Laboratorul 2

Stilul de scriere a codului. Tratarea excepţiilor

1. Obiective
2. Stilul de scriere a codului
3. Tratarea excepţiilor
4. Aplicaţii
A1. Interfaţa cu utilizatorul
A2. Interfaţa grafică cu utilizatorul în Microsoft Visual Studio .NET
A3. Elemente de C#

1. Obiective

Obiectivele laboratorului 2 sunt următoarele:

- Sublinierea importanței unui stil unitar de scriere a codului într-o firmă care dezvoltă produse software;
- Descrierea standardului de scriere a codului pe care îl vom utiliza în ghidul de aplicații de ingineria programării;
- Explicarea modalităților de tratare a excepțiilor în C#;
- Prezentarea unor aspecte legate de dezvoltarea de aplicații cu interfață grafică;
- Prezentarea modului de lucru cu proprietăți C#.

2. Stilul de scriere a codului

Unul din scopurile urmărite la scrierea programelor trebuie să fie întreținerea ulterioară a codului, adică facilitarea modificărilor și completărilor viitoare, foarte probabil de către persoane diferite decât autorul inițial. De asemenea, unele studii au arătat că după 6 luni de la scrierea unui program, acesta îi apare la fel de străin autorului ca și un program scris de altcineva.

Unul din aspectele principale ale codului ușor de întreținut este posibilitatea de a găsi anumite bucăți de cod și de a le modifica fără a afecta celelalte secțiuni. Claritatea este esențială. Altfel, în cazul programelor de mari dimensiuni, așa cum sunt majoritatea situațiilor în cazul produselor software comerciale, în locul lucrului efectiv pentru adăugarea de funcționalități se va pierde timpul încercându-se să se găsească porțiunile relevante de cod care trebuie modificate.

Formatarea codului poate simplifica înțelegerea structurii semantice sau poate cauza confuzie. Poate chiar ascunde defecte greu de depistat, de exemplu:

```
bool error = DoSomething();
if (error)
    Console.WriteLine("Eroare");
    Environment.Exit(1);                /// !!!
```

Nu contează cât de bine este proiectat un program; dacă prezentarea sa arată neglijent, va fi neplăcut de lucrat cu el.

2.1. Acoladele

Există două modalități principale de plasare a acoladelor.

Stilul Kernighan și Ritchie este bazat pe dorința de a afișa cât mai multe informații într-un mod compact:

```
int KernighanRitchie() {
    int a = 0, b = 0;
    while (a != 10) {
        a++;
        b--;
    }
    return b;
}
```

Acest stil poate fi folosit la prezentări de cod sau în situații în care spațiul disponibil pentru afișarea codului este redus, de exemplu într-un material tipărit.

Stilul extins sau *stilul Allman* este recomandat de Microsoft pentru limbajul C#: 

```
int Extended()
{
    int a = 0, b = 0;
    while (a != 10)
    {
        a++;
        b--;
    }
    return b;
}
```

Avantajul principal al acestuia este claritatea, deoarece blocurile de cod sunt evidențiate prin alinierea acoladelor. Este stilul pe care îl vom utiliza în laboratoarele de IP.

2.2. Standarde de programare

Mulți programatori fără experiență industrială, deși foarte buni, refuză la început aplicarea unor standarde impuse. Dacă programul este corect, ei nu înțeleg de ce trebuie aliniat altfel sau de ce trebuie schimbate numele variabilelor sau metodelor.

Este important de avut în vedere faptul că *nu există un stil perfect*, deci războaiele privind cea mai bună formatare nu pot fi câștigate. Toate stilurile au argumente pro și contra. Majoritatea firmelor serioase de software au standarde interne de scriere a programelor, care definesc regulile pentru prezentarea codului. Aceste standarde cresc calitatea programelor și sunt importante deoarece toate proiectele livrate în afara organizației vor avea un aspect îngrijit și coerent, de parcă ar fi fost scrise de aceeași persoană. Existența mai multor stiluri distincte într-un proiect indică lipsa de profesionalism.

Faptul că un programator crede că stilul său propriu este cel mai frumos și cel mai ușor de înțeles nu are nicio importanță. Un stil care unui programator îi pare în mod evident cel mai bun poate reprezenta o problemă pentru altul. De exemplu:

```
using System . Windows . Forms;
namespace LabIP {
  public class HelloWorld:System . Windows . Forms . Form {
    public HelloWorld ( ) {
    InitializeComponent ( );
      }
    protected override void Dispose ( bool disposing ) {
    if( disposing ) {
    if( components != null ) {
    components . Dispose ( );
      }
      }
    base . Dispose ( disposing );
      }
    static void Main ( ) {
    Application . Run ( new HelloWorld ( ) );
      }
    private void button1_Click ( object sender , System . EventArgs e ) {
    string STRING = "Hello World!";
    display ( STRING );
      }
    private void display ( string STR ) {
    MessageBox . Show ( STR , ":-)" );
      }
    }
  }
```

Acest program nu a fost aliniat aleatoriu, ci folosind opțiunile din mediul Visual Studio: *Tools → Options → Text Editor → C# → Formatting*. În același mod, stilul personal al unui programator poate părea la fel de ciudat altuia.

Beneficiile adoptării unui stil unitar de către toți membrii unei organizații depășesc dificultățile inițiale ale adaptării la un stil nou. Chiar dacă nu este de acord cu standardul impus, un

programator profesionist trebuie să se conformeze. După ce va folosi un timp stilul firmei, se va obișnui cu el și i se va părea perfect natural.

Pentru laboratoarele de IP, vom utiliza un standard bazat pe recomandările Microsoft pentru scrierea programelor C#.

Este bine ca regiunile de program să fie delimitate cu ajutorul cuvântului cheie region, de exemplu:

```
#region Fields
private DateOfBirth _dob;
private Address _address;
#endregion
```

Fields

Dacă toate secțiunile unei clase sunt delimitate pe regiuni, pagina ar trebui să arate în felul următor atunci când toate definițiile sunt colapsate:

```
using System;

namespace EmployeeManagement
{
    public class Employee
    {
        Private Member Variables

        Private Properties

        Private Methods

        Constructors

        Public Properties

        Public Methods
    }
}
```

2.3. Convenții pentru nume

Cheia alegerii unor nume potrivite este înțelegerea rolului construcțiilor respective. De exemplu, dacă nu putem găsi un nume bun pentru o clasă sau pentru o metodă, ar fi util să ne întrebăm dacă știm într-adevăr la ce folosește aceasta sau dacă chiar este necesar să existe în program. Dificultățile la alegerea unui nume potrivit pot indica probleme de proiectare.

Un nume trebuie să fie:

- *Descriptiv:* Oamenii își păstrează deseori percepțiile inițiale asupra unui concept. Este importantă deci crearea unei impresii inițiale corecte despre datele sau funcționalitățile unui program prin alegerea unor termeni care să descrie exact semnificația și rolul acestora. Numele trebuie alese din perspectiva unui cititor fără cunoștințe anterioare, nu din perspectiva programatorului;
- *Adecvat:* Pentru a da nume clare, trebuie să folosim cuvinte din limbajul natural. Programatorii au tendința să utilizeze abrevieri și prescurtări, însă acest lucru conduce la

denumiri confuze. Nu are importanță faptul că un identificator este lung câtă vreme este lipsit de ambiguitate. Denumirea st nu este o alegere potrivită pentru conceptul numarStudenti. Regula de urmat este: *trebuie preferată claritatea față de laconism*. Excepțiile sunt contoarele de bucle, de exemplu clasicul **for** (**int** i = 0; i < length; i++). Acestea de multe ori nu au o semnificație de sine stătătoare, sunt construcții specifice (idiomuri) ale limbajelor evoluate din C și de obicei se notează cu o singură literă: i, j, k etc.;

- *Coerent:* Regulile de numire trebuie respectate în tot proiectul și trebuie să se conformeze standardelor firmei. O clasă precum cea de mai jos nu prezintă nicio garanție de calitate:

```
class badly_named : MyBaseClass
{
   public void doTheFirstThing();
   public void DoThe2ndThing();
   public void do_the_third_thing();
}
```

Pentru descrierea tipurilor de nume, există în engleză o serie de termeni care nu au un echivalent exact în limba română:

- *Pascal case:* Primul caracter al tuturor cuvintelor este o majusculă iar celelalte caractere sunt minuscule, de exemplu: **N**umar**S**tudenti;
- *Camel case:* Pascal case cu excepția primului cuvânt, care începe cu literă mică, de exemplu: **n**umar**S**tudenti.

Pentru denumirea conceptelor dintr-un program C#, vom adopta convențiile din tabelul 1.

Tabelul 1. *Convenții pentru denumirea conceptelor dintr-un program C#*

Concept	**Convenție**	**Exemple**
Namespace-uri	Pascal case	namespace **LaboratorIP**
Clase	Pascal case	class **HelloWorld**
Interfețe	Pascal case precedat de *I*	interface **IEntity**
Metode	Pascal case	void **SayHello**()
Variabile locale	Camel case	int **totalCount** = 0;
Variabile booleene	Prefixate cu *is*	bool **isModified**;
Parametrii metodelor	Camel case	void SayHello(string **name**)
Câmpuri private [1]	Camel case precedat de *underscore*	string **_address**;
Proprietăți [2]	Pascal case	**Address**
Constante, câmpuri *readonly* publice [3]	Pascal case	const int **MaxSpeed** = 100;
Controale pentru interfața grafică [4]	Camel case precedat de *tipul controlului*	**buttonOK** **checkBoxTrigonometric** **comboBoxFunction**
Excepții	Pascal case cu terminația *Exception*	**MyException** **PolynomialException**

[1] Până la versiunea Visual Studio 6.0, programatorii utilizau în C++ notația maghiară a lui Simonyi pentru variabile, care indica și tipul acestora, de exemplu nAge pentru int. Pentru variabilele membru se folosea prefixul m_, de exemplu m_nAge. Pentru limbajul C#, Microsoft nu mai recomandă utilizarea acestor notații. Pentru câmpurile private există câteva avantaje la prefixarea cu *underscore*:

- câmpurile clasei vor avea o notație diferită de variabilele locale;
- câmpurile clasei vor avea o notație diferită de parametrii metodelor, astfel încât se vor evita situațiile de inițializare de genul this.x = x, unde this.x este câmpul iar x este parametrul metodei;
- în *IntelliSense*, la apăsarea tastei _ vor apărea grupate toate câmpurile.

Avantajul utilizării acestei convenții se manifestă mai ales în situații precum aceea de mai jos:

```
private int description; // ortografie corectă

public Constructor(int descripton) // ortografie incorectă pentru "description"
{
    this.description = description; // ortografie corectă în ambele părți, câmpul rămâne 0
}
```

[2] Proprietățile sunt detaliate în secțiunea A3.2.

[3] În mare, tot ce e public într-o clasă trebuie să înceapă cu literă mare

[4] Această notație are avantajul că, deși numele sunt mai lungi, sunt lipsite de ambiguitate. Există și stilul prefixării cu o abreviere de 3 litere, de exemplu btn pentru Button, ckb pentru CheckBox etc. Pentru controale mai puțin uzuale, semnificațiile prefixelor nu mai sunt evidente: pbx, rdo, rbl etc.

3. Tratarea excepțiilor

Tratarea erorilor se făcea în C prin returnarea unei valori, de obicei int, care semnifica un cod de eroare. De exemplu, dacă o funcție trebuia să deschidă un fișier, ea putea întoarce 0 dacă totul a funcționat normal, respectiv codul de eroare 1 dacă fișierul nu exista. În funcția apelantă, programatorul trebuia să trateze codul returnat:

```
int err = OpenFile(s);
```

Cu toate acestea, programatorul era liber să apeleze funcția direct, OpenFile(s), fără a mai testa valoarea returnată. Programul putea fi testat cu succes, deoarece fișierul exista, însă putea să nu mai funcționeze după livrare.

Majoritatea funcțiilor *Windows API* returnează un cod dintr-o listă cu sute de valori posibile, însă puțini programatori testează aceste valori individual.

Tratarea excepțiilor a apărut pentru a da posibilitatea programelor să surprindă și să trateze erorile într-o manieră elegantă și centralizată, permițând separarea codului de tratare a erorilor de codul principal al programului, ceea ce face codul mai lizibil. Astfel, este posibilă tratarea:

- tuturor tipurilor de excepții;
- tuturor excepțiilor de un anume tip;

- tuturor excepțiilor de tipuri înrudite.

Odată ce o excepție este generată, ea nu poate fi ignorată de sistem. Funcția care detectează eroarea poate să nu fie capabilă să o trateze și atunci se spune că „aruncă” (throw) o excepție. Totuși, nu putem fi siguri că există un caz de tratare pentru orice excepție. Dacă există o rutină potrivită, excepția este tratată, dacă nu, programul se termină. Rutinele de tratare a excepțiilor pot fi scrise în diverse feluri, de exemplu examinează excepția și apoi închid programul sau re-aruncă excepția.

Structura blocurilor try-catch pentru tratarea excepțiilor este următoarea:

```
FuncțieApelată
{
    ...
    if (condiții)
        throw ...
            ...
}

FuncțieApelantă
{
    ...
    try
    {
        FuncțieApelată();
    }
    catch (Exception e) // catch fără parametru dacă nu interesează detaliile excepției
    {
        // cod pentru tratarea excepției, care prelucrează parametrul e
    }
    ...
}
```

Codul care ar putea genera o excepție se introduce în blocul try. Excepția este aruncată (throw) din funcția care a fost apelată, direct sau indirect. Excepțiile care apar în blocul try sunt captate în mod normal de blocul catch care urmează imediat după acesta. Un bloc try poate fi urmat de unul sau mai multe blocuri catch. Dacă se execută codul dintr-un bloc try și nu se aruncă nicio excepție, toate rutinele de tratare sunt ignorate, iar execuția programului continuă cu prima instrucțiune de după blocul (sau blocurile) catch.

În C#, tipul trimis ca parametru este de obicei Exception:

```
try
{
    FunctieApelata();
}
catch (Exception ex)
{
    ...
}
```

De cele mai multe ori, ne interesează proprietatea Message (de tip string) a unui astfel de obiect, care arată cauza erorii. Textul respectiv este precizat în funcția care aruncă excepția.

```
private void FunctieApelata(int a)
{
    if (a == 0)
        throw new Exception("Argument zero");
}

private void FunctieApelanta()
{
    FunctieApelata(0);
}
```

Dacă excepția nu este tratată, va apărea un mesaj de atenționare cu textul dorit (figura 1).

Figura 1. *Mesaj de excepţie*

Mesajul de eroare trebuie preluat în program în blocul catch:

```
private void FunctieApelanta()
{
    try
    {
        FunctieApelata(0);
    }
    catch (Exception ex)
    {
        // ex.Message este "Argument zero"
    }
}
```

Programatorul își poate defini propriile tipuri de excepții, în cazul în care are nevoie să trimită informații suplimentare privind excepția. Acestea trebuie însă derivate din clasa Exception:

```
public class MyException: Exception
{
    public int _info; // informație suplimentară

    // în constructor se apelează și constructorul clasei de bază
    public MyException(int val) : base()
    {
        _info = val;
    }
}
```

Utilizarea acestui tip se face astfel:

```
throw new MyException(3);
```

După un try pot exista mai multe blocuri catch. Ele trebuie dispuse în ordinea inversă a derivării tipurilor, de la particular la general:

```
try
{
    FunctieApelata(x);
}
catch (MyException myex)
{
    ...
}
catch (Exception ex)
{
    ...
}
```

Altă ordine nu este permisă, eroarea fiind descoperită la compilare: *A previous catch clause already catches all exceptions of this or a super type ('System.Exception').*

3.1. Tratarea excepțiilor pe firul de execuție al aplicației

Uneori pentru a fi siguri că nu am lăsat vreo excepție netratată, putem trata global toate excepțiile apărute pe firele de execuție ale aplicației, în maniera descrisă în continuare:

```
static class Program
{
   static void Main()
   {
      // Adăugarea unui event handler pentru prinderea excepțiilor
      // din firul principal al interfeței cu utilizatorul
      Application.ThreadException += new ThreadExceptionEventHandler(OnThreadException);

      // Adăugarea unui event handler pentru toate firele de execuție din appdomain
      // cu excepția firului principal al interfeței cu utilizatorul
      AppDomain.CurrentDomain.UnhandledException +=  new UnhandledExceptionEventHandler(CurrentDomain_UnhandledException);

      Application.EnableVisualStyles();
      Application.SetCompatibleTextRenderingDefault(false);
      Application.Run(new MyForm());    // MyForm este fereastra principală a programului
   }

   // Tratează excepțiile din firul principal al interfeței cu utilizatorul
   static void OnThreadException(object sender, ThreadExceptionEventArgs t)
   {
      // Afișează detaliile excepției
      MessageBox.Show(t.Exception.ToString(), "OnThreadException");
   }

   // Tratează excepțiile din toate celelalte fire de execuție
   static void CurrentDomain_UnhandledException(object sender, UnhandledExceptionEventArgs e)
   {
      // Afișează detaliile excepției
      MessageBox.Show(e.ExceptionObject.ToString(), "CurrentDomain_UnhandledException");
   }
}
```

4. Aplicații

4.1. Realizați interfața grafică a unei aplicații Windows pentru rezolvarea de ecuații polinomiale de gradul I și II, precum și de ecuații trigonometrice simple (figura 2).

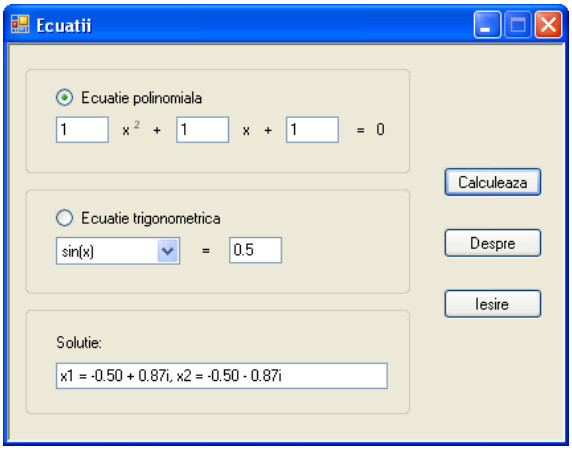

Figura 2. *Exemplu de rezolvare*

4.2. Creați clasele pentru rezolvarea celor două tipuri de ecuații (figura 3). Se vor arunca două tipuri de excepții, conform structurii de mai jos. Fiind două tipuri diferite de ecuații, vom avea o interfață IEquation cu metoda Solve, din care vor fi derivate PolyEquation și TrigEquation.

În clasa principală, evenimentul de tratare a apăsării butonului *Calculeaza* va testa tipul de ecuație.

Pentru o ecuație polinomială, va citi coeficienții ecuației și va instanția un obiect corespunzător:

```
IEquation eq = new PolyEquation(x2, x1, x0);
textBoxSolutie.Text = eq.Solve();
```

Pentru o ecuație trigonometrică, depinzând de tipul de funcție trigonometrică:

```
eq = new TrigEquation(TrigEquation.TrigonometricFunction.Sin, arg);
textBoxSolutie.Text = eq.Solve();
```

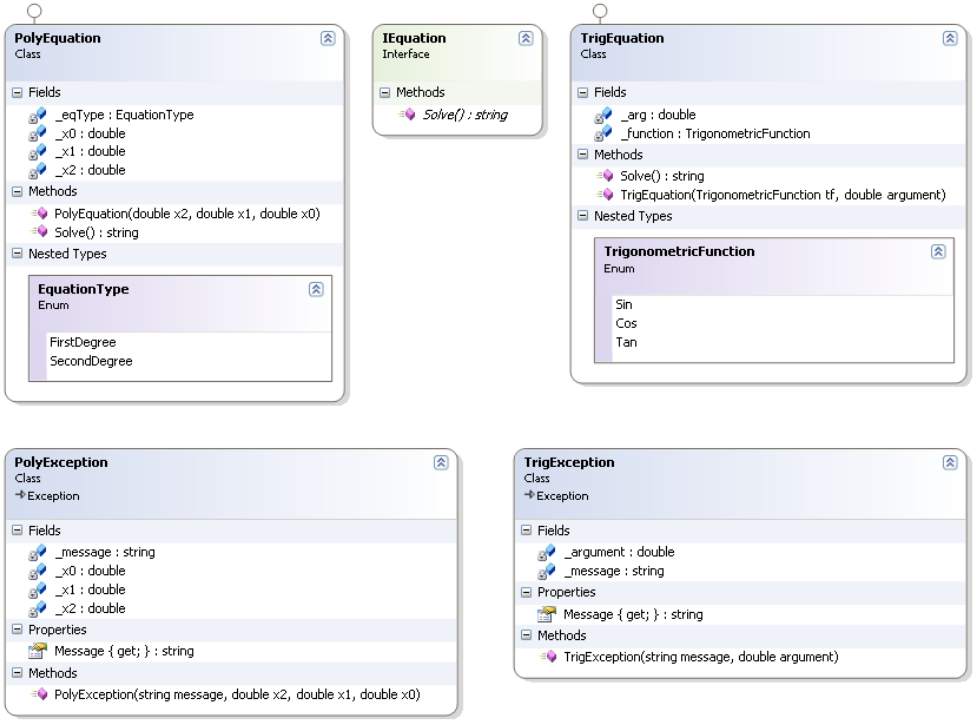

Figura 3. *Exemplu de rezolvare: clasele soluției*

Fiecărui tip de ecuație îi va corespunde propria clasă de excepție: PolyException, respectiv TrigException.

Pentru ecuația polinomială se vor considera următoarele excepții:

- pentru $x_2 = x_1 = x_0 = 0$: „O infinitate de soluții”;
- pentru $x_2 = x_1 = 0$ și $x_0 \neq 0$: „Nicio soluție”.

Pentru ecuația trigonometrică, excepția „Argument invalid” va fi aruncată când $|arg| > 1$, însă numai pentru funcțiile *arcsin* și *arccos*.

În evenimentul butonului vom avea un singur bloc try, urmat de trei blocuri catch: pentru cele două tipuri de excepție, precum și pentru excepțiile generice, rezultate de exemplu la încercarea de a citi coeficienții, dacă utilizatorul introduce caractere nenumerice.

Indicație: codul pentru rezolvarea ecuației polinomiale este prezentat în continuare, unde $\Delta = x_1^2 - 4 \cdot x_2 \cdot x_0$:

```
if (_x2 == 0)
{
        _eqType = EquationType.FirstDegree;
        // soluție: -_x0 / _x1
}
else if (delta > 0)
{
        double sqrtDelta = Math.Sqrt(delta);
        double sol1 = (-_x1 + sqrtDelta) / (2.0 * _x2);
        double sol2 = (-_x1 - sqrtDelta) / (2.0 * _x2);
        // soluții: sol1, sol2
}
```

```
else if (delta == 0)
{
        double sol = (-_x1) / (2.0 * _x2);
        // soluție: sol
}
else
{
        double rsol = -_x1 / (2.0 * _x2);
        double isol = Math.Sqrt(-delta) / (2.0 * _x2);
        // soluții: rsol ± isol
}
```

Atenție: scopul laboratorului este lucrul cu excepții, nu rezolvarea propriu-zisă a ecuațiilor!