-time.* In acest caz, fiecare variabila instanta a clasei Java va fi mapata pe cate o coloana cu acelasi nume din tabelul curent.

La implementarea starii persistente pentru bean-urile aplicatiei curente, s-a utilizat metoda proprietatilor persistente. Fiecare entitate are un cate un camp, *id*, care este cheie primara in tabelul asociat. Acest lucru este specificat prin adnotarea *@Id*. Adnotarea *@GeneratedValue* precizeaza faptul ca id-urile sunt generate automat, de catre sistemul de gestiune al bazei de date. Caracteristicile coloanelor (nume, constrangeri de unicitate sau de elemente nenule, marimea datelor care incap in coloane) sunt descrise prin adnotarea *@Column* si prin proprietatile acesteia : *name, nullable, unique, length*.

Relatiile dintre entitati pot fi de patru tipuri : unu-la-unu (o instanta a unei entitati corespunde unei singure instante ale unei alte entitati), unu-la-n (o instanta a unei entitati corespunde mai multor instante ale unei alte entitati), n-la-unu (mai multe instante ale unei entitati corespund unei singure instante ale unei alte entitati), n-la-n (mai multe instante ale unei

entitati sunt asociate cu mai multe instante ale altei entitati). Totodata, aceste relatii pot fi unidirectionale sau bidirectionale.

Relatiile de tip unu-la-unu sunt specificate prin adnotarea *@OneToOne*. Nu s-au folosit relatii de acest tip la modelarea datelor aplicatiei curente. Relatiile de tip unu-la-n sunt definite prin adnotarea *@OneToMany*. In aplicatia curenta, exista relatii de acest tip intre mai multe perechi de entitati : *Category-Book* (unei categorii de carti ii corespund mai multe carti), *Book-Comment* (pentru o anumita carte exista mai multe comentarii) si *User-Command* (un user poate efectua mai multe comenzi). Prima entitate din aceste relatii reprezinta tabelul sursa, iar cea de-a doua tabelul destinatie. Primele doua relatii enumerate mai sus sunt unidirectionale. Aceasta inseamna ca numai entitatile sursa (*Category*, *Book*) contin referinte la entitatile destinatie (*Book*, *Comment*), nu si invers. La implementarea acestor relatii este utilizata adnotarea *@JoinColumn* (in entitatea sursa, alaturi de adnotarea *@OneToMany*, pentru proprietatile asociate campurilor de tipul entitatii destinatie). Aceasta are atributul *name* care specifica numele coloanei din tabelul sursa care va fi cheie straina in tabelul destinatie. A treia relatie specificata mai sus este bidirectionala. In acest caz, o entitate de tip *User* contine o referinta la o entitate de tip *Command* si invers. Atributul *mappedBy* al adnotarii *@OneToMany* este utilizat, in entitatea *User* (pentru proprietatea asociata campului de tipul *Command*), pentru a specifica proprietatea entitatii *Command* a carei coloana asociata va fi cheie straina in tabelul pe care se mapeaza acest bean. Relatiile de tip n-la-unu sunt descrise prin adnotarea *@ManyToOne*. O singura pereche de entitati este legata printr-un asemenea tip de relatie, si anume : *Command-User* (aceeasi comanda poate fi efectuata de mai multi useri). Aceasta relatie este, de asemenea, bidirectionala. Relatiile de tip n-la-n sunt specificate prin adnotarea *@ManyToMany*. Definirea unei asemenea relatii implica construirea unui nou tabel, asociat cu aceasta. Intre entitatile *Command* si *Book* exista o astfel relatie. Este nevoie de un nou tabel care sa retina, pentru fiecare comanda, cartile asociate (tabelul *book_order*, descris in sectiunea de prezentare a arhitecturii bazei de date a sistemului web). Tabelul acesta este creat prin adnotarea *@JoinTable*. Atributele adnotarii (*name*, *joinColumns* etc) permit specificarea numelui si a coloanelor tabelului tocmai definit. De remarcat ca relatia dintre cele doua entitati este unidirectionala (numai entitatea *Command* are referinte la entitatea *Book*, nu si invers).

5.2.1.2 Modulul de bussines

Modulul de business al aplicatiei este continut in pachetul *business*. Acesta cuprinde urmatoarele clase si interfete : *BooksServiceInterface*, *BooksServiceImplementation*, *UserServiceInterface*, *UserServiceImplementation*.

Toate entitatile definite in modulul anterior sunt gestionate de o instanta a clasei *EntityManager*. Instanta unui *EntityManager* este asociata cu un context de persistenta in care entitatile sunt create, pastrate si modificate. Contextul de persistenta poate cuprinde una sau mai multe baze de date. In cazul de fata, acesta contine doar o singura baza de date, cea specifica sistemului web dezvoltat. Modul in care sunt configurate *EntityManager*-ul aplicatiei curente si contextul de persistenta asociat, este descris, in detaliu, in subcapitolul urmator.

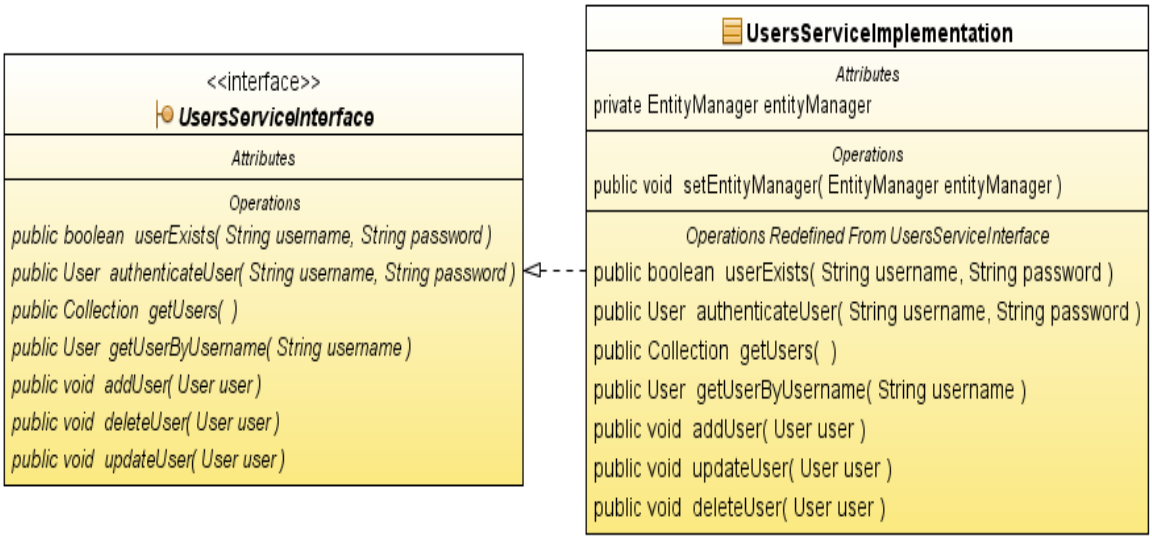
Fiecare din cele doua clase ale pachetului *business* obtin o referinta (prin metoda injectarii de referinte) la *EntityManager*-ul care gestioneaza contextul de persistenta ce cuprinde baza de date a aplicatiei. Clasa *BooksServiceImplementation* implementeaza metodele interfetei

BooksServiceInterface si realizeaza diverse operatii (vizualizare, creare, modificare, stergere) pe tabelele pe care se mapeaza entitatile *Category*, *Book*, *Comment*, *Command*. Clasa *UsersServiceImplementation* implementeaza metodele interfetei *UsersServiceInterface* si realizeaza acelasi tip de operatii pe tabelul pe care se mapeaza entitatea *User*. Asadar, aceste clase utilizeaza bean-urile din pachetul *data*, pentru a realiza diverse operatii cu ele. Metodele definite in componentele acestui modul sunt apelate direct de catre actiunile din pachetul *actions*, pentru a returna anumite informatii din baza de date a aplicatiei sau pentru a efectua anumite modificari pe aceasta.

Se observa ca, pentru setter-ul prin care se injecteaza instanta *EntityManager*-ului in clasele curente, se utilizeaza adnotarea *@PersistenceContext*, pentru a transfera acelasi context de persistenta tuturor componentelor aplicatiei, care sunt implicate intr-o tranzactie si care utilizeaza aceeasi instanta a unui *EntityManager*. De asemenea, fiecare din cele doua clase ale pachetului sunt marcate cu adnotarea *@Transactional*. Aceasta inseamna ca fiecare metoda a acestor clase va fi, implicit, @Transactional. Asadar, fiecare metoda va fi executata intr-o noua tranzactie.

Componentele pachetului curent utilizeaza cateva metode definite in clasa *EntityManager*, pentru a efectua operatiile dorite : *persist* (salveaza o instanta a unei entitati in baza de date a aplicatiei), *remove* (sterge o instanta a unei entitati din baza de date a aplicatiei), *merge* (reataseaza o instanta a unei entitati de contextul de persistenta al aplicatiei curente, in cazul in care nu mai e atasata de acesta, sau o salveaza, daca instanta nu exista), *createQuery* (creeaza interogari pe baza de date utilizand JPQL), *createNativeQuery*(defineste interogari pe baza de date folosind limbajul de interogare specific sistemului de gestiune al bazei de date utilizat, in cazul de fata MySQL).

In schemele de mai jos sunt prezentate diagramele de clasa ale componentelor acestui pachet :



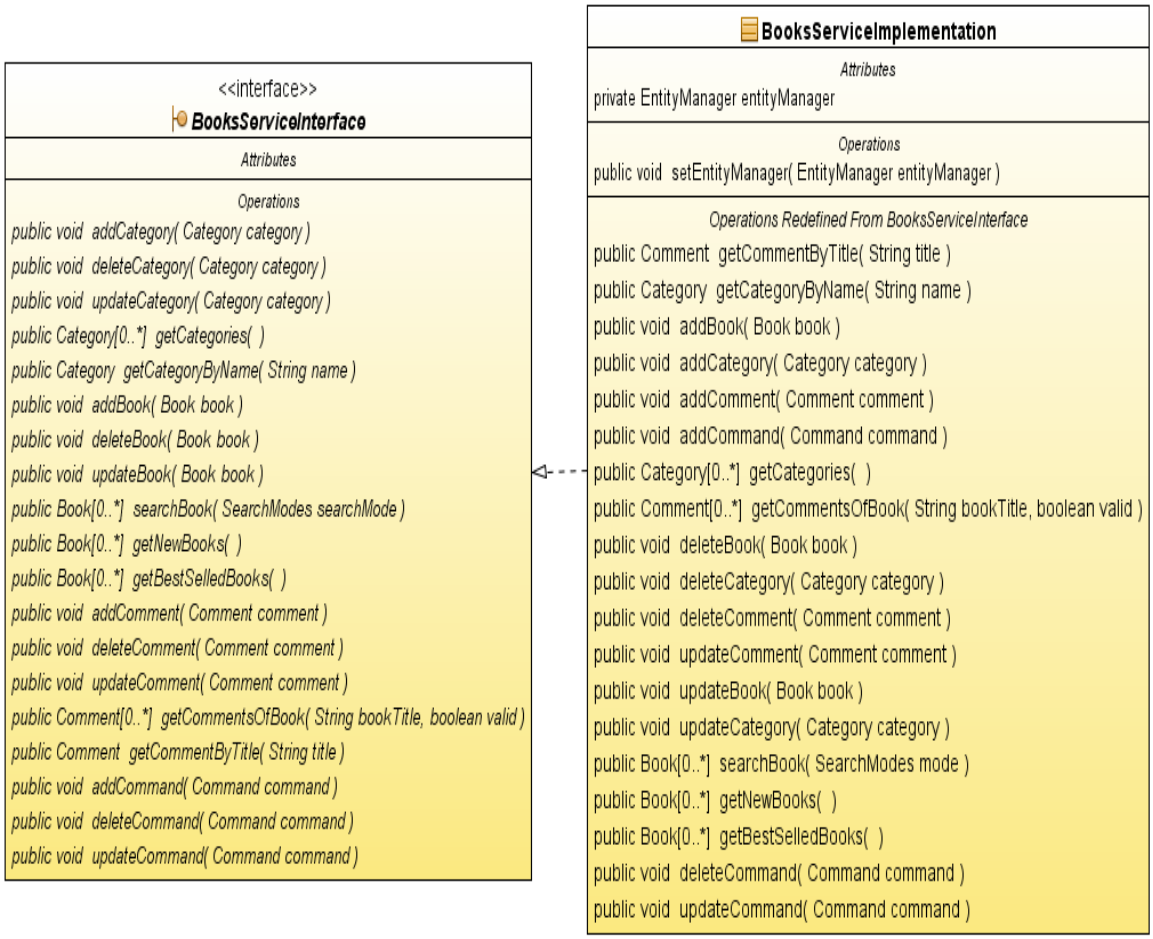


Figura 5.6 : Diagrame de clasa ale modului de business

5.2.1.3 Integrarea modulelor de date si business in aplicatia web

In acest subcapitol este descris modul in care framework-ul Struts2 integreaza si utilizeaza modulele de *date* si de *business* prezentate anterior. Se vor analiza trei din cele patru fisiere de configurare ale aplicatiei, unde sunt realizate setarile necesare pentru ca framework-ul sa poata folosi aceste pachete.

Fisierul *applicationContext.xml* contine configurarile necesare construirii bazei de date a aplicatiei utilizand JPA si modul in care este folosit Spring pentru a gestiona entitatile create astfel. In figura de pe pagina urmatoare este afisat continutul acestui fisier. In prima parte sunt definite spatiile de nume si fisierele necesare pentru a folosi framework-ul Spring impreuna cu Struts2. In sectiunea a doua este declarat un procesor de bean-uri care cauta, in toate bean-urile Spring, adnotari de tipul *@EntityManager*, pentru a injecta acolo, prin metode de tip setter, o instanta a *EntityManager*-ului creat ulterior. In al treilea fragment numerotat sunt create cele doua bean-uri String ale aplicatiei, identificate prin id-urile *usersService* si *booksService*. Acestea corespund claselor pachetului *business*. Ele sunt inserate, prin tehnica Spring *autowiring*, in toate actiunile si interceptorii sistemului web. In acest fel, actiunile pot utiliza clasele pachetului *business* pentru a realiza diverse operatii pe baza de date. In sectiunea a patra

este definit bean-ul *EntityManagerFactory*, prin care se creeaza automat o instanta a *EntityManager*-ului aplicatiei. Acesta este asociat cu un sistem de gestiune al bazelor de date (in acest caz MySQL) si cu o sursa de date, identificata prin id-ul *datasource*, care este tot un bean Spring. Al cincilea fragment al fisierului defineste si descrie proprietatile sursei de date care a fost asociata *EntityManager*-ului definit anterior. Sunt specificate driver-ul folosit pentru conectarea la MySQL, url-ul bazei de date a sistemului web si modul in care e permis accesul la baza de date (prin user si parola). In ultimele doua sectiuni este detaliat modul in care sistemul web utilizeaza tranzactii pentru operatiile sale. Este definit un bean Spring, identificat prin id-ul *transactionManager*, care este asociat *EntityManager*-ului aplicatiei. Se specifica apoi faptul ca se vor utiliza adnotari pentru a defini clasele si metodele care utilizeaza tranzactii.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation = "http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
                             http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.0.xsd">

     <bean class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor" /> (2)

     <bean id="usersService" class="library.model.business.classes.UsersServiceImplementation"></bean>
     <bean id="booksService" class="library.model.business.classes.BooksServiceImplementation"></bean>

     <bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
          <property name="dataSource" ref="dataSource" />
          <property name="jpaVendorAdapter">
               <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
                    <property name="database" value="MYSQL" />
                    <property name="showSql" value="true" />
               </bean>
          </property>
     </bean>

     <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
          <property name="driverClassName" value="org.gjt.mm.mysql.Driver" />
          <property name="url" value="jdbc:mysql://127.0.0.1:3306/library" />
          <property name="username" value="root" />
          <property name="password" value="cataaristotel" />
     </bean>

     <bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
          <property name="entityManagerFactory" ref="entityManagerFactory" />
     </bean>

     <tx:annotation-driven transaction-manager="transactionManager" /> (7)

</beans>
```

(1) (3) (4) (5) (6)

Fisierul *web.xml* contine, in afara controller-ului aplicatiei definit in sectiunea a doua si mapat pe toate cererile aplicatiei, in sectiunea a cincea, alte doua elemente specifice framework-ului Spring.

Pentru a integra plugin-ul Spring in Struts2, sunt necesare anumite librarii .jar, care au fost incluse in fisierul *lib* al sistemului web dezvoltat. Apoi, trebuie creat container-ul Spring care va gestiona bean-urile Spring definite in aplicatia curenta. Pentru a realiza acest lucru, este definit un *application-context listener*, in ultima parte a fisierului curent. Acesta cauta, in fisierul *applicationContext.xml*, bean-urile Spring care vor fi utilizate de container.

Conceptul de *lazy-loading* este una din acele optimizari realizate de catre JPA, pentru a reduce cantitatea de informatii care vin din baza de date. Se refera la incarcarea tarzie (doar la nevoie) a elementelor mai adanci ale unui obiect care urmeaza sa fie afisat. De exemplu, daca se returneaza o categorie de carti din baza de date, cartile care apartin acelei categorii nu vor fi incarcate in acelasi timp user-ul, ci doar cand vor fi referite. Intr-o aplicatie bazata pe MVC si orientata pe actiuni, informatiile din baza de date sunt preluate de catre clasele asociate actiunilor. Apoi, actiunile cedeaza controlul layer-ului de vederi, care vor afisa aceste informatii. In cazul in care *entityManager*-ul este inchis, in momentul afisarii anumitor date, incercarea de a le incarca va esua. Pentru a rezolva aceasta problema, se utilizeaza un filtru servlet care infasoara un *entityManager* in jurul intregului proces de tratare a unei cereri, incluzand aici si layer-ul vederilor, care afiseaza rezultatele cerute. Spring ofera o implementare a unui asemenea filtru, care este configurat in fisierul curent, in a treia si a patra sectiune. De remarcat este faptul ca filtrul Spring trebuie sa fie mapat inaintea filtrului Struts2. Odata definit, filtrul Spring creeaza un *entityManager* si il transmite firului de executie care trateaza cererea curenta. Pentru a realiza acest lucru, utilizeaza *entityManagerFactory*-ul declarat in fisierul *applicationContext.xml*. Continutul fisierului *web.xml* este afisat mai jos :

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                             http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
         version="2.5">

    <filter>
      <filter-name>struts2</filter-name>
      <filter-class>org.apache.struts2.dispatcher.FilterDispatcher</filter-class>
    </filter>

    <filter>
      <filter-name>SpringOpenEntityManagerInViewFilter</filter-name>
      <filter-class>
            org.springframework.orm.jpa.support.OpenEntityManagerInViewFilter
      </filter-class>
    </filter>

    <filter-mapping>
      <filter-name>SpringOpenEntityManagerInViewFilter</filter-name>
      <url-pattern>/*</url-pattern>
    </filter-mapping>

    <filter-mapping>
      <filter-name>struts2</filter-name>
      <url-pattern>/*</url-pattern>
    </filter-mapping>

    <listener>
      <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>

</web-app>
```

La utilizarea JPA, toate entitatile si obiectele legate de persistenta sunt continute intr-o unitate de persistenta. *EntityManagerFactory*-ul gestionat de Spring incapsuleaza toate detaliile unitatii de persistenta asociate (metadata care descrie modul in care clasele Java se mapeaza pe tabelele din baza de date, detalii de conexiune la baza de date etc). Fisierul principal de configurare a unei unitati de persistenta este *persistence.xml*. Continutul acestuia este afisat mai jos. Se specifica numele unitatii de persistenta si un parametru, care poate controla crearea automata a tabelelor din baza de date *library*, pe care se mapeaza clasele pachetului *data*.

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                       http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
   version="1.0">

      <persistence-unit name="cataLibrary">
            <properties>
                  <!--<property name="hibernate.hbm2ddl.auto" value="create"/-->
            </properties>
      </persistence-unit>
</persistence>
```

Astfel, la *deploy-time*, sunt analizate cele trei fisiere prezentate mai sus. Daca setarile efectuate sunt corecte si daca baza de date definita in fisierul *applicationContext.xml* exista, atunci, in functie de valoarea parametrului din fisierul *persistence.xml*, se creeaza tabele in baza de date curenta, prin maparea claselor Java din pachetul *data*, sau se pastreaza structura veche si continutul anterior al bazei de date definite de user.

5.2.1.4 Implementarea logicii aplicatiei

Logica sistemului web este implementata de actiunile din pachetul *actions*. Exista actiuni specifice doar user-ului sau vizitatorului (continute in subpachetul *user*), doar administratorului (incluse in subpachetul *admin*) sau comune celor doi actori (definite in subpachetul *common*).

Pachetul *admin* cuprinde subpachetele *books*, *categories*, *comments*, *orders*, *users*, care implementeaza operatiile asociate cu fiecare entitate specificata prin numele proprii.

Logica fiecarei actiuni este implementata in metoda *execute* asociata. Validarea datelor introduse e realizata prin metoda *validate*, definita de interfata *ValidationAware*, care e implementata de clasa *ActionSupport* (clasa pe care fiecare actiune definita o extinde). In fiecare actiune sunt inserate, prin *autowiring*, unul din cele doua bean-uri care corespund claselor pachetului *business*, *usersService* sau *booksService*. Acestea sunt utilizate pentru a efectua diverse operatii pe baza de date.

In continuare se va afisa diagrama de clasa asociata fiecarui actiuni si se va explica logica implementata de aceasta.

5.2.1.4.1 Pachetul user

5.2.1.4.1.1 ShowMainPage

Actiunea *ShowMainPage* corespunde paginii care apare cand un client acceseaza site-ul. In cadrul metodei *execute*, utilizand bean-ul *booksService*, se extrag din baza de date categoriile de carti, topul zece al celor mai bine vandute si al celor mai noi carti. Totodata, actiunea implementeaza interfetele *ApplicationAware* si *SessionAware*, pentru a avea acces la map-urile asociate domeniilor de vizibilitate aplicatie si sesiune. In map-ul asociat aplicatiei se retine, in cazul in care nu exista deja, un obiect de tip *Cart*, care reprezinta cosul de cumparaturi al clientului. Acesta este folosit pentru a extrage numarul de carti care se afla deja in cos. Map-ul asociat sesiunii e utilizat pentru a verifica daca un user este logat, si, in caz afirmativ, pentru a extrage numele user-ului respectiv. Toate aceste informatii sunt puse pe *ValueStack*, pentru a putea fi preluate de catre pagina JSP asociata cu aceasta actiune. Acest lucru este realizat prin definirea de getters si setters corespunzatori fiecarui tip de informatie care trebuie vizualizata. Metoda *execute* returneaza variabila *success*, ceea ce inseamna ca se executa intotdeauna cu succes.

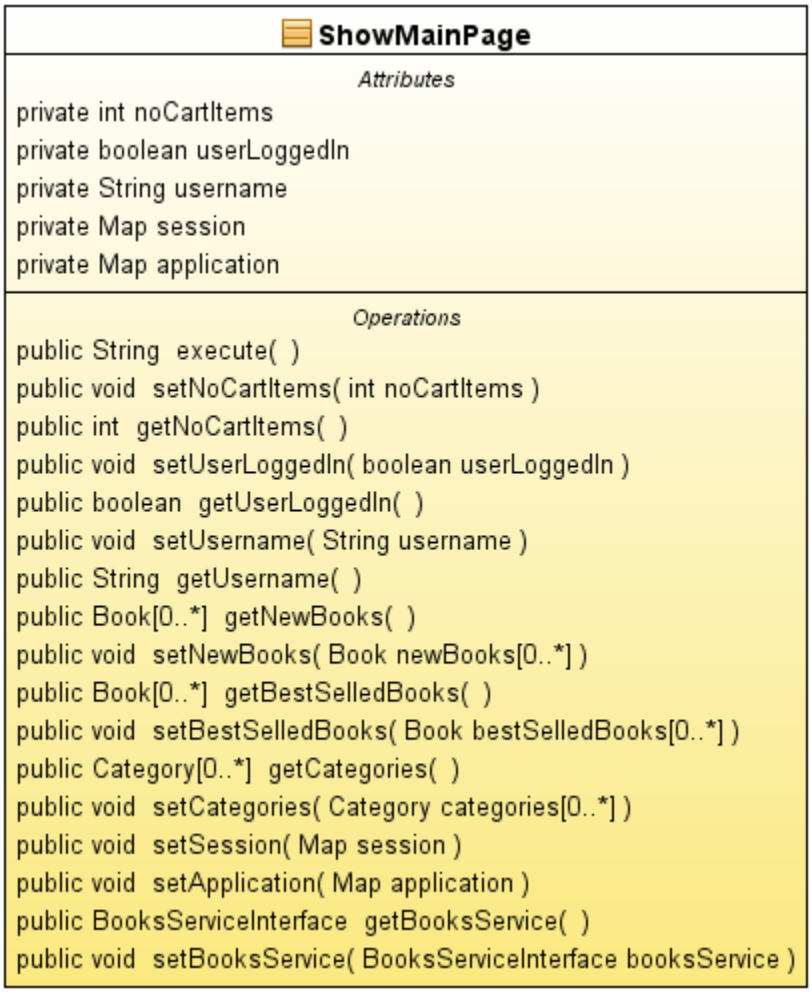


Figura 5.7 : Diagrame de clasa ale modulului de logica a aplicatiei (ShowMainPage)

5.2.1.4.1.2 Register

Actiunea *Register* realizeaza inregistrarea unui vizitator in baza de date a site-ului. Ea implementeaza interfata *ModelDriven*, pentru a putea prelua direct un obiect de tip *User* (utilizand datele introduse in pagina de intregistrare). Se evita astfel definirea de getters si setters pentru toate obiectele atomice care reprezinta particularitati ale user-ului (nume, prenume, e-mail, numar de telefon, user, parola). Din pagina de inregistrare se transmite actiunii *Register* un singur obiect, de tip *User*, care este preluat prin setter-ul definit in actiune. Apoi getter-ul asociat faciliteaza transmiterea obiectului pe *ValueStack*, de unde poate fi preluat de pagina JSP asociata cu aceasta actiune. Metoda *validate* realizeaza validarea datelor introduse de la tastatura. Utilizand bean-ul *usersService*, se verifica daca user-ul si parola introduse mai exista in baza de date. In caz afirmativ, se asociaza o eroare cu campul corespunzator user-ului. Altfel, se verifica daca datele introduse respecta constrangerile definite in clasa *Check*, a pachetului *validation*. La prima neconcordanta gasita, se adauga o eroare in campul asociat informatiei introduse gresit si se opreste executia metodei *validate*. Daca dupa executia metodei *validate* se gasesc erori definite anterior, metoda execute nu se mai executa si actiunea returneaza implicit valoarea *input*. In caz contrar, metoda *execute* se executa. Un obiect de tip *User* este adaugat in baza de date, folosind bean-ul usersService. Apoi se returneaza variabila *success*.

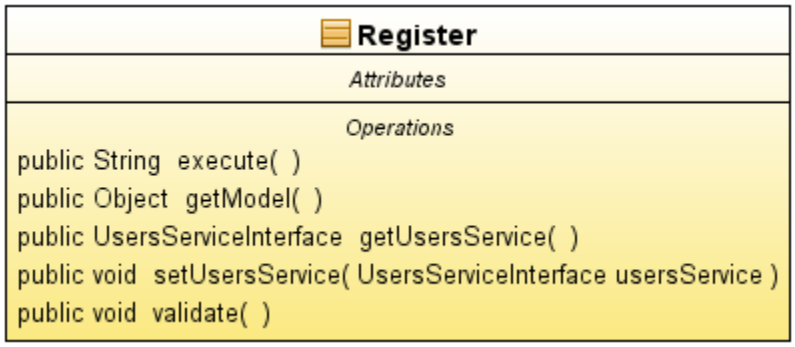


Figura 5.8 : Diagrame de clasa ale modului de logica a aplicatiei (Register)

5.2.1.4.1.3 SaveAccountDetails

Actiunea *SaveAccountDetails* face posibila vizualizarea user-ului si parolei unui client, in momentul in care acesta vrea sa le modifice. Toate programele profesioniste, la efectuarea unor operatii de update, pun la dispozitie vechile valori care urmeaza sa fie modificate. Actiunea implementeaza interfata *UserAware*, din pachetul *utils*, subpachetul *injections*. Prin metoda *setUser* se insereaza user-ul curent (obiectul de tip *User*), pentru ca datele lui sa poata fi accesate. Aceasta metoda este apelata automat, de catre interceptorul definit explicit de clasa *AuthentificationInterceptor*, din pachetul *utils*, subpachetul *interceptors*. Interceptorul este inserat in stiva de interceptori care conditioneaza executia actiunii curente si are rolul de a nu permite executia unor actiuni, daca user-ul nu este logat. In acest caz, un client nu isi poate updata contul pana cand nu se logheaza. Daca clientul care vrea sa isi modifice anumite date nu este logat, interceptorul asociat actiunii de update returneaza un rezultat global (*login-updateAccount* in acest caz). Rezultatele globale sunt definite in fisierul *struts.xml* si au rolul de a redirectiona anumite actiuni. In cazul de fata, clientul este automat redirectionat spre operatia de logare si abia apoi se revine la operatia de updatare de cont. Daca insa clientul este deja logat,

interceptorul permite executia actiunii curente, dupa ce pune la dispozitia acesteia obiectul de tip *User*, prin apelul metodei *setUser* din interfata *UserAware*. Actiunea curenta implementeaza si interfata *ModelDriven*. Prin metoda *getModel* a acesteia, se pune pe *ValueStack* obiectul de tip *User*. De acolo, acesta poate fi preluat de pagina JSP asociata cu actiunea curenta. Metoda *validate* a actiunii este goala si returneaza variabila *success*.

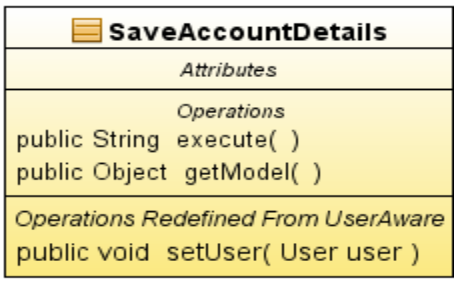


Figura 5.9 : Diagrame de clasa ale modului de logica a aplicatiei (SaveAccountDetails)

5.2.1.4.1.4 UpdateAccount

Actiunea *UpdateAccount* realizeaza operatia propriu-zisa de updatare cont pentru un anumit user. Metoda *validate* verifica daca noile date (username si parola) introduse sunt corecte (daca acestea corespund normelor definite in clasa *Check* a pachetului *validation*), daca acestea nu sunt identice cu cele vechi sau daca nu coincid cu un alt username sau cu o alta parola existente in baza de date. Daca una din conditiile enumerate mai sus nu este respectata, se adauga o eroare la campurile in care s-au introdus noile valori si executia metodei *validate* se suspenda, actiunea returnand, implicit, rezultatul *input*. Daca nu exista erori, se executa metoda *execute*. Utilizand bean-ul *usersService*, se updateaza, in baza de date, vechiul user. Apoi se sterge vechiul user, din map-ul sesiunii curente (se are acces la acesta deoarece actiunea implementeaza interfata *SessionAware*) si este inserat acolo user-ul modificat. Actiunea implementeaza si interfata *ModelDriven*, si suprascrie metoda *getModel*, prin care se pun la dispozitia acesteia noile date introduse de la tastatura, sub forma unui obiect de tipul *User*.

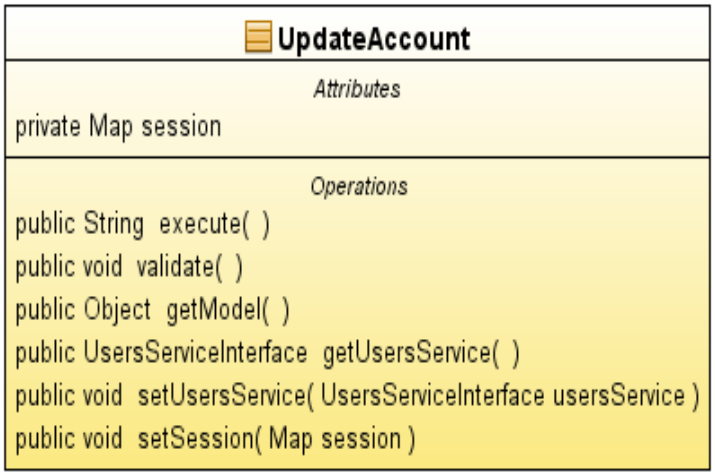


Figura 5.10 : Diagrame de clasa ale modului de logica a aplicatiei (UpdateAccount)

5.2.1.4.1.5 ShowBooks

Actiunea *ShowBooks* este asociata cu a doua pagina pe care o poate vizita user-ul. O parte din datele puse la dispozitie pentru afisare sunt identice cu informatia pusa pe *ValueStack* de actiunea *ShowMainPage*. Diferenta este ca, in loc de a afisa cele mai noi carti, se afiseaza cartile cautate de un client, dupa anumite criterii. Actiunea curenta este declansata atunci cand clientul selecteaza o categorie, pentru a afisa cartile din ea, sau cand userul cauta explicit o carte (dupa autor, titlu, editura sau categorie). La ambele tipuri de cautari, se transmite, ca parametri pentru actiunea curenta, informatia de cautare (numele categoriei dupa care se cauta, in primul caz, sau criteriile de cautare si valorile asociate cu acestea, in al doilea caz). Utilizand-se aceste informatii si bean-ul *booksService*, se returneaza din baza de date o lista de carti care corespund criteriilor de cautare. Se pun pe *ValueStack* doar zece carti din cele gasite, dar exista posibilitatea de a vizualiza toate perechile de cate zece carti existente.

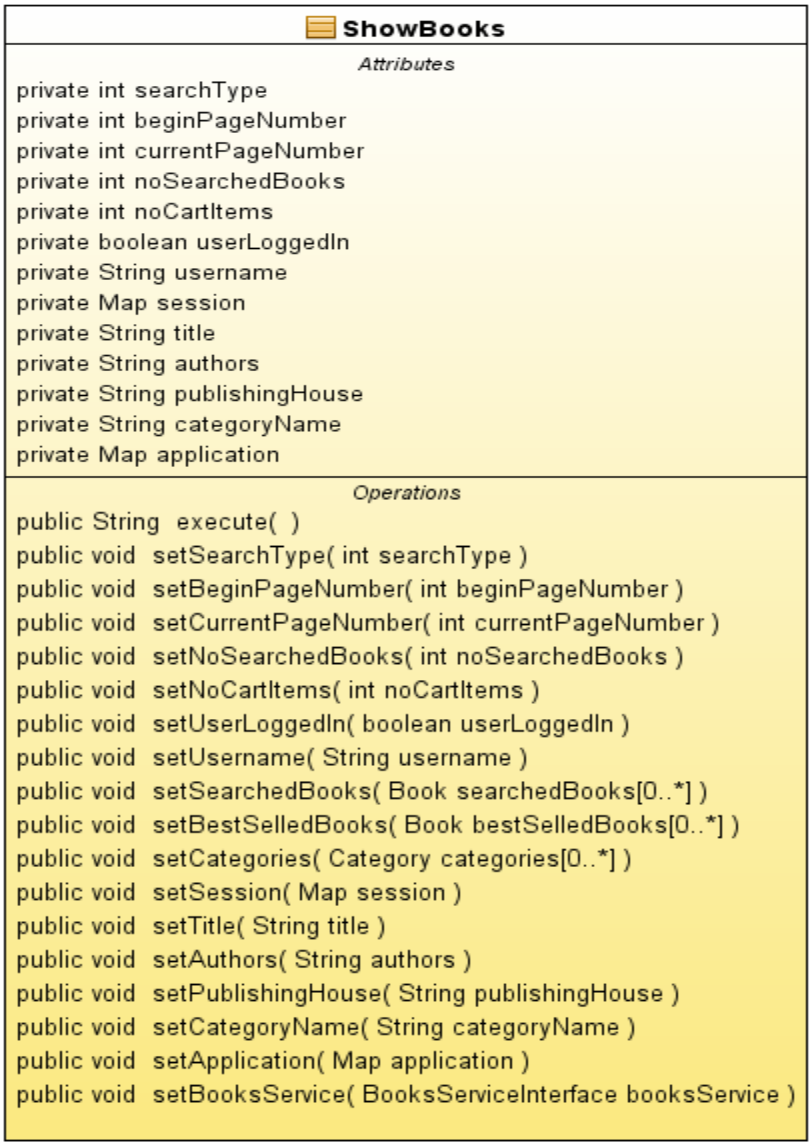


Figura 5.11 : Diagrame de clasa ale modului de logica a aplicatiei (ShowBooks)

5.2.1.4.1.6 ShowBookDetails

Actiunea *ShowBookDetails* este asociata cu a treia pagina pe care o poate vizita user-ul. O parte din datele puse la dispozitie pentru afisare sunt identice cu informatia pusa pe *ValueStack* de actiunea *ShowMainPage*. Diferenta este ca, in loc de a afisa cele mai noi carti, se pun pe *ValueStack* detalii despre o anumita carte. Actiunea curenta este declansata atunci cand clientul selecteaza o anumita carte (fie din cele mai noi, cele mai bine vandute sau cele obtinute in urma unei cautari) sau cand clientul adauga un nou comentariu la o anumita carte. Actiunea curenta mai pune pe *ValueStack* si comentariile cartii selectate (pe care le extrage din baza de date utilizand bean-ul *booksService*), care pot fi vizualizate apoi in perechi de cate zece. Se permite si adaugarea unui nou comentariu. In acest caz, se apeleaza metoda *validate* prin care se verifica corectitudinea datelor introduse. Daca informatia nu este corecta, actiunea returneaza valoarea *input*, cerandu-i clientului sa introduca date valide. Altfel, se executa metoda *execute*, care pune pe *ValueStack* informatiile enumerate anterior. Daca actiunea este invocata pentru a permite vizualizarea detaliilor despre o anumita carte, si nu pentru a adauga comentarii, metoda *validate* nu se mai executa. Metoda *execute* returneaza, in final, variabila *success*.

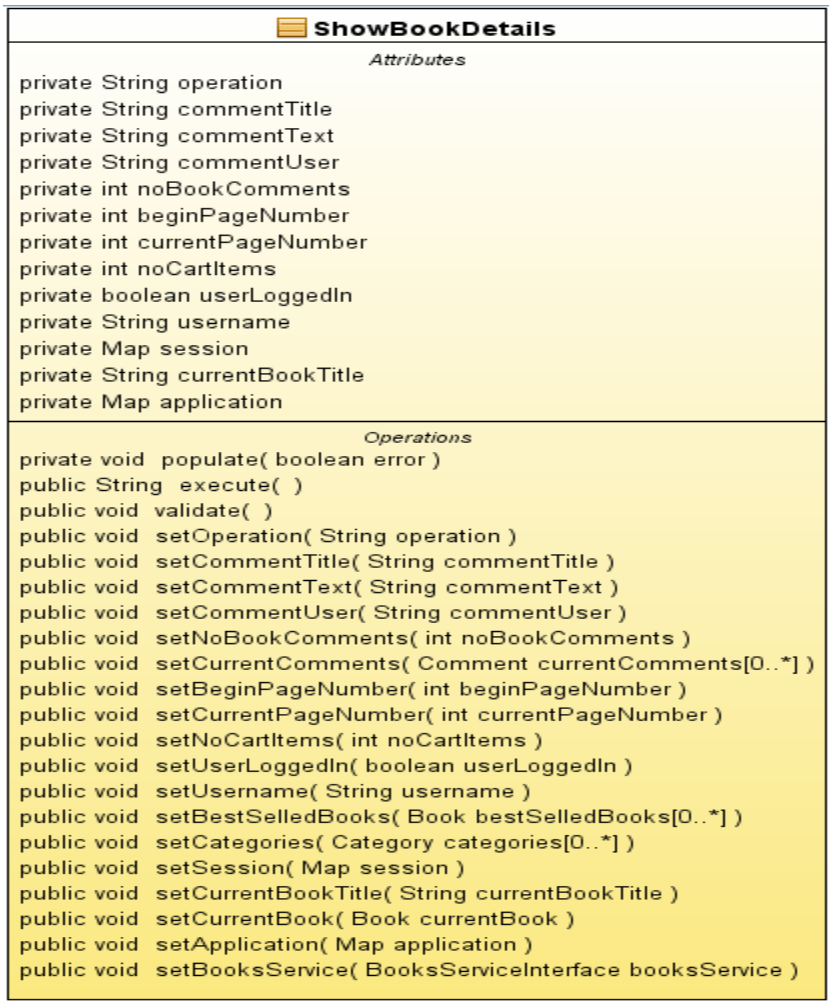


Figura 5.12 : Diagrame de clasa ale modului de logica a aplicatiei (ShowBookDetails)

5.2.1.4.1.7 ShowCart

Actiunea *ShowCart* corespunde ultimei pagini pe care o poate accesa user-ul. Rolul ei este de a oferi clientului posibilitatea de a vizualiza cosul de cumparaturi, de a modifica continutul acestuia (adaugare carti, eliminare carti, modificare numar exemplare comandate dintr-o anumita carte). Acestea sunt situatiile in care actiunea curenta este invocata. De remarcat este faptul ca actiunea implementeaza interfata *ApplicationAware*. Astfel, se poate obtine map-ul asociat contextului respectiv, pentru a extrage obiectul de tip *Cart*, pe care se pot face modificari sau interogari, pentru a obtine anumite informatii. Continutul metodei *validate* este executat doar daca actiunea este invocata ca urmare a unei operatii de modificare a numarului de exemplare dintr-o anumita carte existenta in cosul de cumparaturi. In cazul in care noul numar de exemplare introdus nu respecta constrangerile definite in clasa *Check* a pachetului *validate*, pentru acest tip de informatie, sau daca nu exista atatea exemplare disponibile in baza de date, actiunea returneaza, implicit, valoarea *input*. Altfel, se executa metoda *execute* care, dupa ce pune pe *ValueStack* toate informatiile legate de continutul cosului de cumparaturi, returneaza variabila *success*.

Figura 5.13 : Diagrame de clasa ale modului de logica a aplicatiei (ShowCart)

5.2.1.4.1.8 SendOrder

Actiunea *SendOrder* realizeaza operatia propriu-zisa de trimitere a unei comenzi. Metoda *validate* verifica daca adresa destinatie (unde se livreaza comanda) respecta constrangerile definite in clasa *Check* a pachetului *validation*, pentru acest tip de informatie. Totodata, cosul de cumparaturi nu poate fi gol, pentru nici o comanda. Apoi se verifica inca o data daca se poate efectua comanda (daca exista un numar suficient de bucati, din fiecare carte solicitata). Acest lucru este necesar deoarece pot apare scenarii de tipul : adaugare carti in cos (user1), adaugare carti in cos (user2), efectuare comanda (user2), efectuare comanda (user1). E foarte posibil ca, dupa ce al doilea user face comanda, in baza de date sa ramana un numar mai mic de bucati, dintr-o anumita carte, decat cantitatea pe care a vizualizat-o si a adaugat-o in cos primul user. La aparitia uneia din erorile enumerate mai sus, se retine intr-un String tipul erorii respective si apoi executia metodei *validate* se suspenda. Metoda *execute* analizeaza String-ul construit de metoda *validate*. Daca valoarea lui este aceeasi cu continutul unei erori, se returneaza un alt String asociat cu eroarea respectiva. Altfel, se updateaza numarul de carti ramase in baza de date, dupa efectuarea comenzii si se salveaza, in baza de date, un obiect de tip *Command*, reprezentand comanda respectiva. Acest lucru e realizat cu ajutorul bean-ului *booksService*, a carui metoda *addCommand* este apelata. In final, metoda returneaza variabila *success*.

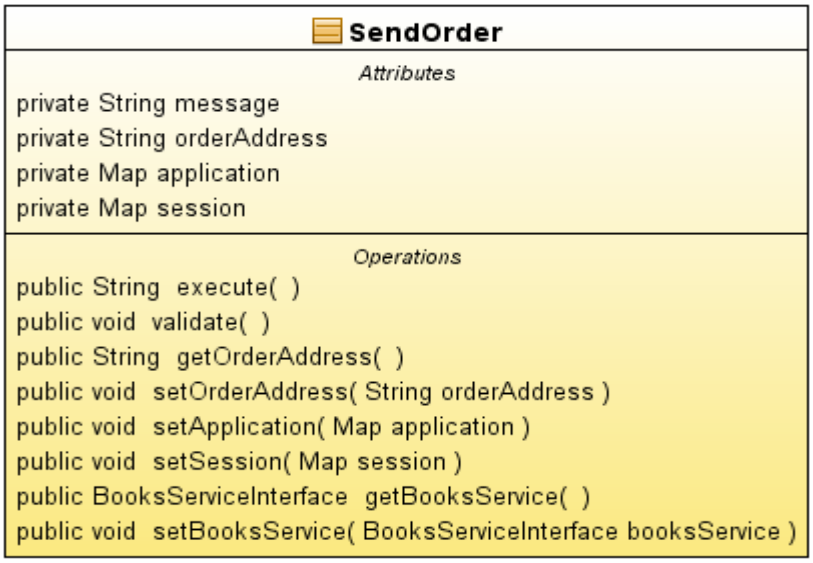


Figura 5.14 : Diagrame de clasa ale modului de logica a aplicatiei (SendOrder)

5.2.1.4.2 Pachetul common

5.2.1.4.2.1 Login

Actiunea *Login* corespunde operatiei de logare a unui user sau a unui administrator. Aceasta implementeaza interfata *ModelDriven*, pentru a prelua datele introduse in fereastra de logare de catre client, sub forma unui obiect de tip *User*. Este implementata si interfata *SessionAware*, pentru a avea acces la map-ul sesiunii curente, unde se doreste salvarea obiectului corespunzator user-ului logat. Actiunea are acces si la o variabila care indica tipul actiunii precedente, pusa pe *ValueStack* de catre o actiune “ajutatoare”, *SaveNextAction*. In functie de

valoarea acestei variabile, se returneaza un anumit String. Pentru fiecare astfel de String exista un alt rezultat care va fi afisat. Astfel, in metoda *validate*, daca username-ul si parola introduse sunt ambele egale cu *admin*, se seteaza variabila precedenta la valoarea *admin*. Altfel, se verifica daca username-ul si parola introduse respecta constrangerile definite in clasa *Check* a pachetului *validation*, pentru aceste tipuri de informatii, si daca aceste date exista in baza de date. Apoi se verifica daca clientul identificat prin datele respective nu este deja logat. Daca are loc vreuna din neconcordantele enuntate mai sus, se adauga o eroare la campurile asociate username-ului sau parolei clientului care vrea sa se logheze. Altfel, noul user este adaugat in map-ul sesiunii curente. Metoda *execute* returneaza variabila *success*, in cazul in care valoarea variabilei care indica tipul actiunii precedente este nula, sau variabila propriu-zisa, in caz contrar.

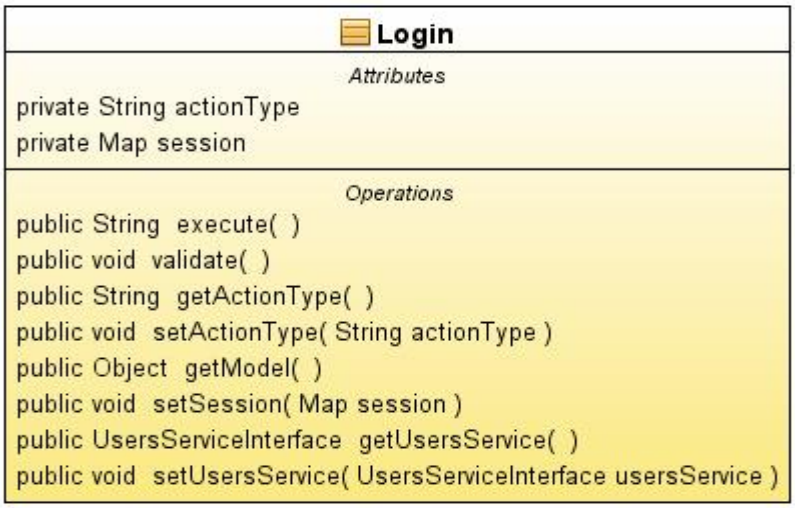


Figura 5.15 : Diagrame de clasa ale modului de logica a aplicatiei (Login)

5.2.1.4.2.2 SaveNextAction

Actiunea *SaveNextAction* este cea care salveaza pe *ValueStack* tipul actiunii precedente actiunii de logare. Este necesara deoarece e nevoie sa se cunoasca pagina sau actiunea la care trebuie sa se revina, dupa efectuarea logarii. Astfel, daca valoarea variabilei care indica tipul actiunii precedente este vida, inseamna ca logarea este facuta explicit de un client, si se revine la pagina principala a clientului. Altfel, daca aceasta variabila are valoarea *admin*, logarea este facuta de administratorul site-ului, si se revine la pagina principala a administratorului. Daca variabila are valoarea *updateAccount* sau *sendOrder*, se revine la actiunile corespunzatoare updatarii contului sau trimiterii comenzii. Se ajunge in una din ultimele doua situatii prezentate mai sus daca un anumit client incearca sa-si updateze contul sau sa efectueze o comanda, fara sa fie insa logat. Astfel, inainte de efectuarea operatiei dorite, clientul este obligat sa se autentifice si abia apoi acesta poate reveni la operatia precendenta, pentru a o executa.

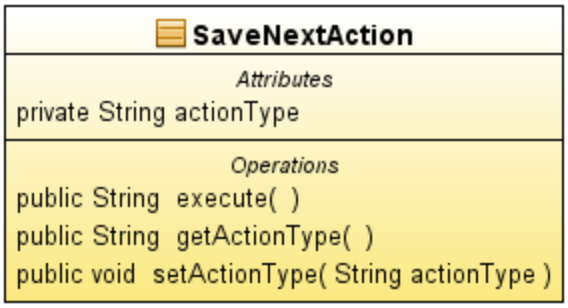


Figura 5.16 : Diagrame de clasa ale modului de logica a aplicatiei (SaveNextAction)

5.2.1.4.2.3 Logout

Actiunea *Logout* este asociata cu operatia de de-logare unui client. Aceasta implementeaza interfata *SessionAware*, pentru a avea acces la map-ul sesiunii curente. Metoda *execute* contine toata logica actiunii curente. Actiunea are acces la o variabila care indica tipul user-ului care vrea sa se delogheze. Aceasta variabila este transmisa, printr-un parametru, de catre pagina JSP din care se apeleaza actiunea curenta. Daca valoarea ei este *admin*, se returneaza un String corespunzator, care redirectioneaza user-ul spre fereastra de logare. Altfel, daca clientul nu este logat (daca map-ul sesiunii curente este vid), se returneaza un String care redirectioneaza user-ul tot spre fereastra de logare. Daca clientul este deja logat, acesta este sters din map-ul sesiunii curente, iar actiunea returneaza valoarea *success*, care va redirectiona clientul spre pagina principala a user-ului.

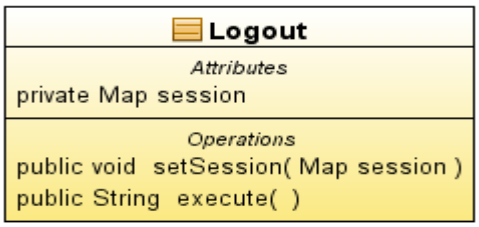


Figura 5.17 : Diagrame de clasa ale modului de logica a aplicatiei (Logout)

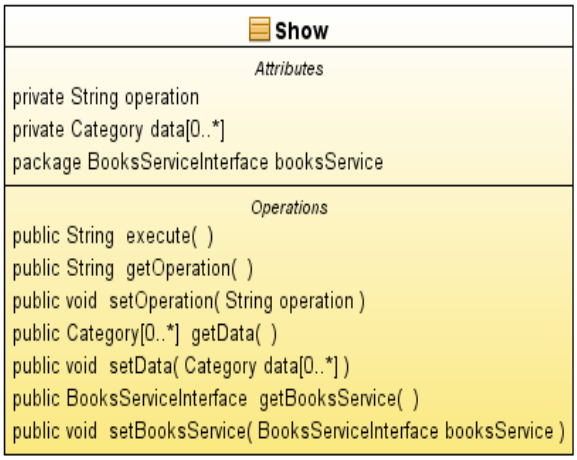
5.2.1.4.3 Pachetul admin

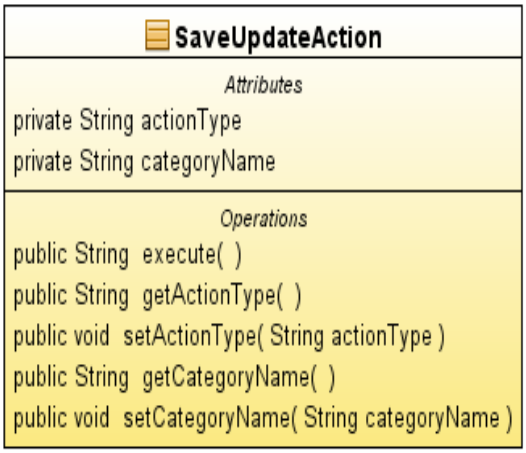
Logica implementarii actiunilor din acest pachet este similara cu logica utilizata pentru realizarea actiunilor prezentate anterior. Aceleasi bean-uri, *booksService* si *usersService*, sunt utilizate pentru implementarea diverselor operatii de vizualizare sau de modificare a bazei de date. Actiunile *Add* realizeaza adaugarea diverselor entitati (carti, categorii) in baza de date. Actiunile *Delete* efectueaza stergerea unor obiecte din baza de date (carti, categorii, comentarii, comenzi, useri), iar actiunile *Update* modifica anumite entitati (carti, categorii, comentarii – pot fi validate sau invalidate si, initial, sunt invalidate, comenzi – pot fi finalizate si, initial, sunt inregistrate). Actiunile *Show* vizualizeaza toate entitatile din baza de date (carti, categorii, comentarii, comenzi, useri), permitand administratorului sa analizeze informatia de care dispune si sa o modifice, daca doreste. Actiunile ajutatoare, *SaveAddAction* si *SaveUpdateAction*, sunt folosite pentru a salva pe *ValueStack* tipul actiunii efectuate, in cazul adaugarii sau modificarii

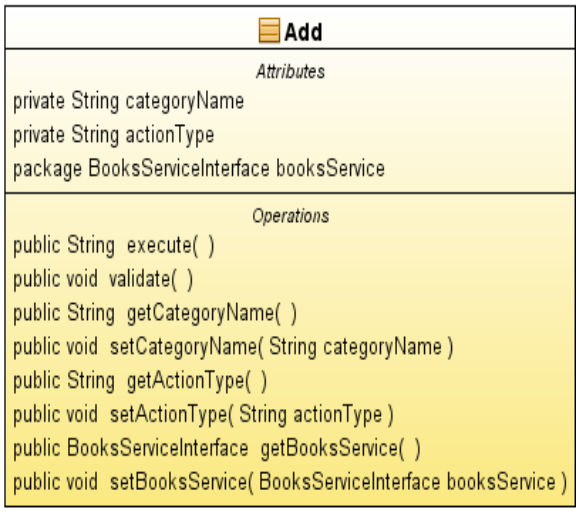
de carti si categorii de carti. Este nevoie de ele deoarece se utilizeaza aceeasi fereastra atat pentru adaugarea de carti sau categorii cat si pentru updatarea de carti sau categorii. Astfel, JSP-ul comun trebuie sa cunoasca tipul operatiei care se doreste a fi executata, pentru a fi capabil sa o termine cu succes.

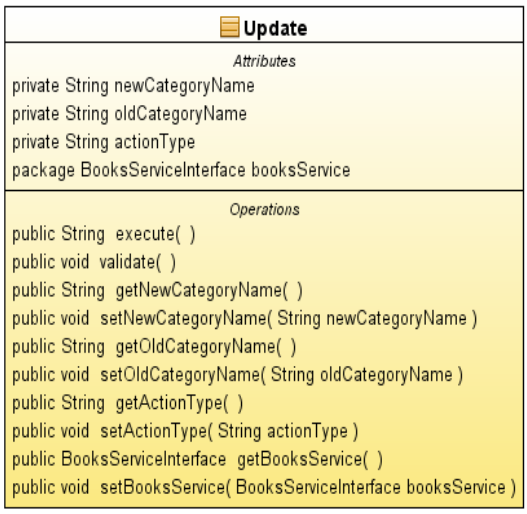
Am descris, sumar, logica actiunilor din subpachetele pachetului *admin*. Vor fi prezentate, in continuare, diagramele de clasa ale acestor actiuni.

5.2.1.4.3.1 Actiunile din pachetul categories









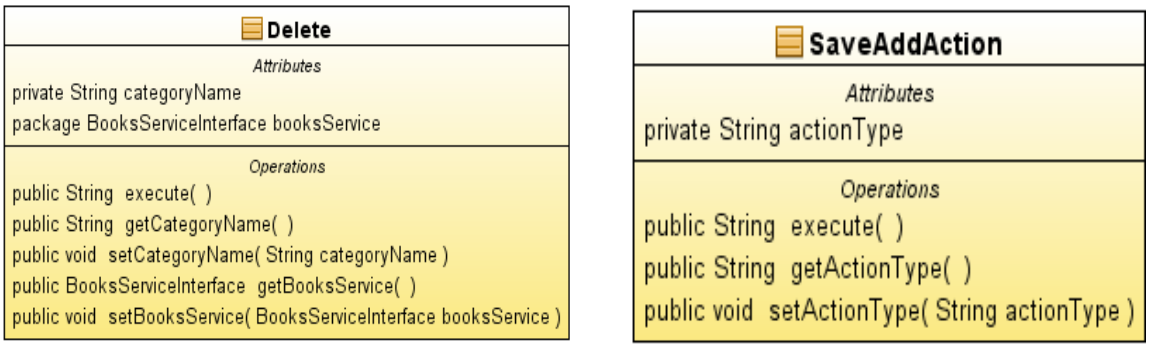
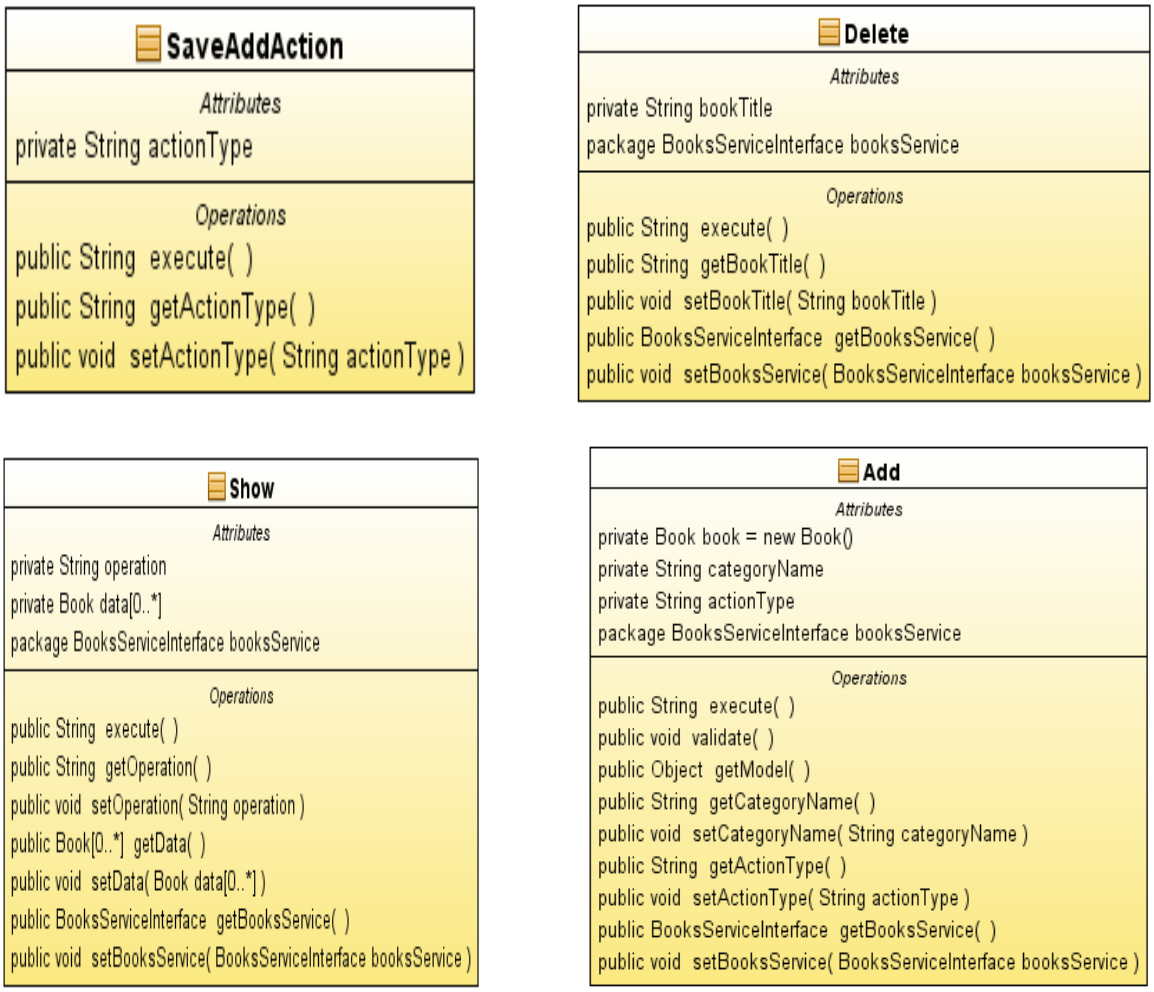


Figura 5.18 : Diagrame de clasa ale modului de logica a aplicatiei (pachetul categories)

5.2.1.4.3.2 Actiunile din pachetul books



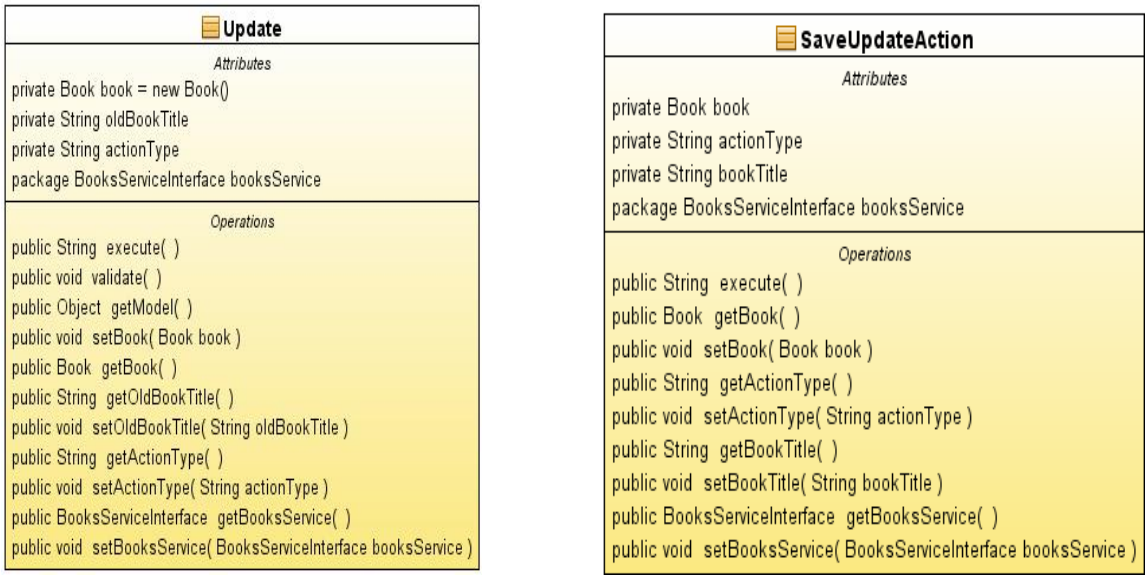


Figura 5.19 : Diagrame de clasa ale modului de logica a aplicatiei (pachetul books)

5.2.1.4.3.3 Actiunile din pachetul comments

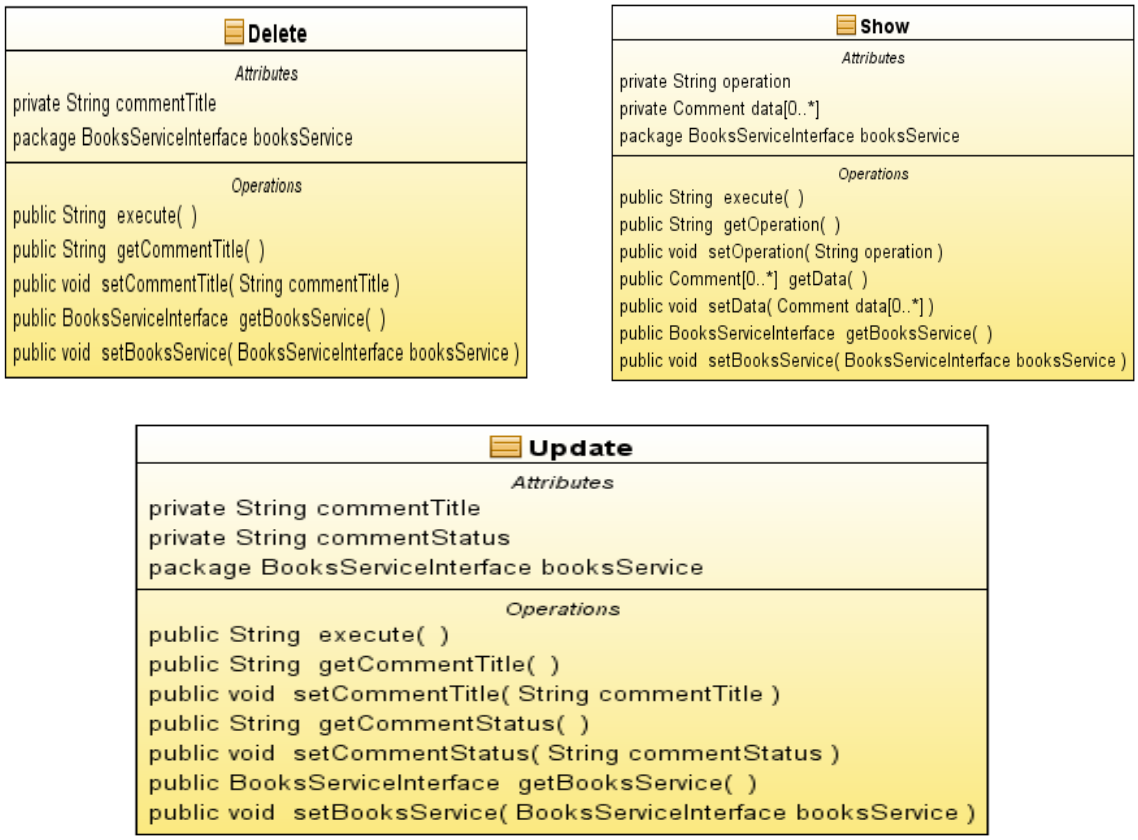


Figura 5.20 : Diagrame de clasa ale modului de logica a aplicatiei (pachetul comments)

5.2.1.4.3.4 Actiunile din pachetul orders

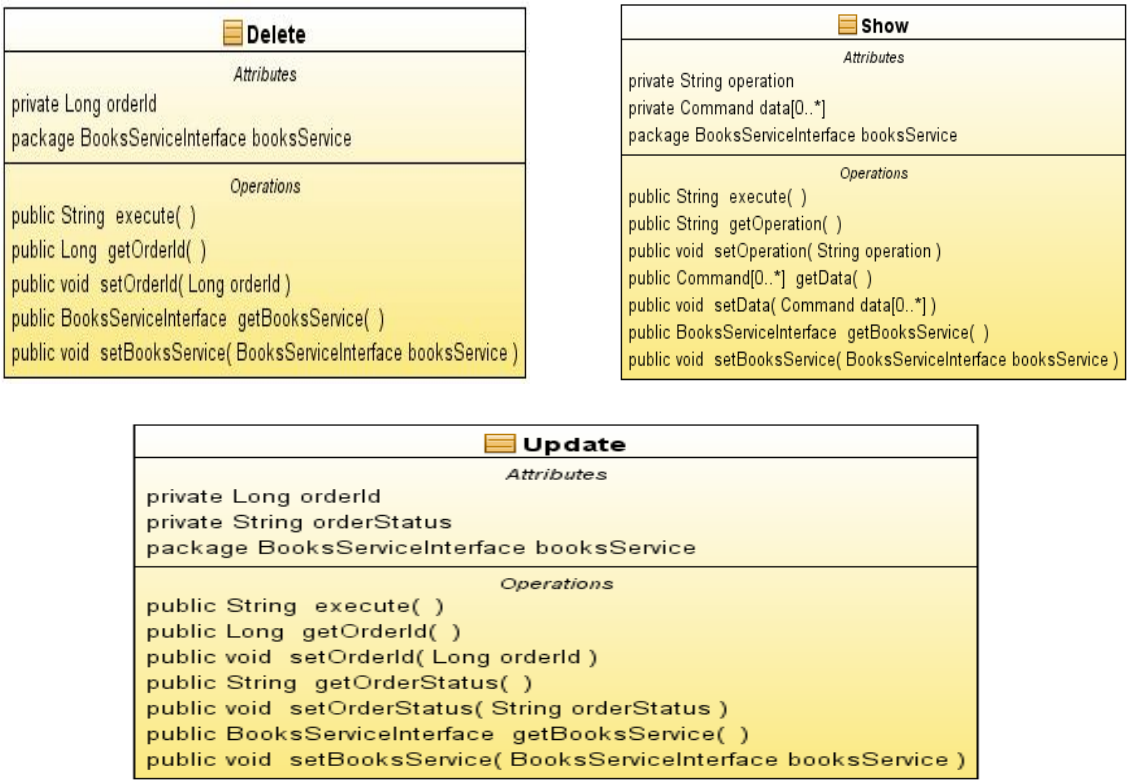


Figura 5.21 : Diagrame de clasa ale modului de logica a aplicatiei (pachetul orders)

5.2.1.4.3.5 Actiunile din pachetul users

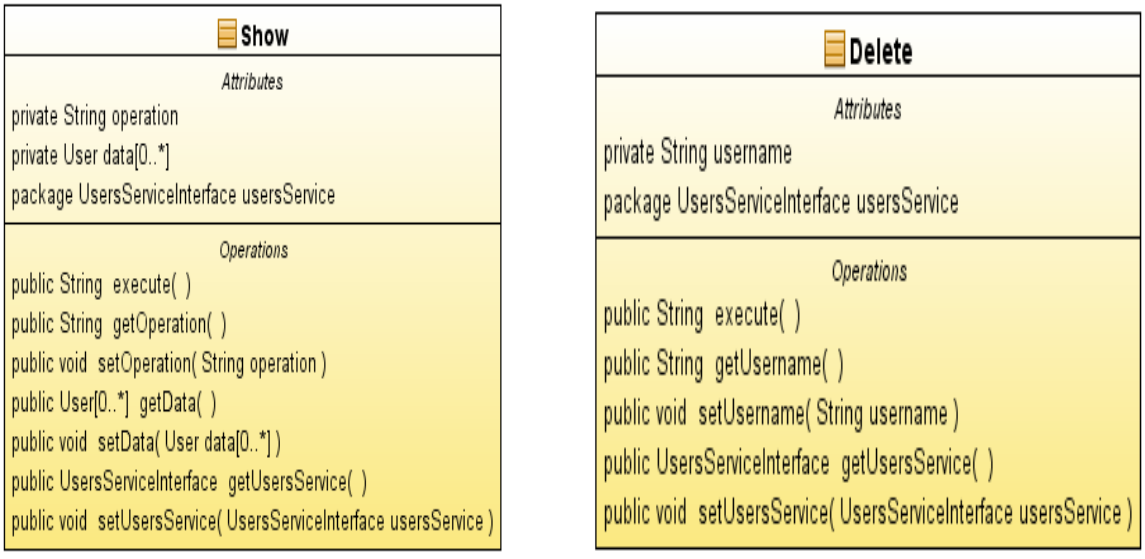


Figura 5.22 : Diagrame de clasa ale modului de logica a aplicatiei (pachetul users)

5.2.1.5 Interfete si clase aditionale

Clasele si interfetele aditionale sunt utilizate de actiunile aplicatiei, pentru realizarea unor functionalitati. Acestea sunt cuprinse in doua pachete, *validation* si *utils*.

Pachetul *validation* contine clasa *Check*, care este folosita de catre metoda *validate* a actiunilor, la verificarea datelor introduse de la tastatura. Pentru fiecare atribut al entitatilor introduse in baza de date a aplicatiei sunt definite cateva restrictii. Metodele clasei *Check* returneaza un String egal cu "OK", daca atributul curent indeplineste restrictiile impuse de programator. In caz contrar, String-ul contine un mesaj asociat cu restrictia care este incalcata. Daca in urma apelului metodelor clasei Check se gasesc erori, datele nu sunt adaugate in baza de date si utilizatorul este informat referitor la constrangerile nerespectate de informatia pe care a introdus-o de la tastatura.

Figura 5.23 : Diagrama de clasa a clasei pentru validarea datelor (Check)

Pachetul *utils* contine cinci subpachete si anume : *business*, *constants*, *enums*, *injections* si *interceptors*.

Subpachetul *business* cuprinde clasa *Cart*, care reprezinta cosul de cumparaturi virtual. Clasa contine un map cu cartile adaugate in cos. Pentru fiecare carte in parte, se retine si numarul de exemplare comandate. Clasa are metode care permit adaugarea sau eliminarea de carti din cosul de cumparaturi, dar si actualizarea numarului de exemplare ale unei carti comandate. De asemenea, se retine si costul total al comenzii, dar si numarul total de exemplare comandate.

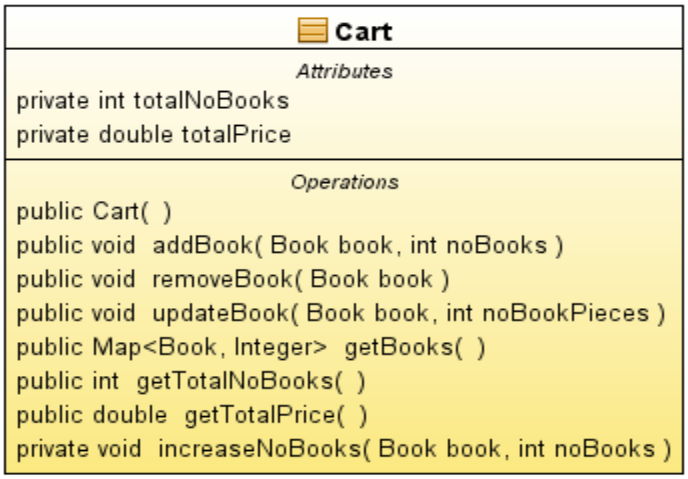


Figura 5.24 : Diagrame de clasa ale modulului de clase ajutatoare (clasa Cart)

Subpachetul *constants* contine clasa *CurrentDate*, care retine anul curent. Acesta este utilizat pentru validarea informatiei referitoare la anul publicarii unei carti, care nu trebuie sa fie mai mare decat anul curent.

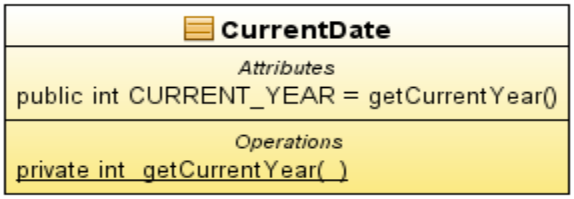


Figura 5.25 : Diagrame de clasa ale modulului de clase ajutatoare (clasa CurrentDate)

Subpachetul *enums* cuprinde enum-ul *SearchModes*, care retine toate criteriile posibile de cautare a unei carti (dupa titlu, autor, editura, categorie sau combinatii intre acestea). Criteriul de cautare a unei carti este transmis ca parametru metodei *searchBook* a interfetei *BooksServiceInterface* si, in functie de valoarea lui, se construieste interogarea prin care se cauta in baza de date cartile dorite.

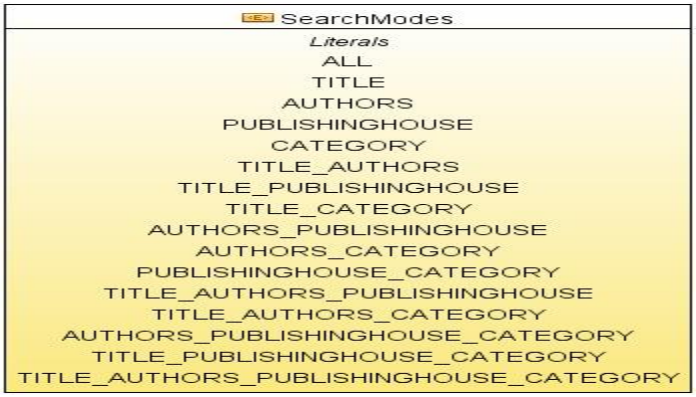


Figura 5.26 : Diagrame de clasa ale modulului de clase ajutatoare (clasa SearchModes)

Subpachetul *injections* contine interfata *UserAware*, care e utilizata de interceptorul *AutenticationInterceptor*, pentru a injecta user-ul curent in actiunea *SaveAccountsDetails*, a carei executie este precedata de executia interceptorului amintit anterior. Actiunea *SaveAccountsDetails* pune pe *ValueStack* datele user-ului la care are acces astfel, iar JSP-ul corespunzator (care creeaza fereastra ce permite updatarea contului unui user) actiunii le poate accesa direct de pe *ValueStack*, pentru a le afisa.

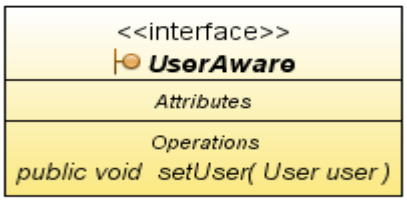


Figura 5.27 : Diagrame de clasa ale modulului de clase ajutatoare (interfata UserAware)

Subpachetul *interceptors* cuprinde interceptorul *AutenticationInterceptor*. Rolul acestuia este de a nu permite user-ului sa efectueze anumite actiuni daca nu este logat (nu-l lasa sa isi updateze contul si nici sa trimita comenzi, daca nu este logat). In cadrul metodei *intercept*, interceptorul obtine o referinta la map-ul sesiunii curente, dar si la actiunea urmatoare care va fi executata. Daca user-ul nu este logat si daca vrea sa isi updateze contul sau sa trimita o comanda, interceptorul nu permite executia acestor comenzi, ci returneaza un String, si anume un asa numit *rezultat global*, care reditrectioneaza user-ul spre actiunea de logare. Altfel, actiunile respective se pot executa, lucru posibil prin apelarea, din cadrul metodei *intercept*, a metodei *invoke*, pe obiectul care reprezinta actiunea curenta.

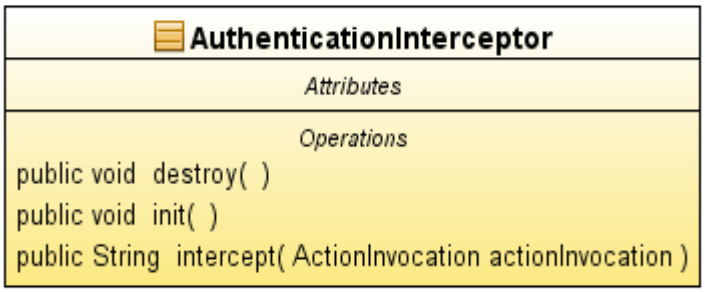


Figura 5.28 : Diagrame de clasa ale modulului de clase ajutatoare (clasa AuthentificationInterceptor)

5.2.2 Controller-ul aplicatiei

Controller-ul este pus la dispozitie de framework-ul Struts2 si este configurat in fisierul central al aplicatiei, *web.xml*. Fisierul *web.xml* este ilustrat la pagina 46 a lucrarii curente. Configurarea controller-ului este realizata in sectiunile a doua si a cincea ale acestui fisier. Controller-ul este un filtru Struts2, numit *struts2*, iar clasa care il implementeaza este *org.apache.struts2.dispatcher.FilterDispatcher*. Controller-ul se mapeaza pe toate cererile care vin de la clienti. Aceasta inseamna ca el preia aceste cereri si gaseste, pentru fiecare dintre acestea, actiunea asociata.

5.2.3 Vederile aplicatiei

Vederile aplicatiei sunt reprezentate de pagini JSP. Acestea constituie interfata aplicatiei cu clientii. Fiecarei actiuni ii corespunde o pagina JSP. Paginile JSP au acces la toate datele puse de actiuni pe *ValueStack*. Utilizandu-se structuri de control specifice framework-ului Struts2, datele de pe *ValueStack* sunt iterate, selectate si prelucrate, pentru ca apoi sa fie afisate in paginile JSP asociate. Modul in care informatia este afisata in pagini este definit de CSS-urile asociate acestora.

Exista trei foldere care contin fisierele JSP ale aplicatiei curente si anume : *common*, *user* si *admin*.

In folder-ul *common* este cuprinsa vederea comuna administratorului si user-ului : fereastra de logare (fisierul *LoginPage.jsp)* :

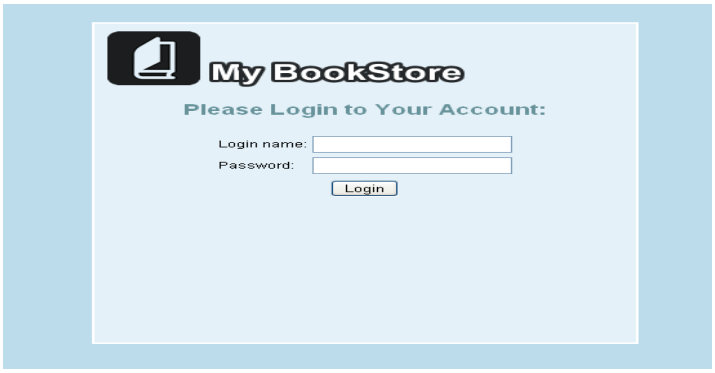
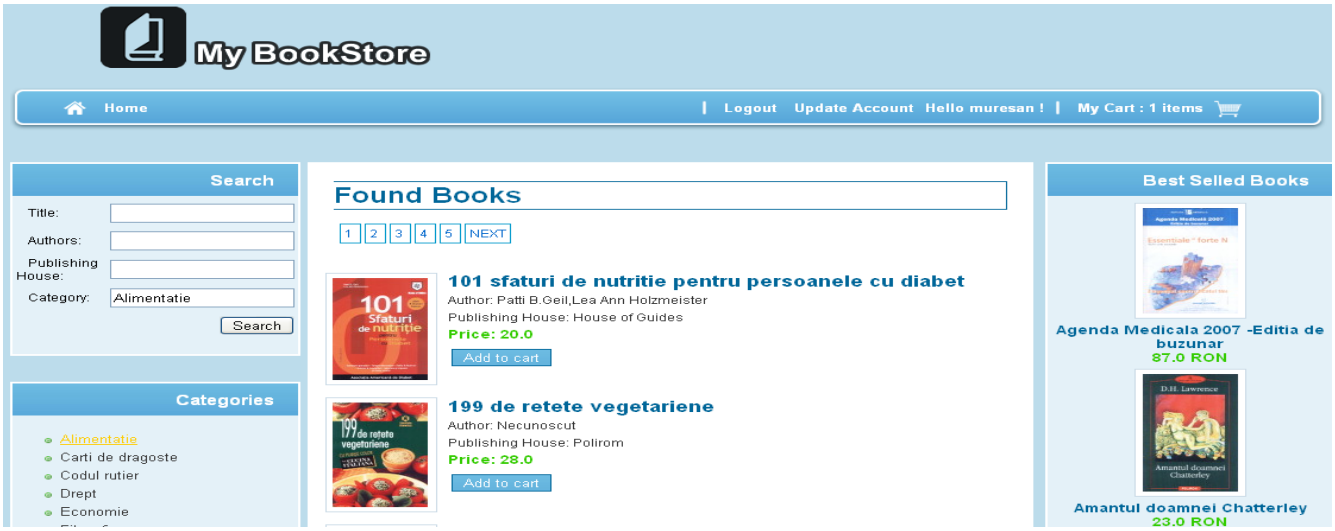


Figura 5.29 : Fereastra de logare a aplicatiei

In folder-ul *user* sunt cuprinse vederile specifice unui client obisnuit : fereastra de inregistrare in baza de date (*RegisterPage.jsp*), fereastra de updatare de cont (*UpdateAccountPage.jsp*), fereastra principala a user-ului (*UserMainPage.jsp*), fereastra care afiseaza cartile cautate (*UserBooksPage.jsp*), fereastra care afiseaza detalii despre o anumita carte si care ofera posibilitatea de a posta comentarii pe site (*UserBookDetailsPage.jsp*) si fereastra care permite efectuarea unei comenzi (*UserCartPage.jsp*).


My BookStore
Home
Logout Update Account Hello muresan ! My Cart : 1 items
Search
Title:
Authors:
Publishing House:
Category:
Search
Categories
Alimentatie
Carti de dragoste
Codul rutier
Drept
Economie
New Books
Agenda Medicala 2007 -Editia de buzunar
Author: Cornel Chirita
Publishing House: Medicala
87.0 RON
Add to cart
American Vertigo
Author: Bernard Henri Lvy
Publishing House: Nemira
35.0 RON
Add to cart
10 abordari psihoterapeutice ale depresiei
Best Selled Books
Agenda Medicala 2007 -Editia de buzunar
87.0 RON
Amantul doamnei Chatterley
23.0 RON


My BookStore
Home
Logout Update Account Hello muresan ! My Cart : 1 items
Search
Title:
Authors:
Publishing House:
Category: Alimentatie
Search
Categories
Alimentatie
Carti de dragoste
Codul rutier
Drept
Economie
Found Books
1 2 3 4 5 NEXT
101 sfaturi de nutritie pentru persoanele cu diabet
Author: Patti B.Geil,Lea Ann Holzmeister
Publishing House: House of Guides
Price: 20.0
Add to cart
199 de retete vegetariene
Author: Necunoscut
Publishing House: Polirom
Price: 28.0
Add to cart
Best Selled Books
Agenda Medicala 2007 -Editia de buzunar
87.0 RON
Amantul doamnei Chatterley
23.0 RON


My BookStore
Home
Logout Update Account Hello muresan ! My Cart : 1 items
Search
Title:
Authors:
Publishing House:
Category:
Search
Categories
Alimentatie
Carti de dragoste
Codul rutier
Drept
Economie
Filosofie
Informatica
Current Book:
Title: Agenda Medicala 2007 -Editia de buzunar
Author: Cornel Chirita
Publishing House: Medicala
Publishing Year: 2007
Price: 87.0
Description: Descriere agenda medicala
Add to cart
NO COMMENTS FOR THIS BOOK !
ADD A COMMENT
Title:
Text :
User:
Add Comment
Best Selled Books
Agenda Medicala 2007 -Editia de buzunar
87.0 RON
Amantul doamnei Chatterley
23.0 RON

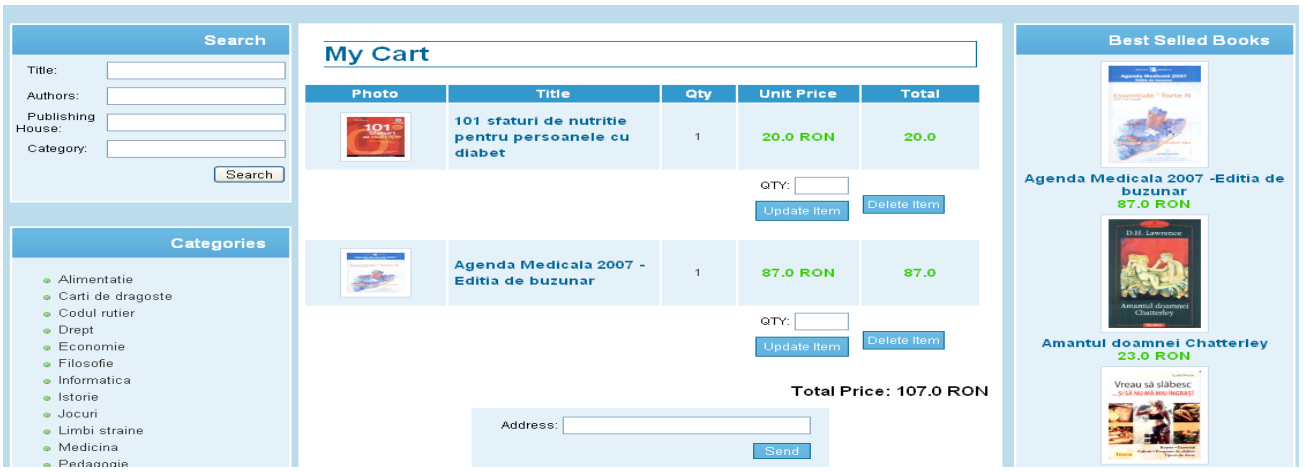
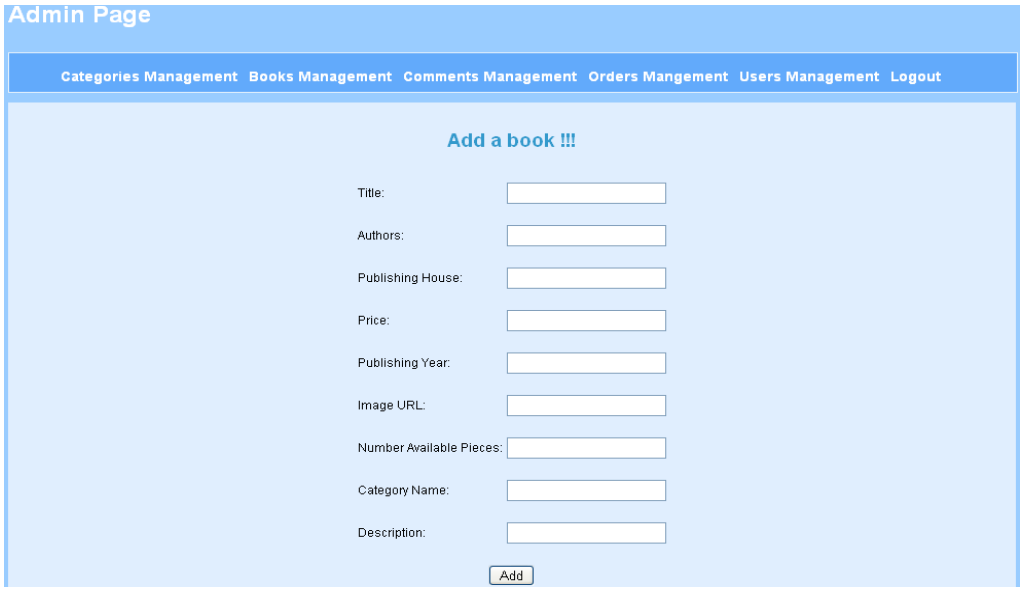


Figura 5.30 : Pagini accesibile vizitatorilor si utilizatorilor

In folder-ul *admin* sunt cuprinse vederile specifice administratorului site-ului : fereastra principala (*AdminPage.jsp*), fereastra care permite adaugarea cartilor in baza de date (*BookOperationsPage.jsp*), fereastra care permite adaugarea categoriilor in baza de date (*CategoryOperationsPage.jsp*). Administratorul poate efectua operatii pe toate entitatile din baza de date.



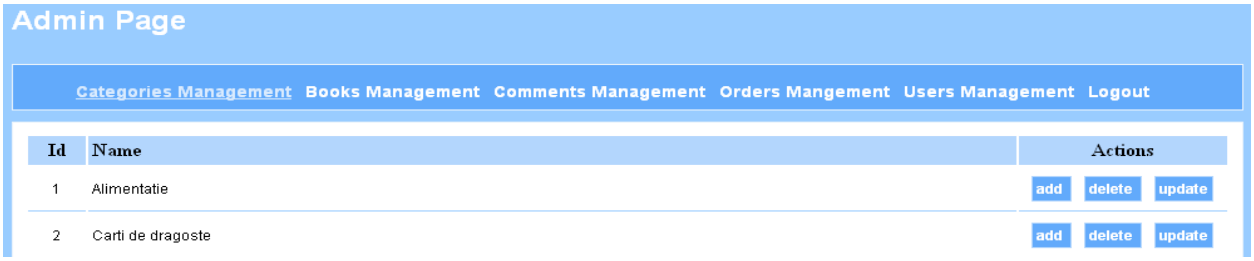


Figura 5.31 : Pagini accesibile administratorului

5.3 Functionarea sistemului

5.3.1 Diagrame de secventa

Diagramele de secventa sunt diagrame de interactiune care care pun in evidenta ordinea temporala a mesajelor. Aceste diagrame arata o multime de obiecte impreuna cu mesajele expediate si receptionate de catre acestea. [15]

Cererile clientilor pot fi de doua feluri : de tip GET (parametrii cererii sunt transmisi prin intermediul tag-ului *param*) sau de tip POST (parametrii cererii sunt transmisi cu ajutorul formelor de intrare, pe care clientii trebuie sa le completeze). Acestor tipuri de cereri le corespund doua diagrame de secventa. Aceste diagrame sunt formele generale pentru diagramele de secventa asociate cu cererile efectuate.

Forma generala a diagramelor de secventa asociate cu cererile de tip GET arata astfel :

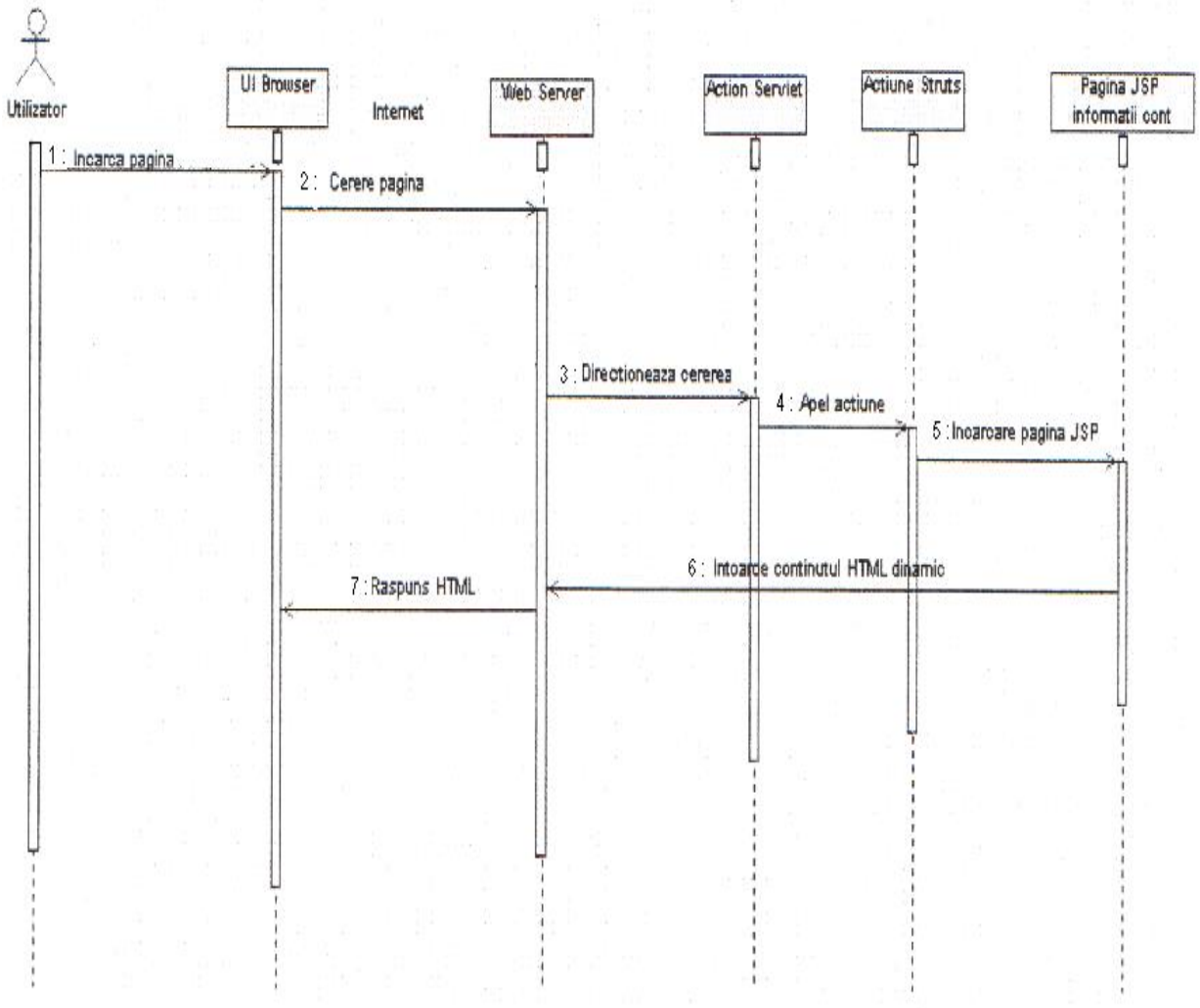


Figura 5.32 : Diagrama de secventa corespunzatoare cererilor de tip GET

Forma generala a diagramelor de secventa asociate cu cererile de tip POST arata astfel :

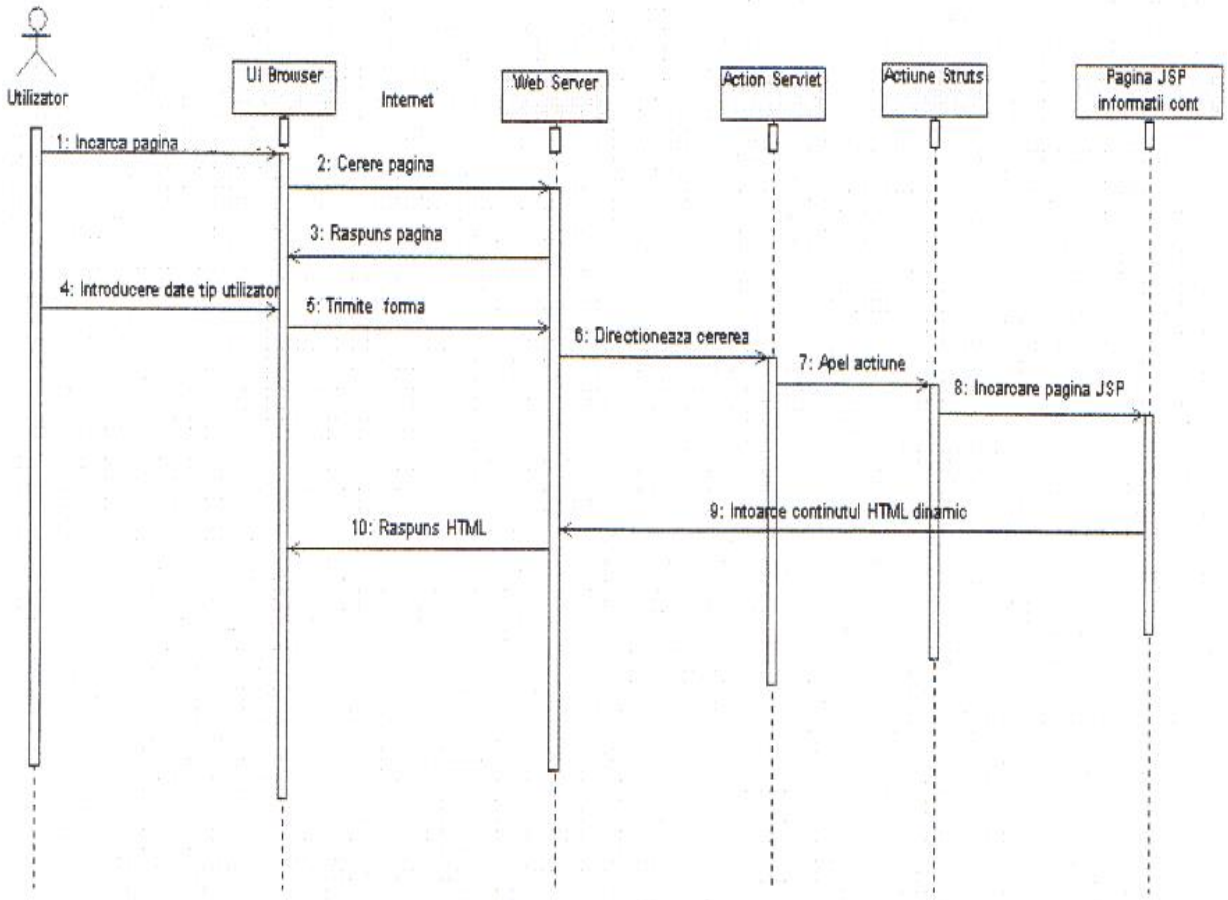


Figura 5.33 : Diagrama de secventa corespunzatoare cererilor de tip POST

5.3.2 Relatii intre componentele sistemului

In subcapitolul 5.2 s-au prezentat, individual, componentele aplicatiei Struts2, corespunzatoare sablonului arhitectural MVC, pe care il implementeaza. Aceste componente sunt legate printr-un fisier, *struts.xml*. Acesta este fisierul in care se configureaza actiunile si stivele de interceptori asociate acestora. Fiecarei actiuni ii este asociata o clasa Java, care o implementeaza, si unul sau mai multe rezultate, care constau din pagini JSP. Acestea afiseaza raspunsul dat de sistem la cererile clientilor. Raspunsul unei actiuni poate insemna si o redirectare spre alta actiune, nu numai afisarea unei pagini JSP. De fiecare data cand se efectueaza o cerere, filtrul *struts2* al aplicatiei cauta actiunea asociata acesteia, in fisierul *struts.xml*, si o executa. Apoi controlul este dat rezultatelor actiunii, care pot afisa un raspuns sau pot efectua o redirectare.

Fisierul *struts.xml* al aplicatiei curente contine doua pachete si anume : *library-public* si *library-secure*. In cadrul pachetului *library-public* sunt definite actiunile care pot fi executate, fara ca user-ul curent sa fie logat, dar si actiunile pe care le poate efectua administratorul. Pachetul *library-secure* contine actiunile pe care un user nu le poate efectua daca acesta nu este

logat (updatare cont si trimitere comanda). In afara acestor actiuni, in cadrul pachetului este redefinita stiva de interceptori, prin adaugarea la vechea stiva, a interceptorului definit in aplicatia curenta, cu numele *authenticationInterceptor* (care nu permite executia actiunilor din pachetul curent, daca user-ul nu este logat). Apoi sunt declarate si doua rezultate globale, *login-updateAccount* si *login-sendOrder*, corespunzatoare celor doua actiuni ale pachetului. Interceptorul *authenticationInterceptor* returneaza, pentru fiecare din cele doua actiuni, cate o valoare egala cu numele acestor rezultate globale, daca se incearca executia acestor actiuni fara ca user-ul sa fie logat. Aceste rezultate globale au rolul de a redirecta user-ul spre actiunea de logare, transmitand totodata, printr-un parametru, si actiunea la care trebuie sa se revina, daca user-ul se logheaza cu succes. Mai jos este ilustrat un fragment din pachetul *library-secure*, al fisierului *struts.xml* :

```
<package name = "library-secure" namespace = "/" extends = "struts-default">

    <interceptors>

        <interceptor name = "authenticationInterceptor" class = "library.model.utils.interceptors.AuthenticationI

            <interceptor-stack name = "secureStack">

                <interceptor-ref name = "authenticationInterceptor"/>
                <interceptor-ref name = "defaultStack"/>

            </interceptor-stack>

    </interceptors>

    <default-interceptor-ref name="secureStack"/>

    <global-results>
        <result name = "login-updateAccount" type = "redirect">
            Pre-Login.action?actionType=updateAccount
        </result>
        <result name = "login-sendOrder" type = "redirect">
            Pre-Login.action?actionType=sendOrder
        </result>
    </global-results>

    <action name = "Pre-UpdateAccount" class = "library.model.actions.user.SaveAccountDetails">
```

6. Instalare si testare

In primul rand, trebuie instalat produsul JDK 6.0, care se gaseste la adresa http://java.sun.com. Pentru instalarea acestui produs, se recomanda stergerea si dezinstalarea versiunilor anterioare ale acestuia.

Tomcat este server-ul cel mai popular din punct de vedere al suportului pentru servlet-uri si pagini JSP, fiind totodata si server-ul de referinta in acest sens. Pentru aplicatia curenta, am utilizat versiunea 6.0 a acestuia. Server-ul Tomcat poate fi descarcat de la adresa http://tomcat.apache.org/. Se instaleaza arhiva intr-un director numit *Tomcat 6.0*. Pentru pornirea server-ului, se va rula comanda *tomcat6.exe*, din subdirectorul *bin* al directorului *Tomcat 6.0*. Pentru a testa faptul ca server-ul a fost instalat si functioneaza corect, se acceseaza dintr-o fereastra browser de pe masina curenta urmatoarea adresa : http://localhost:8080. Daca se incarca corect pagina implicita a server-ului Tomcat, atunci instalarea a fost facuta cu succes. [12]

Pentru partea de baze de date a aplicatiei, trebuie instalata tehnologia MySQL. Pentru acest proiect, trebuie descarcata arhiva pentru versiunea 5.0 a acesteia, de la adresa dev.mysql.com/downloads/. Arhiva respectiva este instalata pe disc. La instalarea MySQL, se seteaza o parola cu ajutorul careia se va accesa MySQL, din linia de comanda. Dupa instalarea MySQL, se creeaza o baza de date numita *library*.

Pentru partea de client, trebuie sa existe pe calculator unul din browser-ele : Mozzila Firefox (de la versiunea 3.0.8 in sus), Internet Explorer (de la versiunea 8 in sus), Google Chrome.

Livrarea aplicatiei se face un forma unei arhive *CataInAction.war* (Web Archive). Aceasta arhiva contine aplicatia web care va fi instalata in server-ul Tomcat. Procesul de instalare al unei astfel de arhive este simplu. Se copiaza in directorul Tomcat 6.0\webapps\. La pornirea server-ului Tomcat, aplicatia este dezarhivata automat in structura de directoare din arhiva.

Rularea aplicatiei se efectueaza in doua faze : se porneste server-ul Tomcat si apoi se porneste unul din browser-ele anterioare, cu comanda http://localhost:8080/CataInAction. Va apare prima pagina a user-ului. Daca se doreste rularea in modul administrator, se acceseaza fereastra de logare. Se completeaza user-ul *admin* si parola *admin* si se intra astfel in modul administrator.

7. Concluzii

7.1 Realizari

Se constata ca s-au indeplinit cerintele functionale si non-functionale impuse. Aplicatia creata are o parte de utilizator si una de administrator. Pentru fiecare dintre aceste parti s-au implementat toate cerintele functionale definite in capitolul al doilea al documentului curent.

Cerintele sunt implementate corect, iar aplicatia nu are erori. Aplicatia este distribuita, mai multi clienti de pe masini diferite putand vizualiza site-ul creat. Validitatea aplicatiei este asigurata de catre metodele de validare care nu permit clientului sa introduca date incorecte in sistem. Sistemul este stabil, sigur. Pentru partea de administrator, dar si pentru anumite operatii ale utilizatorului, clientul trebuie sa se autentifice. Aplicatia este scalabila, deoarece arhitectura creata permite separarea diferitelor componente ale acesteia (model, vizualizare, control). Astfel, se poate rescrie o componenta foarte usor, utilizand alte tehnologii (de exemplu componenta de vizualizare). Asadar, si cerintele non-functionale definite la inceput sunt implementate corect.

7.2 Directii de dezvoltare

Aplicatia ar putea fi imbunatatita in cateva moduri. Pot fi aduse modificari in special componentei de vizualizare, dar pot fi adaugate si cateva noi functionalitati.

Pe partea de vizualizare, s-ar putea folosi plugin-uri specifice Struts2, sau alte tehnologii decat JSP, tag-uri Struts2 si CSS (*Velocity*, *FreeMarker* etc). De asemenea, s-ar putea integra componente JSF in aplicatia curenta, pentru a face aspectul vizual mai frumos. Apoi, s-ar putea utiliza AJAX (Asynchronous JavaScript and XML) pentru partea care realizeaza interactiunea cu un utilizator obisnuit. AJAX este o tehnica prin care se pot crea mai bine si mai repede aplicatii web mai dinamice, mai rapide si mai prietenoase. Cu aceasta tehnologie, scriptul JavaScript poate comunica direct cu server-ul, utilizand obiectul JavaScript *XmlHttpRequest*. Asadar se pot schimba date cu un server web, fara a reincarca pagina. Se utilizeaza cereri HTTP asincrone si au loc transferuri asincrone de date intre browser si server-ul web, permitandu-se ca paginile web sa solicite cantitati mai mici de informatie de la server, in loc sa ceara ca toata pagina sa fie reincarcata. AJAX este bazata pe JavaScript, XML, HTML si CSS. Un alt avantaj al acestei tehnologii este faptul ca e independenta de softul in care e scris server-ul web, de browser si de platforma utilizata.

Pot fi adaugate si cateva noi functionalitati. Pentru partea de utilizator, pot fi create rating-uri pentru fiecare carte a site-ului. De asemenea, se poate crea posibilitatea, pentru administrator, de a trimite e-mail-uri utilizatorilor site-ului cand apare o modificare in baza de date a acestuia. Apoi, pot fi create mecanisme de cautare mai complexe a entitatilor bazei de date, la partea de administrator.

Scopul acestei lucrari a fost de a exemplifica modalitatea de creare a unei aplicatii distribuite web, in Struts2, utilizand elementele de baza ale framework-ului. Se poate imbunatati si extinde implementarea aplicatiei, utilizand alte facilitati mai avansate ale framework-ului, multe dintre ele amintite in lucrarea de fata.

8. Bibliografie

[1] **Bryan Basham**, **Kathy Sierra**, **Bert Bates**, "*Head First Servlets and JSP Second Edition*", Editura O'Reilly, 2008

[2] **Donald Brown**, **Chad Michael Davis**, **Scott Stanlick**, "*Struts2 in Action*", Editura Manning, 2008

[3] **Bill Burke**, **Richard Monson-Haefel**, "*Enterprise Java Beans, 3.0*", Editura O'Reilly, 2006

[4] **Robert Dollinger**, **Luciana Andron**, "*Baze de date si gestiunea tranzactiilor*", Editura Albastra, Cluj-Napoca, 2004

[5] **Bruce Eckel**, "*Thinking in Java Fourth Edition*", Editura Prentice Hall, 2006

[6] **Simona Preda**, **Titi Preda**, "*HTML pentru World Wide Web cu XHTML si CSS*", Editura Corint, 2003

[7] **Ioan Salomie**, **Tudor Cioara**, **Ionut Anghel**, **Tudor Salomie**, "*Distributed computing and systems*", Editura Albastra, Cluj-Napoca, 2008

[8] **Struts2**, *http://www.roseindia.net/struts/struts1-vs-struts2.shtml*

[9] **Understanding the Client-Server Architecture**,

http://www.webdevelopersnotes.com/basics/client_server_architecture.php3

[10] **HTML**, *http://www.w3schools.com*

[11] **MySQL**, *http://www.mysql.com*

[12] **Apache Tomcat**, *http://www.apache.org*

[13] **Sun Microsystems**, *http://www.sun.com*

[14] **Understanding the Three-Tier Architecture**, http*://www.oracle.com/technology/products/ias/toplink/doc/1013/main/_html/undtldev010.htm*

[15] **Diagrame UML**, *http://www.cs.ubbcluj.ro/~tzutzu/Didactic/AnalizaGestSisteme*

9. Glosar

AJAX (Asynchronous JavaScript and XML) este o combinatie de tehnologii (HTML, CSS, JavaScript) pentru crearea de aplicatii web interactive

API (Application Programming Interface) defineste modul in care o aplicatie acceseaza functionalitatea unei alte aplicatii

CGI (Common Gateway Interface) reprezinta un standard pentru comunicatii intre aplicatii externe si servere de informatii precum HTTP si servere web

CSS (Cascading Style Sheets) reprezinta un limbaj pentru specificarea modului de formatare a elementelor intr-o pagina web

FTP (File Transfer Protocol) reprezinta un protocol pentru transferul de fisiere intre diverse calculatoare

HTML (Hyper Text Markup language) reprezinta un limbaj de marcare care are doua caracteristici : hipertextul si universalitatea

HTTP (Hyper Text Transfer Protocol) reprezinta un protocol pentru transferul diverselor fisiere (text, sunet, imagine etc) care sta la baza World Wide Web

J2EE (Java 2 Enterprise Edition) reprezinta standardul Java pentru dezvoltarea aplicatiilor industriale pe mai multe nivele, bazate pe componente

J2ME (Java 2 Micro Edition) reprezinta un set de tehnologii si specificatii Java pentru realizarea de aplicatii pentru telefoane mobile, PDA-uri, imprimante, etc

J2SE (Java 2 Standard Edition) reprezinta mediul standard pentru realizarea aplicatiilor Java si reprezinta platforma de baza pentru celelalte tehnologii Java

JAR (Java Archive) reprezinta o arhiva de tip zip in care se pastreaza fisierele java de tip .class

JPA (Java Persistence API) este un API pus la dispozitie de limbajul Java, care permite acesarea bazelor de date relationale prin elemente ale programarii orientate obiect

JSF (Java Server Faces) este o tehnologie Java pentru dezvoltarea componentelor pe partea de server, bazata pe conceptele introduse de JSP-uri, care faciliteaza realizarea interfetelor web

JSP (Java Server Pages) reprezinta componente Java pe partea de server, care extind functionalitatea servlet-urilor, permintand inserarea codului Java in cadrul paginilor web

JVM (Java Virtual Machine) reprezinta masina virtuala Java, care are rolul de a tranforma byte-code-ul programelor Java in instructiuni specifice masinii pe care acesta e executat

MVC (Model View Controller) reprezinta un design pattern care separa logica aplicatiei de nivelul de prezentare a datelor si de procesare a cererilor

OGNL (Object-Graph Navigation Language) este o tehnologie care, utilizand expresii simple, permite setarea si accesul proprietatilor bean-urilor Java si executia metodelor Java

SGBD reprezinta un sistem de gestiune a fisierelor unei baze de date, independent de aplicatiile care o utilizeaza

SMTP (Simple Mail Transfer Protocol) este un protocol de nivel aplicatie, care permite trimiterea de e-mail-uri

TCP (Transmission Control Protocol) este un protocol de nivel Transport, orientat pe conexiune, care are ca principala sarcina asigurarea unei transmisii sigure a informatiei, in cadrul retelei, sub forma de pachete

UML (Unified Modelling Language) reprezinta limbajul standard pentru analiza si design-ul aplicatiilor precum si pentru specificarea structurii si a comportamentului sistemului

URL (Uniform Resource Locator) reprezinta adresa unei resurse pe internet, fiind de obicei o adresa World Wide Web

WAR (Web Archive) reprezinta o arhiva zip, care contine o aplicatie web

XML (Extensible Markup Language) reprezinta un standard deschis al consortiului World Wide Web proiectat pentru interschimbarea documentelor structurate pe web

10. Anexa

Lista Figurilor