Proiect 1 – PPD

Analiza cerintelor

O sala de spectacole vinde bilete la spectacolele organizate printr-o aplicatie client-server. Sala organizeaza cel mult un spectacol pe zi si are un numar maxim de locuri numerotate de la 1.

Pentru fiecare spectacol avem informatii de tip (data, titlu, pret_bilet).

Permanent sala mentine o evidenta actualizata pentru:

- Informatii despre bilete pentru fiecare spectacol (ID_spectacol, lista_locuri_vandute)
- Vanzarile efectuate, o vanzare = (data_vanzare, ID_spectacol, numar_bilete, lista_locurilor)
- Soldul total

Periodic sistemul face o verificare a locurilor vandute prin prin verificarea corespondentei corecte intre locurile libere si vanzarile facute, sumele incasate per vanzare si soldul total. Rezultatul trebuie salvat pe suport extern (fisier text).

Serverul se inchide dupa un interval de timp precizat si notifica clientii activi referitor la inchidere.

Cerinte functionale:

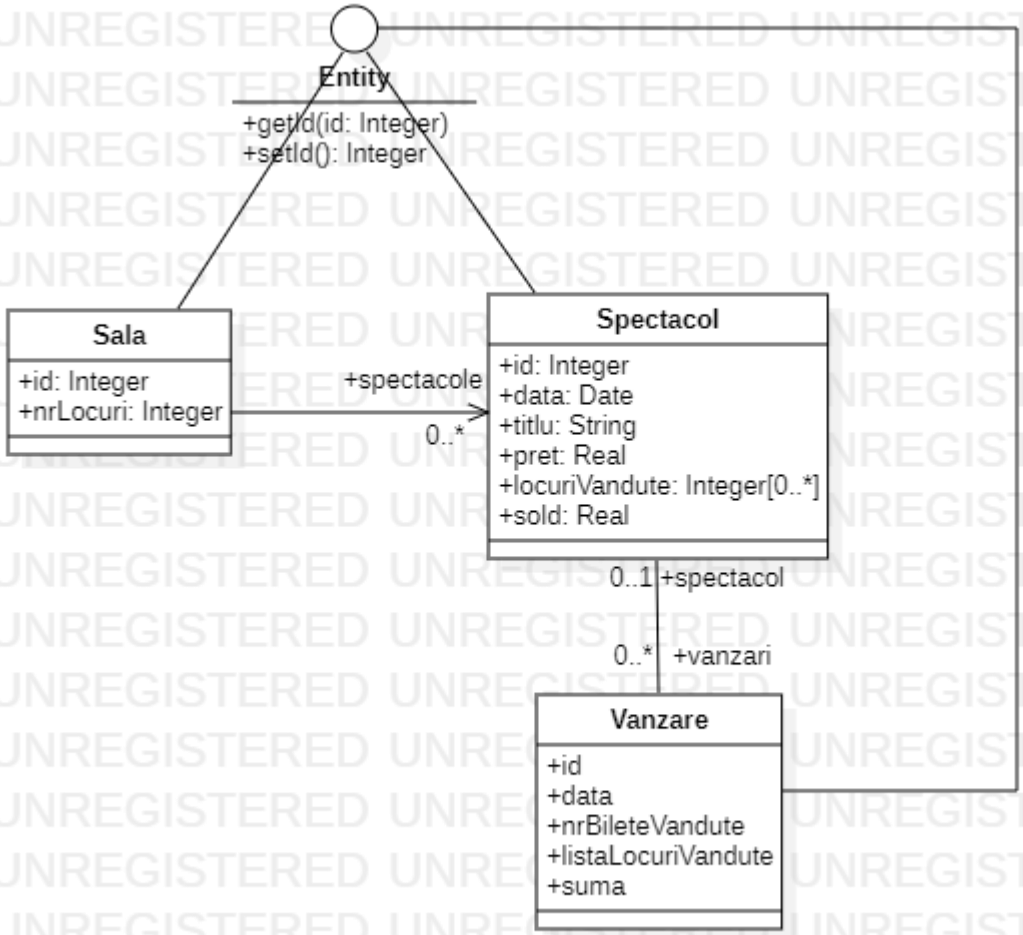
- Vanzare bilete
- Verificarea corectitudinii datelor

Cerinte nefunctionale:

- Sistemul foloseste un mecanism de tip “Thread Pool” pentru rezolvarea task-urilor
- Pentru testare se va considera ca fiecare client initiaza la interval de 2 secunde o noua cerere de vanzare folosind date generate aleatoriu si primeste de la server o notificare – vanzare reusita sau nereusita.

Proiectare

Urmatoarea diagrama prezinta modelul conceptual al aplicatiei.



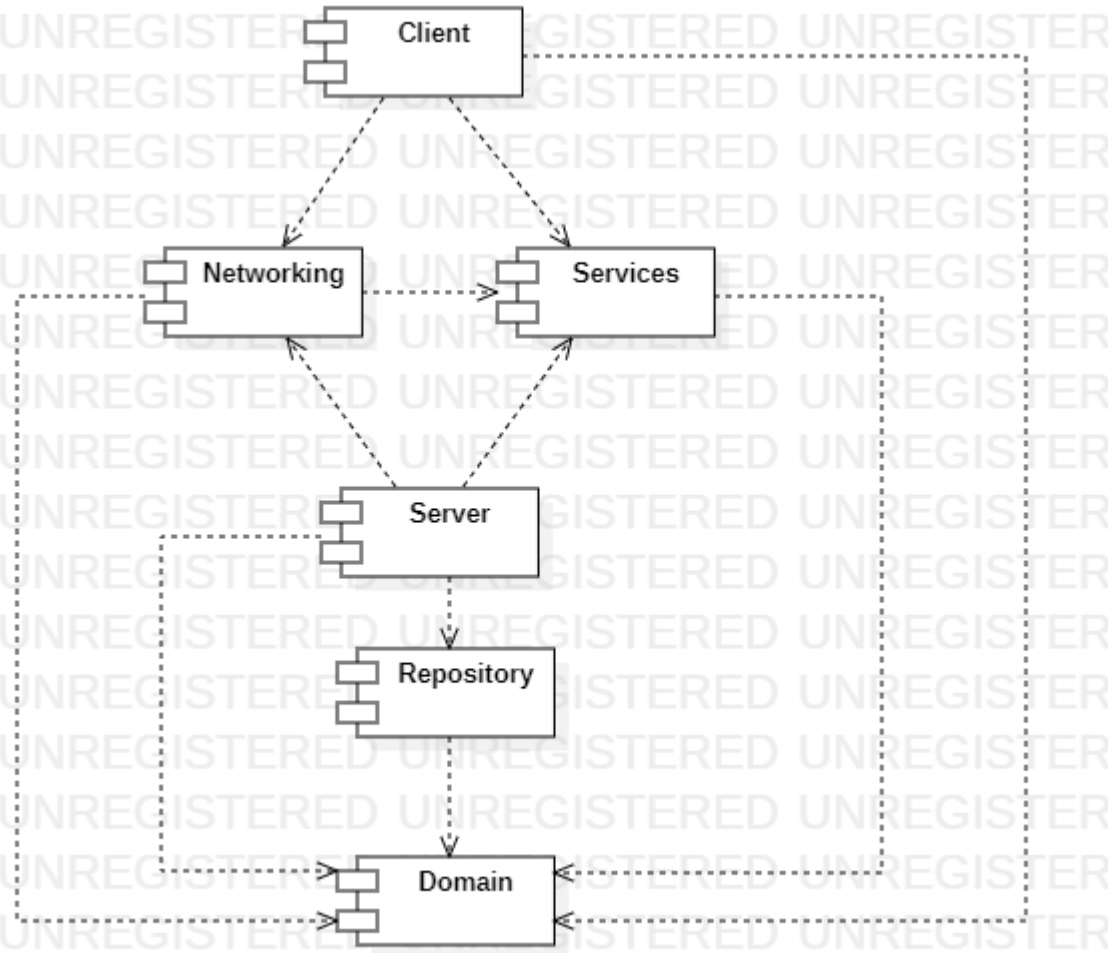
Interfata `Entity` reprezinta o abstractizare a notiunii de “entitate”, oferind o modalitate mai usoara de lucru cu entitatile pe care le-am definit pe baza identificatorului lor unic. La fel cum reiese si din modelul conceptual, fiecare entitate implementeaza aceasta interfata, comportamentul implementat fiind unul similar la toate entitatile – manipularea atributului `id`.

Atributele entitatilor sunt extrase din Analiza cerintelor, iar intre acestea sunt stabilite relatii. Intre `Sala` si `Spectacol` are loc o relatie unidirectionala cu multiplicitatea `0..*`, deoarece intr-o sala se pot organiza mai multe spectacole. `Spectacol` nu are o referinta la `Sala`, deoarece in Analiza cerintelor s-a precizat ca exista o singura sala. Intre `Spectacol` si `Vanzare` are lor o relatie bidirectionala – un spectacol are mai multe achizitii de vanzari (deci multiplicitate `0..*`), iar o vanzare se refera la un singur spectacol (deci multiplicitate `0..1`). Cand am definit relatiile, am ales sa relaxam multiplicitatiile, permitand multiplicitatatea 0.

Detalii de implementare

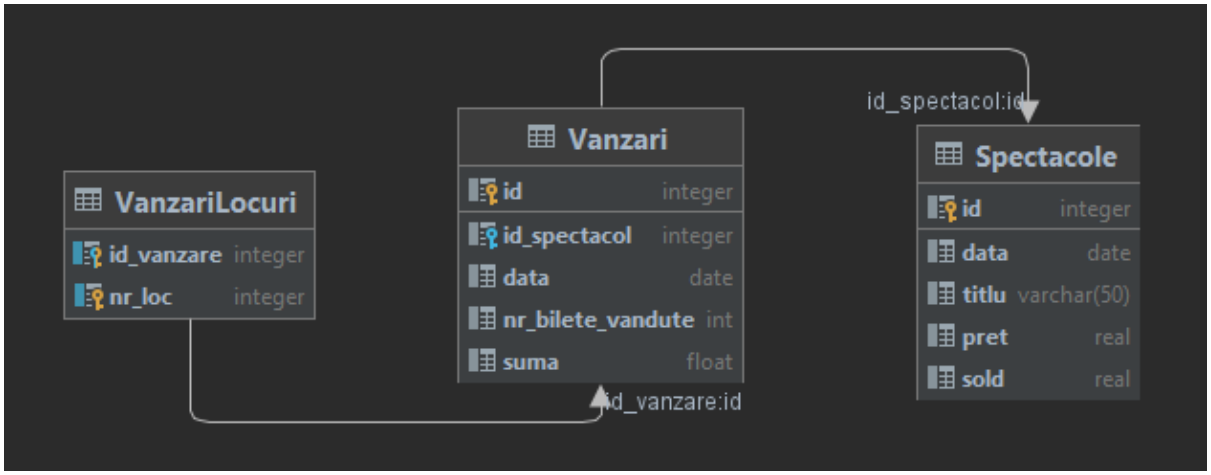
Proiectul a fost realizat in limbajul de programare Java.

Urmatoarea diagrama prezinta proiectele Java care au fost definite, impreuna cu dependentele dintre ele.



Proiectul `Domain` contine clasele care sunt prezente in modelul conceptual, prezentat in sectiunea Proiectare.

`Repository` implementeaza legatura cu baza de date, care va fi responsabila pentru persistenta spectacolelor si a vanzarilor aferente. Pentru conexiunea aplicatiei cu baza de date s-a folosit JDBC, iar ca si sistem de gestiune a bazei de date s-a folosit PostgreSQL. La calea `/Repository/src/main/resources/createDatabase.sql` este inclus un script SQL care, executat, va genera structura bazei de date. Urmatoarea diagrama prezinta tabelele care au fost definite, impreuna cu relatiile dintre ele. Aceasta respecta in mare parte modelul conceptual, singura abatere fiind reprezentata de includerea tabelei `VanzariLocuri`, ce va pastra locurile rezervate la fiecare vanzare. Includerea acestei tabele a fost necesara pentru a respecta modelul relational.



Proiectul `Services` contine 2 interfete care definesc metodele pe care le expune aplicatia la clienti (`services.Service`), precum si tipul de comunicare ce poate avea loc de la server la client (`services.Observer`, pentru a notifica clientii activi ca serverul se inchide).

Proiectul `Server` contine implementarea serviciului (mai precis, a interfetei `services.Service`) prin clasa `service.MasterService`. Aceasta clasa are referinte la repository-urile aferente entitatilor definite in `Domain`, precum si o lista de `services.Observer`, care va fi folosita pentru a notifica clientii ca serverul se inchide. In cadrul implementarilor metodelor `check` si `buyTicket` au loc verificari precum verificarea soldului unui spectacol cu vanzarile aferente sau faptul ca in lista locurilor vandute nu exista duplicate. Este important de mentional faptul ca metoda `check` verifica corectitudinea tuturor spectacolelor viitoare.

`Networking` contine atat implementarea serverului, cat si a comunicarii dintre client si server.

Comunicarea dintre client si server se realizeaza folosind protocolul RCP, definindu-se astfel clasele `rpcprotocol.ClientRpcWorker` si `rpcprotocol.ServicesRpcProxy`.

`ClientRpcWorker` poate fi vazut ca un "reprezentant al clientului pe partea de server", acesta continand informatiile clientului (adresa ip, faptul ca este conectat sau nu etc.). Acesta transforma cererile clientului care vin pe protocolul TCP intr-un format cu care `MasterService` poate lucra. `ClientRpcWorker` implementeaza interfata `Observer`, trimitand o notificare clientului atunci cand se sesizeaza faptul ca serverul s-a inchis.

Simetric, `ServicesRpcProxy` poate fi vazut ca un "reprezentat al serverului pe partea de client", fiind de asemenea o implementare a sablonului de proiectare Proxy. Acesta implementeaza interfata `services.Service`, clientul nestiind faptul ca nu comunica direct cu serviciul propriu-zis. Rolul lui `ServicesRpcProxy` este de a trimite cererile clientului la server si de a primi tot

ce primeste de la server, fiind mentinut un thread separat care va asculta dupa mesaje si le va pune intr-o coada (`BlockingQueue`).

Aplicatia foloseste un server concurent, implementat in `utils.RpcConcurrentServer`. Acesta mentine un thread, cu ajutorul unei instante a clasei `java.util.Timer`, pentru a opri serverul dupa un anumit timp, notificandu-se si clientii. Printr-o instanta a clasei `java.util.concurrent.ScheduledExecutorService`, se mentine un thread care va apela metoda de `check`, implementata in `service.MasterService`, continuu la un anumit interval de timp precizat. Printr-o instanta a clasei `java.util.concurrent.ExecutorService`, se mentine un thread pool care va asigna un thread la fiecare cerere a unui client, folosindu-se astfel de clasa `rpcprotocol.ClientRpcWorker`. Atunci cand serverul se inchide, se apeleaza si metoda `shutdown` de la acesti `ExecutorService`-uri.

In proiectul Client se creaza o instanta de rpcprotocol.ServicesRpcProxy, are loc conectarea cu serverul (prin apelul metodei addObserver, poate fi vazuta ca si un login clasic), iar apoi sunt trimise cereri pentru a cumpara bilete la fiecare interval de timp precizat. Continutul cererilor este fie aleatoriu, fie hardcodat, pentru a ne genera un continut care rezulta intr-o cumparare de bilet reusita, cu o probabilitate de 50%.

Rezultate

Nr. min. rulare server	Interval realizare verificare (s)	Nr. thread-uri server	Nr. clienti simultani	Nr. cereri de cumparare bilet solutionate	Nr. verificari efectuate
2	5	4	1	55	24
			2	103.6	24
		8	1	56.25	24
			2	97.75	24
	10	4	1	54	12
			2	100	12
		8	1	54	12
			2	97.5	12

Nota: fiecare test a fost rulat de 10 ori si pentru evaluarea timpului de executie s-a considerat media aritmetica a acestor 10 rulari. Fiecare test a fost rulat pe o masina cu Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz 2.60 MHz 2 cores 4 Logical Processors cu sistem de operare Windows 10 versiunea 21H1.

Analiza rezultatelor

Se poate observa ca in cadrul fiecarui test, s-a respectat ca verificarea corectitudinii datelor sa se efectueze de cate ori a fost nevoie ($\frac{nr.\ minute\ rulare\ server*60}{interval\ efectuare\ verificare\ (s)}$).

Daca in testele in care a fost folosit un singur client este aproximativ constant numarul de cereri de cumparare bilet solutionate, acesta fluctueaza daca sunt folositi mai multe clienti, numarul de bilete vandute depinzand de numarul de thread-uri pe care le aloca serverul in thread pool-ul folosit.

Astfel, se poate observa ca in general, daca sunt folosite mai multe thread-uri, numarul de cereri pe care le poate gestiona serverul este mai mic. Un posibil motiv este faptul ca masina pe care au fost rulate testele are doar 4 procesoare logice, iar folosirea a 8 thread-uri, la care se adauga cele pentru efectuarea verificarii si procesele client care au fost deschise, genereaza un overhead mult prea mare.

De asemenea, se observa faptul ca, in general, pot fi solutionate mai multe cereri de catre server daca se efectueaza mai multe verificari pentru corectitudinea datelor. Un motiv include faptul ca thread-ul care este responsabil cu efectuarea verificarilor sta mult mai putin in starea de asteptare daca intervalul de timp precizat este mai mic, iar astfel, se fac mai putine verificari relativ la cuanta de timp care a trecut de la ultimul “check”.