

Reutilizarea codului cu ajutorul DLL-urilor

1. Obiective
 2. Bibliotecile legate dinamic
 3. Crearea DLL-urilor în C#
 4. Aplicații
- A1. Grafică în C#

1. Obiective

Obiectivele laboratorului 3 sunt următoarele:

- Descrierea DLL-urilor;
- Legarea statică a DLL-urilor .NET, cea mai simplă și folosită formă;
- Legarea dinamică a DLL-urilor .NET, utilă pentru crearea *plug-in*-urilor;
- Prezentarea unor aspecte privind grafica în C#.

2. Bibliotecile legate dinamic

Bibliotecile legate dinamic (engl. “Dynamic Link Libraries”, DLL), reprezintă implementarea Microsoft a conceptului de *bibliotecă partajată* (engl. “shared library”). În trecut, DLL-urile au dat dezvoltatorilor posibilitatea de a crea biblioteci de funcții care puteau fi folosite de mai multe aplicații. Însuși sistemul de operare Windows a fost proiectat pe baza DLL-urilor. În timp ce avantajele modulelor de cod comun au extins oportunitățile dezvoltatorilor, au apărut de asemenea probleme referitoare la actualizări și revizii. Dacă un program se baza pe o anumită versiune a unui DLL și alt program actualiza același DLL, de multe ori primul program înceta să mai funcționeze corect.

Pe lângă problemele legate de versiuni, dacă se dorea dezinstalarea unei aplicații, se putea șterge foarte ușor un DLL care era încă folosit de un alt program.

Soluția propusă de Microsoft a fost introducerea posibilității de a urmări folosirea DLL-urilor cu ajutorul registrului, începând cu Windows 95. Se permitea unei singure versiuni de DLL să ruleze în memorie la un moment dat. Când era instalată o nouă aplicație care folosea un DLL existent, se incrementa un contor. La dezinstalare, contorul era decrementat și dacă nicio aplicație nu mai folosea DLL-ul, atunci acesta putea fi șters.

Apare însă o altă problemă deoarece când un DLL este încărcat, Windows va folosi versiunea ce rulează până când nicio aplicație nu o mai folosește. Astfel, chiar dacă DLL-ul sistemului este în regulă, sau o aplicație are o copie locală pe care se lucrează, când aplicația *precedentă* a pornit cu o versiune incompatibilă, atunci noua aplicație nu va merge.

Această problemă se poate manifesta în situații precum următoarele: o aplicație nu funcționează atunci când rulează o altă aplicație sau, și mai ciudat, o aplicație nu funcționează dacă o altă aplicație a rulat (dar nu mai rulează neapărat în prezent). Dacă aplicația A încarcă o bibliotecă incompatibilă sau coruptă, atunci aplicația B lansată folosește această bibliotecă. Această versiune va sta în memorie chiar după ce aplicația A nu mai există (atât timp cât aplicația B încă rulează), deci aplicația B s-ar putea să înceteze să funcționeze din cauza aplicației A, chiar dacă aceasta nu mai rulează. O a treia aplicație C poate să eșueze (câtă vreme aplicația B încă rulează) chiar dacă este pornită după ce aplicația A a fost închisă.

Rezultatul s-a numit *infernul DLL* (engl. “DLL hell”).

Rezolvarea infernului DLL a fost unul din scopurile platformei .NET. Aici pot exista mai multe versiuni ale unui DLL ce rulează simultan, ceea ce permite dezvoltatorilor să adauge o versiune care funcționează la programul lor fără să se îngrijoreze că un alt program va fi afectat. Modul în care .NET reușește să facă aceasta este prin renunțarea la folosirea registrului pentru a lega DLL-urile de aplicații și prin introducerea conceptului de *assembly*.

3. Crearea DLL-urilor în C#

Din fereastra corespunzătoare *File* → *New* → *Project*, se alege tipul proiectului *Class Library* (figura 1). În *namespace*-ul proiectului pot fi adăugate mai multe clase. În acest caz, din exterior fiecare clasă va fi accesată ca *Namespace.ClassName*. Dacă se elimină *namespace*-ul, clasa va fi accesată direct cu numele clasei: *ClassName*. Spre deosebire de o aplicație executabilă, aici nu va exista o metodă *Main*, deoarece DLL-ul este numai o bibliotecă de funcții utilizabile din alte programe executabile.

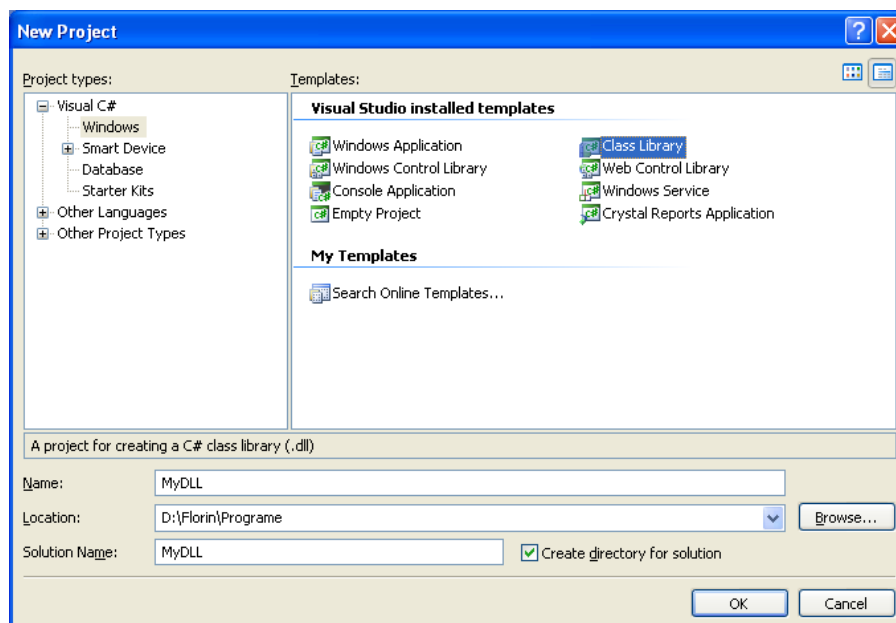


Figura 1. Crearea unui proiect de tip *Class Library* (DLL)

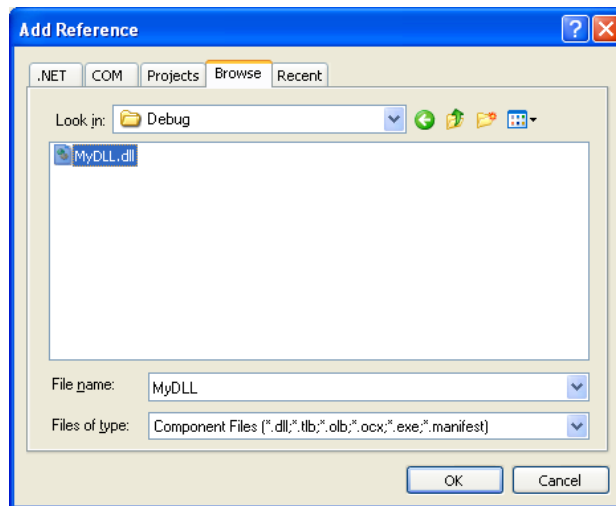



Figura 2. Adăugarea unei referințe la un DLL

3.1. Legarea statică


După ce s-a creat un proiect corespunzător unei aplicații executabile, din *Project* → *Add Reference*, se selectează de pe harddisk în *tab-page-ul Browse* fișierul DLL care trebuie adăugat în proiect. În continuare, în program vor fi utilizate ca atare toate funcțiile din DLL (figura 2).

Să considerăm următorul exemplu: avem într-un DLL numit *Operatii.dll* o clasă *Putere* în *namespace-ul* *Matematica* având o metodă *double Patrat(double x)* care returnează valoarea parametrului ridicată la pătrat. Mai întâi, se va căuta și selecta fișierul *Operatii.dll* de pe harddisk. Apoi, într-o metodă din clasa programului principal, se va apela direct metoda de ridicare la putere: 

```
double a = Matematica.Putere.Patrat(5.5);
```

3.2. Legarea dinamică

Legarea statică presupune că se cunoaște ce DLL va trebui încărcat înainte de executarea programului. Există totuși situații în care acest lucru este imposibil: de exemplu, dacă o aplicație necesită o serie de *plug-in-uri*, acestea pot fi adăugate sau șterse, iar aplicația principală trebuie să determine după lansarea în execuție cu ce DLL-uri poate lucra. Un alt avantaj este faptul că programatorul poate testa existența unui anumit DLL necesar și poate afișa un mesaj de eroare și eventual o modalitate de corectare a acesteia.

C# permite încărcarea dinamică a DLL-urilor. Să considerăm tot exemplul anterior. Apelul metodei de ridicare la pătrat se face în modul următor: 

```
// se încearcă încărcarea DLL-ului
Assembly a = Assembly.Load("Operatii");

// se identifică tipul (clasa) care trebuie instanțiată
// dacă în clasa din DLL există un namespace,
// se folosește numele complet al clasei din assembly, incluzând namespace-ul
Type t = a.GetType("Matematica.Putere");

// se identifică metoda care ne interesează
MethodInfo mi = t.GetMethod("Patrat");

// se creează o instanță a clasei dorite
// aici se apelează constructorul implicit
object o = Activator.CreateInstance(t);

// definim un vector de argumente pentru a fi trimis metodei
// metoda Pătrat are numai un argument de tip double
object[] args = new object[1];
double x = 5.5;
args[0] = x;

// apelul efectiv al metodei și memorarea rezultatului
double result = (double)mi.Invoke(o, args);
```

Acesta este modul general de încărcare. Este obligatorie *tratarea excepțiilor* care pot apărea datorită unei eventuale absențe a DLL-ului sau a încărcării incorecte a unei metode. De aceea, fluxul de mai sus va trebui împărțit în mai multe blocuri care să permită tratarea excepțiilor, *încărcarea o singură dată* a bibliotecii și *apelarea de câte ori este nevoie* a metodelor dorite.

Dacă se apelează dinamic o metodă statică, se folosește `null` în loc de obiectul `o`.

Pentru compilarea codului de mai sus este necesară includerea în program a *namespace*-ului `System.Reflection`.

3.3. Depanarea unui DLL

Deoarece un DLL nu este direct executabil, există două metode pentru dezvoltarea și depanarea unei astfel de componente.

Abordarea cea mai simplă este crearea unei aplicații executabile, de cele mai multe ori de tip consolă în cazul în care DLL-ul va conține funcții de calcul și nu grafice. În proiect va exista *namespace*-ul DLL-ului cu toate clasele aferente, iar în plus o clasă cu o metodă `Main`, din care se vor putea apela și depana metodele din DLL. La sfârșit, după ce DLL-ul este corect, se poate exclude clasa cu `Main` și se poate recompila proiectul ca DLL: *Project Properties* → *Application* → *Output type*. Alternativ, se poate crea un nou proiect de tip *Class library* în care se va copia codul DLL-ului.

A doua metodă este includerea a două proiecte într-o soluție în Visual Studio. Un proiect va fi DLL-ul iar celălalt proiect va fi unul executabil. Din proiectul executabil trebuie adăugată referința la DLL. În consecință, fișierul *dll* rezultat din compilare va fi copiat automat în directorul fișierului *exe*.



4. Aplicații

4.1. Realizați un DLL numit *Prim* care să conțină o clasă cu o metodă care testează dacă un număr natural, primit ca parametru, este prim. *Notă:* 1 nu este număr prim.

4.2. Realizați un program executabil care testează conjectura lui Goldbach: orice număr par mai mare sau egal ca 4 poate fi scris ca sumă de 2 numere prime și orice număr impar mai mare sau egal ca 7 poate fi scris ca sumă de 3 numere prime (figura 3). Va fi folosită funcția de test pentru numere prime din *Prim.dll*. Legarea se va face *static*.

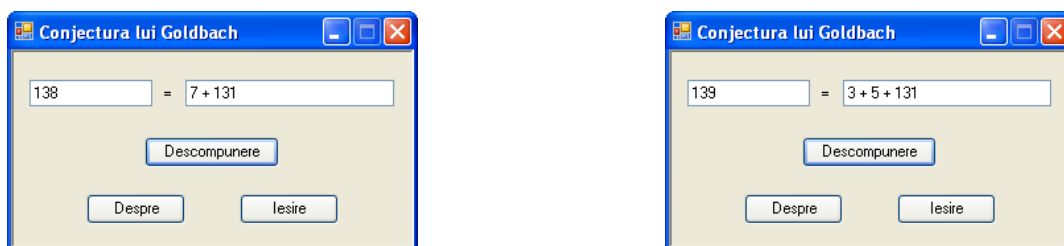


Figura 3. Exemplu de rezolvare

4.3. Modificați *Prim.dll* prin adăugarea unei metode *int NumaraPrime(int n)*, care calculează numărul numerelor prime mai mici sau egale cu n . Verificați că *Suma.exe* se execută corect după modificarea DLL-ului.

4.4. Realizați un program executabil care încarcă *dinamic* biblioteca *Prim.dll* și apelează metoda *NumaraPrime*.

4.5. (opțional) Realizați un program executabil care să afișeze graficul funcției:

$$f(n) = \text{numărul de numere prime } \leq n, n > 0.$$

precum și o aproximare a acestui număr, de forma:

$$g(n) = \frac{n}{\log n}, n > 2.$$

Pentru calculul exact, se va utiliza metoda *int NumaraPrime(int n)* din *Prim.dll*. Pentru aproximare, se va crea un DLL numit *Aproximare*, cu o clasă cu același nume care să conțină o metodă *double XLogX(double x)*. Legarea se va face *dinamic*. Verificați tratarea excepțiilor în cazul utilizării unui DLL cu o semnătură incorectă a metodei *XLogX*.

Indicații. Pentru fiecare metodă din DLL-uri, este utilă definirea unei metode corespunzătoare în programul principal. De exemplu, pentru metoda *EstePrim* din *Prim.dll*, se poate crea o metodă de tipul:

```
private bool MyPrimEstePrim(int n)
```

care să apeleze metoda EstePrim și să returneze rezultatul. În continuare, în program se va apela direct metoda MyPrimEstePrim. Încărcarea DLL-ului trebuie făcută *o singură dată*, la pornirea programului, chiar dacă apelul metodelor se va face ori de câte ori este necesar.

Cele două funcții nu se suprapun, doar forma lor este asemănătoare. De aceea, cele două grafice trebuie scalate diferit pe axa Y (figura 4).

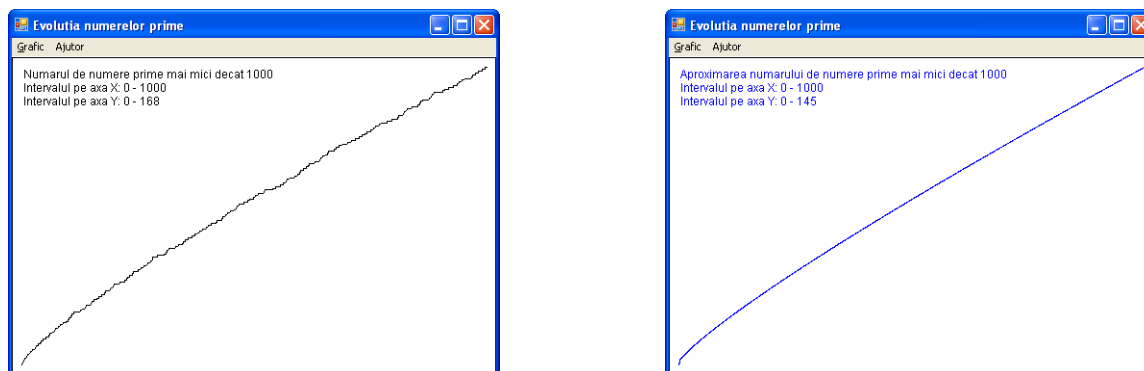


Figura 4. Exemplu de rezolvare