

Arhitectura MVC

1. Obiective
2. Introducere. Arhitectura cu trei straturi
3. Arhitectura *MVC*
4. Arhitectura *MVP*
5. Aplicații

1. Obiective

Obiectivul principal al acestui laborator este implementarea unui program după șablonul arhitectural *Model-Vizualizare-Prezentator* (engl. “Model-View-Presenter”, MVP), o variantă a arhitecturii clasice *Model-Vizualizare-Controlor* (engl. “Model-View-Controller”, MVC).

Ca obiective detaliate, vom avea:

1. Obiective de proiectare: particularizarea arhitecturii *MVP* pentru o aplicație cu interfață de tip consolă, subliniind faptul că arhitectura este independentă de tipul interfeței cu utilizatorul;
2. Obiective de programare: realizarea unui meniu consolă structurat pe niveluri;
3. Obiective diverse: calcularea distanței între două puncte de pe suprafața Pământului definite de coordonatele lor geografice.

2. Introducere. Arhitectura cu trei straturi

Una din recomandările de bază ale ingineriei programării este structurarea arhitecturii unei soluții pe niveluri, adică împărțirea sistemului în mai multe componente ordonate ierarhic, fiecare cu limitări legate de modul de interacțiune.

Din punct de vedere al terminologiei, se folosește „strat” (engl. “tier”) pentru a indica o separare fizică a componentelor, adică *assembly*-uri (dll, exe) diferite pe aceeași mașină sau pe mașini diferite. Termenul de „nivel” (engl. “layer”) indică o separare logică a componentelor, de exemplu *namespace*-uri diferite.

O abordare simplă și des întâlnită este utilizarea unei arhitecturi cu două straturi (engl. “two-tier architecture”). În acest caz, aplicația separă stratul de *prezentare* de stratul *datelor* aplicației. Datele reprezintă entitățile care definesc problema și de obicei corespund unor tabele în baze de date. Clasele de prezentare au responsabilități precum recepționarea intrărilor de la utilizator (texte introduse, apăsarea unor butoane, alegerea unor opțiuni din meniuri etc.), apelurile către stratul de date, deciziile privind informațiile care vor fi arătate utilizatorului și afișarea ieșirilor (texte, grafice etc.). Aceste responsabilități sunt destul de numeroase și, pe măsură ce sistemul evoluează, stratul de prezentare poate deveni supraîncărcat.

O soluție naturală este divizarea acestui strat prea extins în două alte straturi: de *prezentare*, pentru preluarea intrărilor și afișarea ieșirilor, și respectiv de *logică a aplicației*, pentru asigurarea comunicațiilor cu stratul de acces la date și pentru luarea deciziilor de control. Stratul de logică include toate prelucrările efective care manipulează datele interne și pe cele ale utilizatorului, cu ajutorul algoritmilor specifici, pentru a controla fluxul aplicației.

Figura 1 prezintă comparativ arhitecturile cu două și trei straturi.

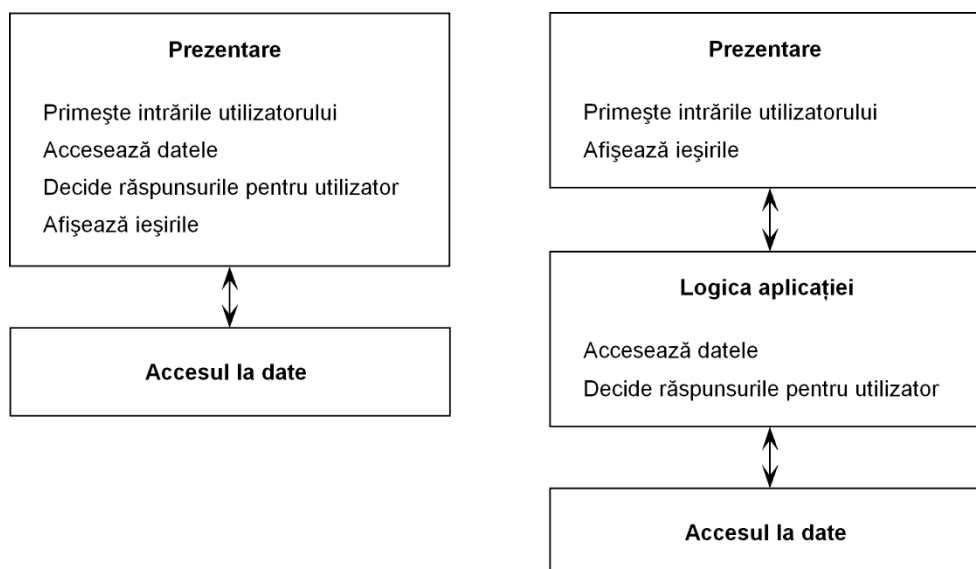


Figura 1. Arhitectura cu două straturi; arhitectura cu trei straturi

3. Arhitectura MVC

Arhitectura cu două straturi, în care se consideră că schimbul de informații are loc între interfața cu utilizatorul și bazele de date (sau în general orice modalitate de stocare a datelor, inclusiv fișiere text, *xml* etc.) presupune ca din interfață să se acceseze și să se modifice direct datele.

Totuși, această abordare are câteva probleme semnificative. În primul rând, interfața cu utilizatorul se schimbă de obicei mai des decât baza de date. În al doilea rând, majoritatea aplicațiilor conțin cod funcțional (logica) ce realizează prelucrări mult mai complexe decât simpla transmitere de date.

Șablonul arhitectural *Model-Vizualizare-Controlor* izolează interfața, codul funcțional și datele, astfel încât modificarea unuia din aceste trei componente să nu le afecteze pe celelalte două. O aplicație bazată pe șablonul MVC va avea trei module corespunzătoare:

1. **Model:** conține datele, starea și logica aplicației. Deși nu cunoaște Controlorul și Vizualizarea, furnizează o interfață pentru manipularea și preluarea stării și poate trimite notificări cu privire la schimbarea stării. De obicei primește cereri privind starea datelor de la Vizualizare și instrucțiuni de modificare a datelor sau stării de la Controlor;
2. **Vizualizare:** afișează Modelul într-o formă potrivită pentru utilizator. Pentru un singur Model pot exista mai multe Vizualizări, de exemplu o listă de elemente poate fi afișată într-un control vizual precum *ListBox*, într-o consolă sau într-o pagină web;

3. **Controlor:** primește intrările de la utilizator și apelează obiectele Modelului pentru a prelucra noile informații.

Există mai multe variante ale arhitecturii *MVC*, însă în general fluxul de control este următorul:

1. Utilizatorul interacționează cu interfața aplicației, iar Controlorul preia intrarea și o interpretează ca pe o acțiune ce poate fi recunoscută de către Model;
2. Controlorul trimite Modelului acțiunea utilizatorului, ceea ce poate conduce la schimbarea stării Modelului;
3. În vederea afișării rezultatului de către Vizualizare, Controlorul îi poate trimite acesteia o cerere de actualizare, sau Modelul îi trimite o notificare privind schimbarea stării sale;
4. Vizualizarea preia datele necesare din Model și le afișează.

În continuare, aplicația așteaptă o nouă acțiune a utilizatorului iar ciclul se reia.

Trebuie precizat că Modelul nu este doar o bază de date, ci încapsulează și logica domeniului necesară pentru manipularea datelor din aplicație. De multe ori se folosește și un mecanism de stocare persistentă a acestora, de exemplu într-o bază de date, însă arhitectura *MVC* nu menționează explicit stratul de acces la date; acesta se consideră implicit ca parte din Model.

Figura 2 prezintă relațiile structurale între cele trei module.

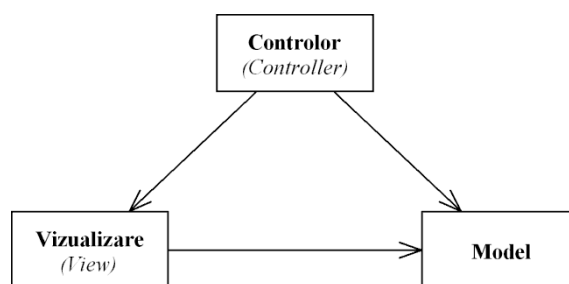


Figura 2. Relațiile structurale între componentele arhitecturii *MVC*

Pentru o aplicație mai simplă, aceste module pot reprezenta clase. Din punctul de vedere al implementării, săgețile de asociere înseamnă că:

- Vizualizarea va avea un câmp de tip Model; de obicei va primi ca parametru în constructor o referință la obiectul Model;
- Controlorul va avea două câmpuri de tip Vizualizare și Model; de obicei va primi ca parametri în constructor referințe la obiectele Vizualizare și Model.

Mai ales în aplicațiile web, este clar definită separația dintre Vizualizare (*browser-ul*) și Controlor (componentele server care răspund cererilor *http*).

Prin separarea celor trei funcționalități se atinge o cuplare slabă între module, caracteristică dorită în toate programele deoarece modificările dintr-o secțiune a codului nu necesită modificări și în alte secțiuni. Decuplarea scade complexitatea proiectării și crește flexibilitatea și potențialul de reutilizare.

Avantajele principale ale șablonului *MVC* sunt următoarele:



- **Modificări rapide.** Clasele șablonului trebuie doar să implementeze niște interfețe prestabilite, astfel încât acestea să cunoască metodele pe care le pot apela în celelalte clase. Când se doresc modificări, nu trebuie rescrisă o clasă, se poate implementa una nouă și se poate utiliza direct, chiar alături de una veche. De asemenea, Vizualizările și Modelele existente pot fi refolosite pentru alte aplicații cu un Controlor diferit.
- **Modele de date multiple.** Modelul nu depinde de nicio altă clasă din șablon. Datele pot fi stocate în orice format: text, *xml* sau baze de date Access, Oracle, SQL Server etc.;
- **Interfețe multiple.** Deoarece Vizualizarea este separată de Model, pot exista în aplicație mai multe tipuri de Vizualizări ale acelorași date. Utilizatorii pot alege mai multe scheme de afișare: mai multe *skin*-uri sau comunicarea în mai multe limbi. Aplicația poate fi extinsă ușor pentru a include moduri de vizualizare complet diferite: consolă, interfață grafică cu utilizatorul în ferestre (*desktop*), documente web sau pentru PDA-uri.

4. Arhitectura *MVP*

În abordarea clasică, descrisă de Trygve Reenskaug pe când lucra la limbajul Smalltalk la Xerox PARC (1978-1979), logica este în Model, iar Controlorul gestionează intrările de la utilizator.

Pentru aplicațiile de tip consolă, este relativ simplu ca intrările să fie preluate de Controlor iar afișarea să se facă de către Vizualizare, pe baza datelor din Model. Pentru aplicațiile moderne, cu interfețe grafice cu utilizatorul (GUI) precum ferestrele Windows, clasele de vizualizare sunt cele care primesc intrările utilizatorului. De aceea, în astfel de situații, Vizualizarea și Controlorul nu mai sunt clar delimitate.

Pentru a răspunde noilor realități ce privesc interacțiunea utilizatorilor cu interfețele aplicațiilor, a fost propus șablonul *Model-Vizualizare-Prezentator*, *MVP*. Aici, stratul de *prezentare* constă în obiecte de Vizualizare iar *logica aplicației* constă în obiecte de control (Prezentator/Controlor). Pentru fiecare obiect de vizualizare există un obiect de control.

Din punct de vedere al terminologiei, se poate utiliza termenul de Controlor pentru Prezentator în șablonul *MVP*, deoarece acesta poate fi considerat o variantă modernă a șablonului clasic *MVC*.

Deși ambele șabloane, *MVC* și *MVP*, se bazează pe principiul comun al arhitecturii cu trei straturi, acestea au două diferențe majore:

1. În *MVC*, Controlorul primește și prelucrează intrările de la utilizator iar în *MVP*, Vizualizarea primește intrările și apoi deleagă prelucrările către Controlorul corespunzător;
2. În *MVC*, Vizualizarea primește notificări privind schimbările Modelului și afișează noile informații pentru utilizator. În *MVP*, Controlorul modifică direct Vizualizarea, ceea ce face șablonul *MVP* mai ușor de folosit decât șablonul *MVC*.

Aceste diferențe fac șablonul *MVP* mai atractiv decât șablonul *MVC* pentru aplicațiile cu interfață vizuală din prezent.

4.1. Variante de actualizare a *Vizualizării*

Când Modelul este actualizat, Vizualizarea trebuie de asemenea actualizată pentru a reflecta modificările. Actualizarea Vizualizării poate fi realizată în două variante: *Vizualizarea pasivă* (engl. “Passive View”) și *Controlorul supervisor* (engl. “Supervising Controller”). Figura 3 ilustrează modelele logice ale celor două variante.

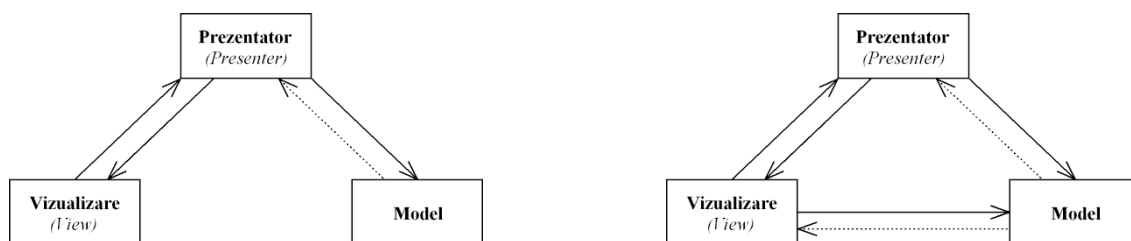


Figura 3. Vizualizarea pasivă și Controlorul supervisor

În abordarea *Vizualizării pasive*, Prezentatorul actualizează Vizualizarea pentru a reflecta schimbările din Model. Interacțiunea cu Modelul este gestionată exclusiv de Prezentator iar Vizualizarea nu își poate da seama direct de schimbările din Model. La rândul său, Vizualizarea este actualizată exclusiv de către Prezentator.

Abordarea este utilă atunci când Prezentatorul trebuie să realizeze unele prelucrări complexe asupra datelor, înainte de a afișa informațiile pentru utilizator. Acestea apar de exemplu atunci când stările controalelor din interfața grafică depind de anumite prelucrări ale datelor. Să considerăm cazul în care un utilizator dorește să împrumute o carte a unui anumit autor de la bibliotecă. Dacă toate cărțile autorului respectiv sunt deja împrumutate, butonul *Împrumută* poate fi dezactivat. În această situație, faptul că împrumutul este imposibil nu este reținut în Model, ci este determinat de Prezentator, care apoi îi cere Vizualizării să dezactiveze butonul respectiv.

În abordarea *Controlorului supervisor*, Vizualizarea interacționează direct cu Modelul pentru a transfera date, fără intervenția Prezentatorului. Prezentatorul actualizează Modelul și manipulează starea Vizualizării doar în cazurile în care există o logică complexă a interfeței cu utilizatorul. Vizualizarea poate fi actualizată și direct, pe baza modificărilor datelor din Model.

Astfel de transferuri simple, în care Vizualizarea comunică direct cu Modelul, sunt în general situațiile în care se schimbă unele date în Model iar acestea sunt preluate și afișate în Vizualizare, fără prelucrări suplimentare. De exemplu, interfața afișează o listă de cărți împrumutate de un student de la bibliotecă, listă păstrată de Model într-o bază de date. Când studentul împrumută o nouă carte, baza de date se schimbă iar Vizualizarea preia direct din Model elementele listei pentru afișare.

Vizualizarea pasivă este asemănătoare arhitecturii cu trei straturi tipice, în care stratul de logică a aplicației se interpune între stratul de prezentare și stratul de acces la date.

Decizia asupra alegerii uneia din cele două variante depinde de prioritățile aplicației. Dacă este mai importantă testabilitatea, *Vizualizarea pasivă* este mai potrivită, deoarece se poate testa

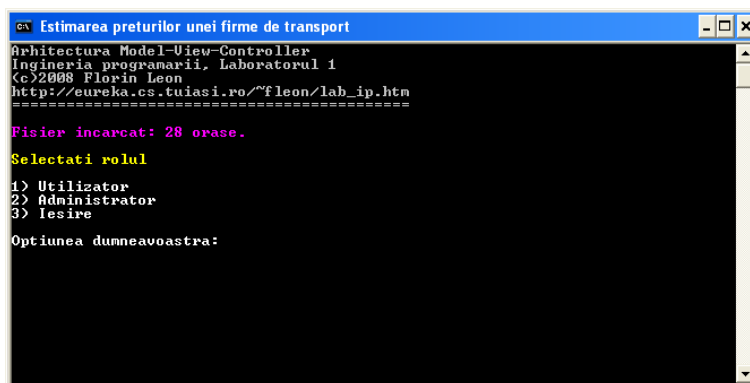
toată logica interfeței cu utilizatorul prin testarea Prezentatorului. Pe de altă parte, dacă simplitatea este mai importantă, *Controlorul supervisor* este o opțiune mai bună deoarece, pentru schimbări mici în interfață, nu mai trebuie inclus cod în Prezentator pentru actualizarea Vizualizării. Astfel, *Controlorul supervisor* necesită de obicei mai puțin cod întrucât Prezentatorul nu mai efectuează actualizările simple ale Vizualizării.

Șabloanele *MVC* și *MVP* au scopuri similare, însă diferă prin modalitățile în care își ating aceste scopuri.

5. Aplicații

5.1. Realizați un program de tip consolă cu arhitectura *MVP*, *Controlor supervisor* pentru determinarea costurilor unei firme de transport. Se vor putea calcula costurile de transport între două orașe, identificate prin nume, latitudine și longitudine.

Aplicația va permite două roluri: administrator și utilizator, cu funcții diferite (figura 4).



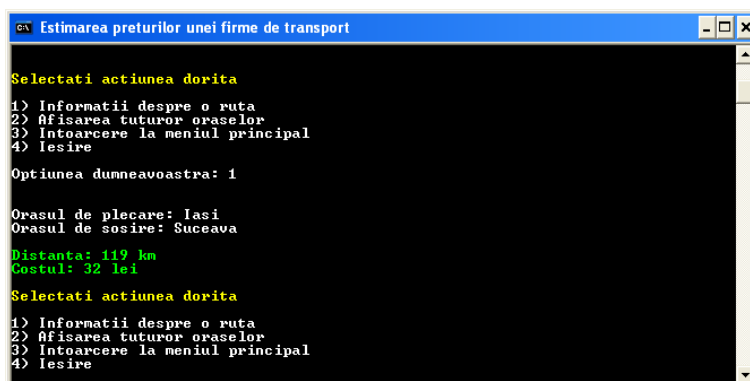
```

Estimarea preturilor unei firme de transport
Arhitectura Model-View-Controller
Ingineria programarii, Laboratorul 1
(c)2008 Florin Leon
http://eureka.cs.tuiasi.ro/~fleon/lab_ip.htm
=====
Fisier incarcata: 28 orase.
Selectati rolul
1) Utilizator
2) Administrator
3) Iesire
Optiunea dumneavoastra:

```

Figura 4. Meniul principal

Pentru rolul de utilizator comenzile disponibile sunt prezentate în figura 5.



```

Estimarea preturilor unei firme de transport
Selectati actiunea dorita
1) Informatii despre o ruta
2) Afisarea tuturor oraselor
3) Intoarcere la meniul principal
4) Iesire
Optiunea dumneavoastra: 1
Orasul de plecare: Iasi
Orasul de sosire: Suceava
Distanța: 119 km
Costul: 32 lei
Selectati actiunea dorita
1) Informatii despre o ruta
2) Afisarea tuturor oraselor
3) Intoarcere la meniul principal
4) Iesire

```

Figura 5. Meniul rolului de utilizator

Pentru rolul de administrator comenzile disponibile sunt prezentate în figura 6.

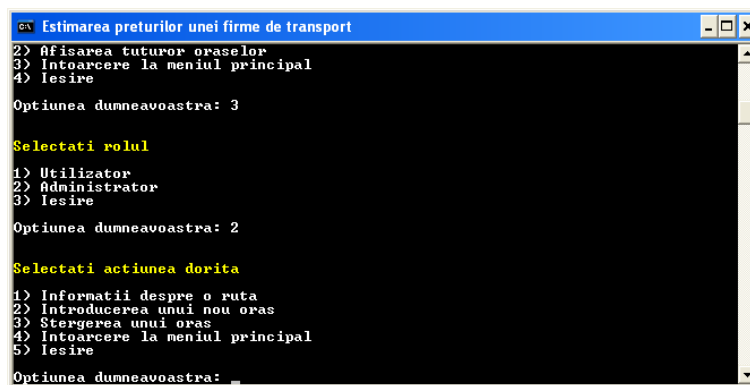


Figura 6. Meniul rolului de administrator

Indicații:

- Se furnizează codul sursă pentru Model, împreună cu fișierul text cities.txt care conține mai multe orașe;
- Se furnizează în fișierul Metode si structuri de date.txt codul sursă pentru mai multe structuri de date și metode care trebuie plasate corect în Vizualizare sau în Prezentator;
- Întrucât cele trei meniuri au structuri similare, se recomandă crearea unei metode comune care să primească drept parametru lista de opțiuni posibile;
- Se recomandă ca opțiunile alese de utilizator să fie tratate ca o enumerație.

Metoda Main din clasa Program poate avea conținutul următor:

```
static class Program
{
    static void Main()
    {
        IModel model = new Model();
        IView view = new ConsoleView(model);
        IPresenter presenter = new Presenter(view, model);
        view.SetPresenter(presenter);
        ((ConsoleView)view).Start();
    }
}
```

Un exemplu de diagramă de clase pentru aplicație este prezentată în figura 7. Pentru creșterea clarității, nu s-au mai reprezentat explicit relațiile de asociere pentru câmpurile de tip IPresenter, IModel, IView și nici numele metodelor din interfețe implementate de clasele concrete.

Metoda ListAll din clasele View apelează direct metoda ListAll din clasa Model. Aici apare diferența între abordarea *Controlorului supervisor* și cea a *Vizualizării pasive*. Dacă s-ar fi utilizat cea de a doua abordare, ListAll din View ar fi apelat o metodă corespunzătoare din Presenter, iar aceasta ar fi apelat metoda ListAll din Model.

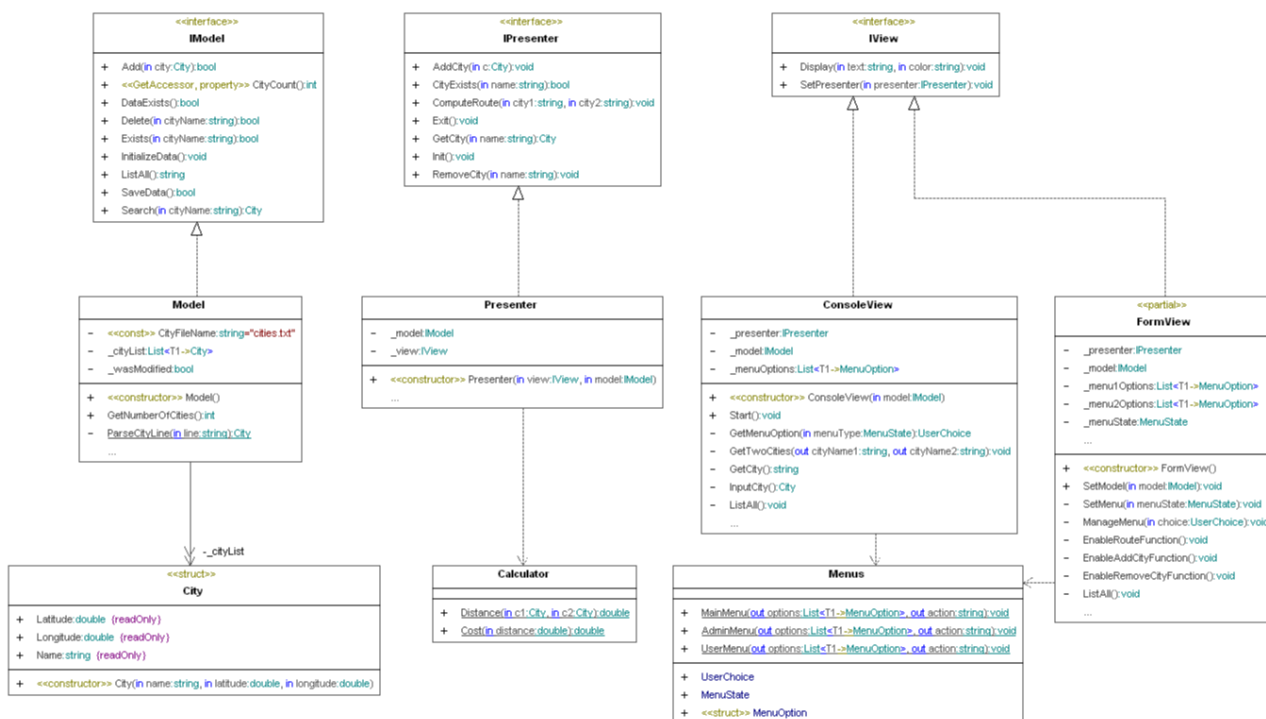


Figura 7. Exemplu de rezolvare: diagrama de clase

Proiectele soluției și fișierele sursă pot fi structurate ca în figura 8.

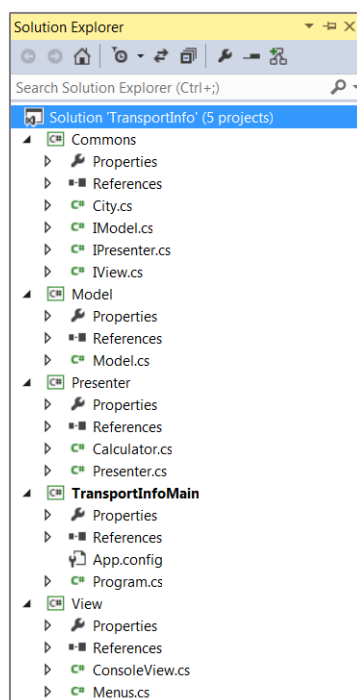


Figura 8. Exemplu de rezolvare: structurarea soluției

Adăugarea referințelor între proiectele soluției se face după schema din figura 9.

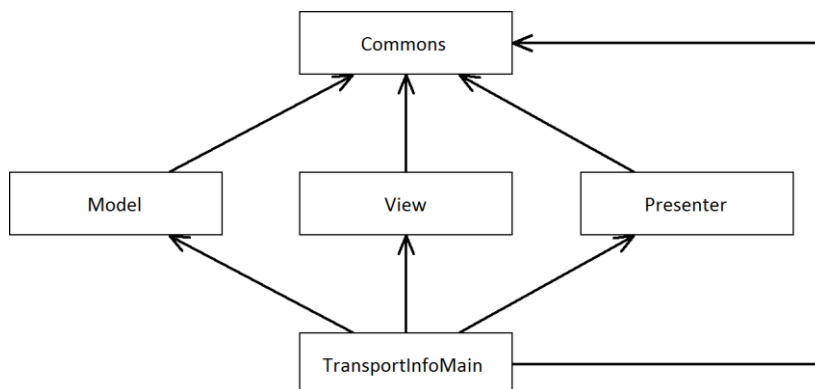


Figura 9. Structura de adăugare a referințelor

5.2. (opțional) Păstrând aceleași clase pentru Model și Prezenter, realizați o Vizualizare nouă, cu o interfață grafică de tip *Windows Forms*, cu aceeași funcționalitate ca și aplicația consolă dezvoltată anterior.

Pentru a forța utilizatorul să folosească doar opțiunea de *Ieșire* pentru a închide programul, butonul de închidere a ferestrei poate fi dezactivat cu ajutorul secvenței următoare de cod:

```

#region Disable Close X Button
const int MF_BYPOSITION = 0x400;

[DllImport("User32")]
private static extern int RemoveMenu(IntPtr hMenu, int nPosition, int wFlags);

[DllImport("User32")]
private static extern IntPtr GetSystemMenu(IntPtr hWnd, bool bRevert);

[DllImport("User32")]
private static extern int GetMenuItemCount(IntPtr hWnd);

private void FormView_Load(object sender, EventArgs e)
{
    IntPtr hMenu = GetSystemMenu(this.Handle, false);
    int menuItemCount = GetMenuItemCount(hMenu);
    RemoveMenu(hMenu, menuItemCount - 1, MF_BYPOSITION);
}
#endregion
  
```

Un exemplu de interfață grafică este prezentat în capturile ecran din figurile 10, 11 și 12.

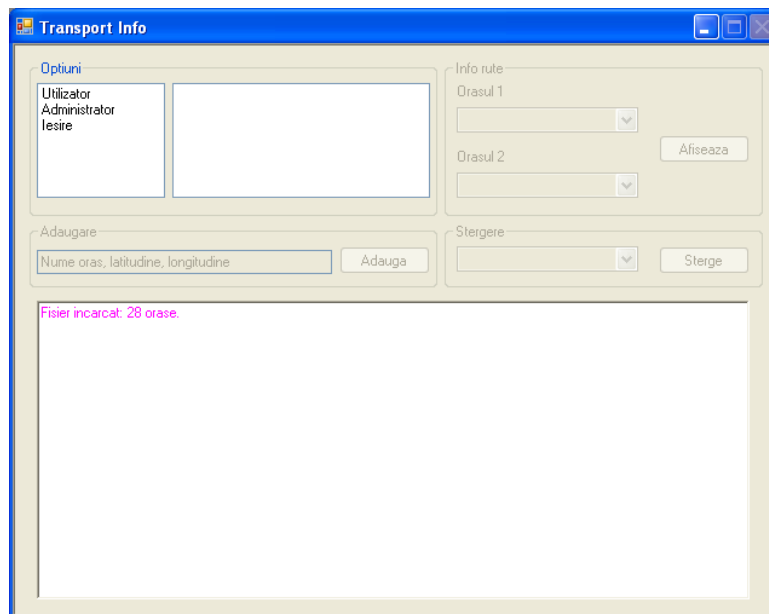


Figura 10. Exemplu de rezolvare: meniul principal

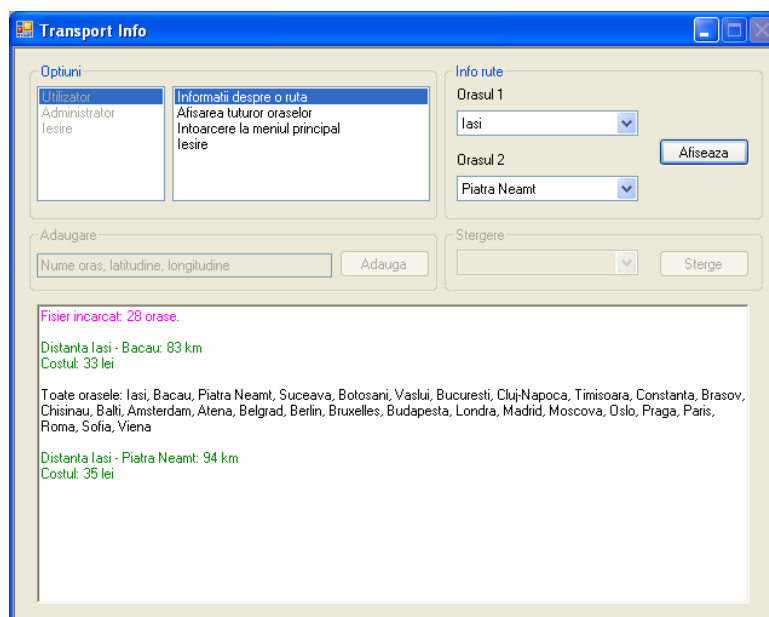


Figura 11. Exemplu de rezolvare: rolul de utilizator

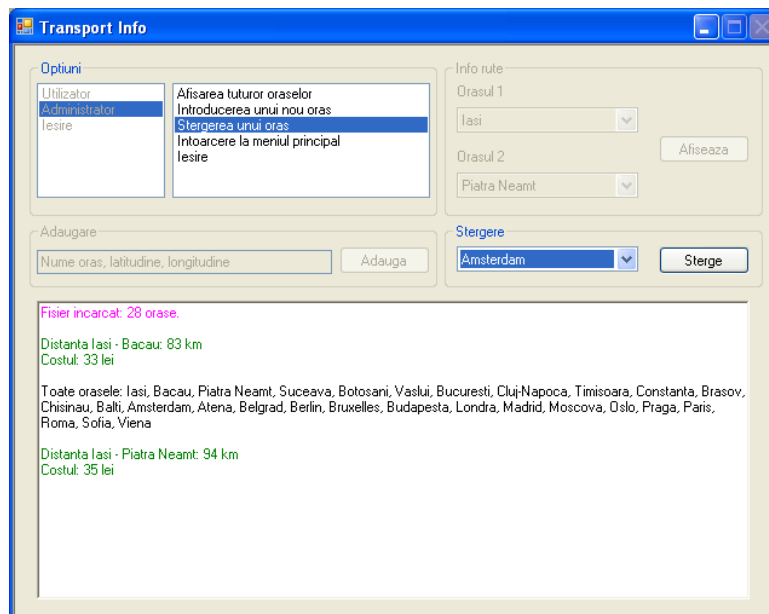


Figura 12. Exemplu de rezolvare: rolul de administrator