****

**FACULTATEA DE AUTOMATICA SI CALCULATOARE**

**SISTEME DISTRIBUITE**

**Assignment 2.2**

RPC application using distributed objects

**Profesor Indrumator: Student:**

**Cioara Tudor Bucea Valentina Sorina**

**Grupa 30244**

CUPRINS

[1. Arhitectura conceptuala a sistemului distribuit 3](#_Toc466818846)

[2. Diagrama UML de implementare 5](#_Toc466818848)

[3. Readme file 6](#_Toc466818849)

[Bibliografie 8](#_Toc466818850)

# Arhitectura conceptuala a sistemului distribuit

Aceasta aplicatie se structureaza pe o arhitectura de tip client-server, in care clientul solicita anumite informatii, urmand ca serverul sa ii raspunda cererii. Clientul nu impartaseste niciuna dintre resursele sale, doar solicita anumite functii de continut din partea serverului, in functie de necesitatile sale. Prin urmare, clientii initiaza sesiuni de comunicare cu serverele care asteapta sa primeasca request-uri.

Comunicarea intre client si server se stabileste pe baza apelurilor de metode in mod remote (RPC), folosindu-se obiecte distribuite. Aplicatiile *Remote Method Invocation* cuprind adesea doua programe separate, cum este si in cazul proiectului creat de mine, unul fiind partea de server, iar celalalt partea ce se ocupa de afisarea unei interfete clientului si apelarea metodelor din server, pentru a se obtine taxa sau pretul de vanzare al unei masini.

Un program de server tipic crreaza unele obiecte de la distanta, facand referire la aceste obiecte accesibile, si asteapta clientii sa invoce metode pe aceste obiecte create. Un program client tipic obtine o referinta la distanta, la unul sau mai multe obiecte aflate pe server si apoi invoca metode pe ele. RMI prevede un mecanism prin care serverul si clientul comunica si transmite informatii inainte si inapoi.

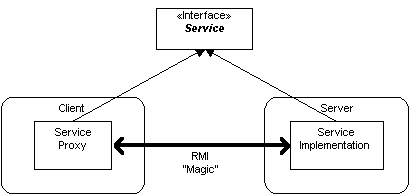
Aplicatia creata de mine poate fi mentionata ca si o aplicatie obiect distribuit deoarece urmareste urmatoarele:

* Localizeaza obiectele la distanta – Aplicatia utilizeaza diverse mecanisme pentru a obtine referinte la obiectele aflate la distanta: inregistreaza obiectele sale folosind registrul RMI
* Comunica cu obiectele aflate la distanta – Detaliile de comunicare intre obiecte la distanta sunt tratate de RMI, pentru dezvoltator, comunicarea in mod remote, pare similara cu invocarea unor metode Java
* Incarca definitia claselor pentru obiectele ce sunt procesate – Deoarece RMI permite obiectelor sa fie trecute inainte si inapoi, acesta prevede un mecanism de incarcare a definitiilor clasei unui obiect, precum si pentru transmiterea datelor unui obiect

Unele dintre caracteristicile centrale si unice ale RMI-ului este capacitatea sa de a descarca definitia clasei unui obiect in cazul in care clasa nu este definita in masina virtuala a receptorului Java. Toate tipurile si comportamentul unui obiect, disponibile anterior numai intr-o singura masina virtuala Java, poate fi transmis la o alta masina virtuala Java, eventual de la distanta. RMI-ul trece obiectele prin clasele lor reale, astfel incat comportamentrul obiectelor nu se modifica atunci cand sunt trimise catre o alta masina virtuala Java. Aceasta capacitate permite noi tipuri si comportamente care urmeaza sa fie introduse intr-o masina virtuala Java, de la distanta, extinzand astfel in mod dinamic comportamentul aplicatiei.

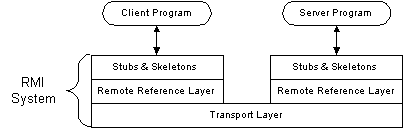
La fel ca orice alta aplicatie Java, o aplicatie distribuita construita prin utilizarea Java RMI-ului este formata din interfete si clase. Interfetele de *IPriceService* si *ITaxService* declara metode ce sunt apelate de client si implementate de server pentru a constitui functionalitatea intregii aplicatii. Un obiect devine obiect la distanta prin implementarea unei interfete la distanta, care are urmatoarele caracteristici: O interfata de la distanta extinde interfata *java.rmi.Remote* si fiecare metoda a interfetei declara *java.rmi.RemoteException* in clauza aruncarii, in plus fata de orice exceptii specifice aplicatiei. RMI trateaza un obiect la distanta in mod diferit fata de un obiect de baza non-la distanta, atunci cand obiectul este transmis de la o JVM la o alta JVM. Mai degraba decat a face o copie a implementarii obiectului in JVM-ul receptor, RMI-ul trece un *Stub* la distanta pentru un obiect aflat la distanta. *Stub* actioneaza in calitate de reprezentant local sau proxy, pentru obiectul de la distanta si, practic este, pentru client, referinta la distanta. Clientul invoca o metoda aflata pe *Stub*-ul local, care este responsabil pentru efectuarea invocarii metodepei pe obiectul la distanta. Un *Stub* pentru un obiect aflat la distanta implementeaza acelasi set de interfete de la distanta, ca si obiectul in sine. Aceasta proprietate permite ca un *Stub* sa fie exprimat sub forma oricarei interfete pe care obiectul o implemeteaza (cast). Cu toate acestea, numai acele metode definite intr-o interfata de la distanta sunt disponibile pentru a fi apelate de receptorul JVM.

*Stub* si *Skeleton* ascund dezvoltatorului detaliile de comunicare. *Stub* este clasa care implementeaza interfata la distanta, acesta servind ca si substituent al partii clientului pentru obiectul aflat la distanta. El comunica cu *Skeleton-ul*, ce se afla pe partea serverului, fiind omologul sau, acestia comunicand prin intermediul retelei. *Skeleton* stie adevaratii delegati a obiectelor aflate la distanta, pe care *Stub*-ul ii solicita, returnandu-i raspunsul potrivit. Cand se invoca o metoda din partea *Stub-*ului, se executa urmatoarele: se initiaza o conexiune cu JVM-ul aflat la distanta, care contine obiectul de la distanta; scrie si transmite parametrii JVM-ului aflat la distanta; asteapta rezultatul invocarii metodei; citeste valoarea de returnare sau exceptia returnata si returneaza valoarea apelantului. *Stub* ascunde serializarea parametrilor si a comunicarii la nivel de retea, in scopul prezentarii unui mecanism simplu de invocare a apelantului. *Skeleton* este responsabil pentru expedierea apelului la obiectul de la distanta ce ofera implementarea cuvenita. Atunci cand el primeste o metoda de invocare se realizeaza urmatoarele: citeste parametrii pentru metoda de la distanta; invoca metoda privind punerea in aplicare a implementarii obiectului de la distanta; noteaza si transmite rezultatul (valoarea de returnare sau de exceptie) apelantului. In Java 2 SDK Standard Edition v1.2 a fost introdus un protocol aditional de tip *Stub*, ce a elimina necesitatea *Skeleton-ului*. In schimb, este folosit codul generic pentru a indeplini sarcinile realizate de *Skeleton* in JDK1.1. *Stub* si *Skeleton* sunt generate de compilatorul rmic. Incepand cu JDK1.2, clasele de implementare server deleaga task-urile unui *Dispatcher*, implementand *reflection*, mai degraba decat folosind o clasa *Skeleton. Stub* si *Dispatcher* comunica prin protocolul *Java Remote Method Protocol* (JRMP), incorporat in *HTTP*, care este construit pe partea de sus a *TCP/IP*-ului.

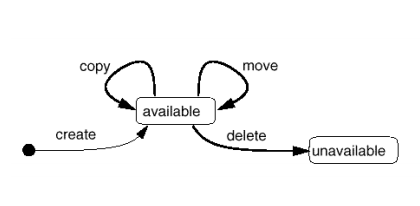
Arhitectura sistemului distribuit se bazeaza pe un principiu important: definirea comportamentului si punerea in aplicare a acestui comportament sunt concepte separate. RMI permite codului care defineste comportamentul si codului care implementeaza comportamentul sa ramana separat si sa ruleze pe JVM-uri separate. Acest lucru se potriveste foarte bine cu nevoile unui sistem distribuit, in cazul in care clientii sunt preocupati de definitia unui serviciu si serverele sunt axate pe furnizarea serviciului. Mai exact, in RMI, definitia unui serviciu de la distanta este codat folosind o interfata Java. Implementarea serviciului aflat la distanta este codat intr-o clasa, prin acest lucru, intelegerea RMI-ului se reduce la amintirea faptului ca interfetele definesc comporamentul si clasele definesc punerea in aplicare, implementarea. RMI-ul sustine doua clase care implementeaza aceeasi interfata: prima clasa este punerea in aplicare a comportamentului si ruleaza pe server, iar cea de-a doua actioneaza ca un proxy pentru serviciul de la distanta, ruland pe client.

[RMI Architecture 
Layers]

Arhitectura aplicatiei este construita, in esenta, din trei nivele logice, abstractizate. Primul este nivelul *Stub*-ului si al *Skeleton*­-ului, care se afla chiar sub punctul de vedere al apelurilor metoda, la nivelul dezvoltatorului. Acest nivel interpreteaza apelurile metodelor realizate de client la referinta variabilelor de interfata si redirectioneaza apelurile serviciilor RMI la distanta. Urmatorul nivel este *Remote Reference Layer*, acesta intelegand cum sa interpreteze si sa gestioneze referintele facute de clienti serviciilor aflate la distanta. Ultimul nivel este cel de transport, bazandu-se pe conexiuile *TCP/IP* intre masinile dintr-o retea. Acesta ofera conectivitate de baza intre masinile virtuale Java, precum si unele strategii de penetrare firewall.



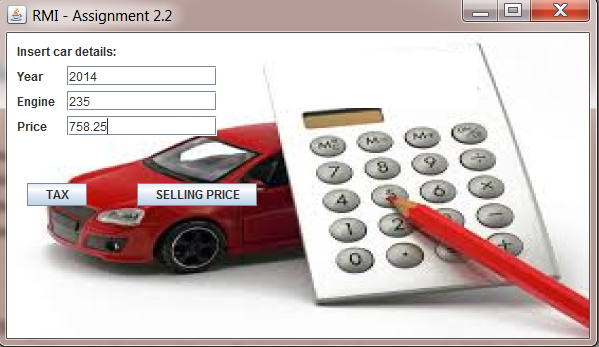
# Obiectele locale si distribuie difera in mai multe privinte: ciclul de viata – crearea, migrarea si stergerea obiectelor distribuite este mult mai costisitoare ca a celor locale; referinta – trimiterile la distanta a obiectelor distribuite sunt mult mai complexe decat indicii simple la adresele de memorie, acestea necesita informatii locale, de securitate si referinte la tipurile obiectelor; latenta – o cerere de obiect distribuit implica un grad ridicat de latenta decat invocarea metodei locale; activarea obiectelor – obiectele distribuite pot sa nu fie intotdeauna disponibile pentru a servi o cerere deobiect, in orice moment de timp; paralelism – obiectele distribuite pot fi executate in paralel; comuicarea – exista diferite metode primitive de comunicare ce pot fi folosite pentru cererile obiectelor distribuite; eroarea – obiectele distribuite au mai multe puncte de esec decat obiectele tipice, locale; securitatea – distributia le face vulnerabile la atacuri.

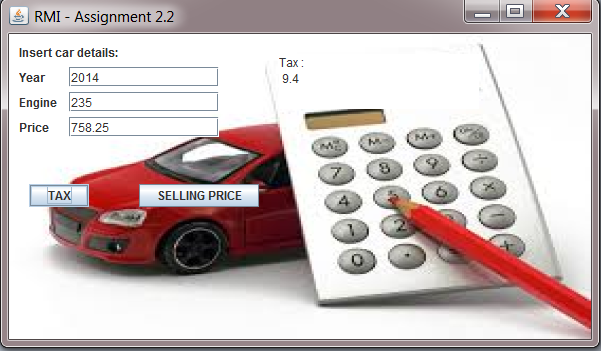


# Diagrama UML de implementare

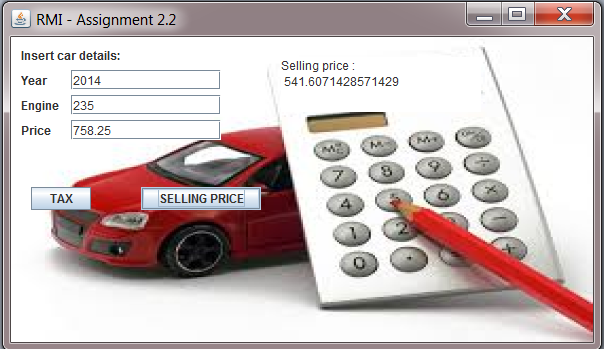
# Readme file

* Seteaza GIT-ul si descarca proiectul de pe <https://SorinaVB@bitbucket.org/SorinaVB/ds2016.git>
  + Creaza un folder local gol in spatiul de lucru de pe calculatorul tau
  + Click dreapta in folder si selecteaza Git Bash
  + Introdu comenzile:
* git init
* git remote add origin <https://SorinaVB@bitbucket.org/SorinaVB/ds2016.git>
* git pull origin master
* Importa proiectul in Eclipse
  + FILE-> Import->Maven-> Existing Maven Projects-> Navigheaza pana la folder-ul cu proiectul descarcat
* Ruleaza proiectul
* click dreapta pe proiectul Assignment2\_2Server: Run As -> Java Application
* click dreapta pe proiectul Assignment2\_2Client: Run As -> Java Application
* Introdu specificatiile masinei pentru care vrei sa aflii taxa si pretul de vanzare





* Apasa butonul *TAX* pentru a vizualiza taxa masinii ce are detaliile introduse de tine
* Selecteaza *SELLING PRICE* pentru a afla ce pret de vanzare poti castiga pentru masina introdusa



# Bibliografie

* <https://docs.oracle.com/javase/tutorial/rmi/overview.html>
* <https://www.cs.uic.edu/~troy/fall04/cs441/drake/rmi.html>
* <http://docstore.mik.ua/orelly/java-ent/security/ch06_01.htm>
* <http://dsnet.tu-plovdiv.bg/website/container/rmi/index.html>