Laboratorul 8

Şabloanele de proiectare Singleton şi Prototip

1. Obiective

Obiectivele laboratorului 8 sunt următoarele:

- Implementarea unui program după şablonul de proiectare *Singleton*;
- Implementarea unui program după şablonul de proiectare *Prototip* (engl. ”Prototype”).

2. Şablonul creaţional *Singleton*

Scopul şablonului *Singleton* este să garanteze faptul că o clasă poate avea doar o singură instanţă şi să asigure un punct global de acces la ea.

În unele situaţii, este important ca o clasă să aibă o singură instanţă, de exemplu atunci când avem dispozitive hardware sau accesorii ataşate unui calculator şi dorim să prevenim accesul concurent la acestea. Alte situaţii sunt cele în care se doreşte lucrul cu registrul Windows (unic) sau atunci când se lucrează cu un bazin (engl. “pool”) de fire de execuţie. În general, şablonul este util când o resursă unică trebuie să aibă un corespondent unic care o accesează din program.

Este nevoie deci de o modalitate de a împiedica instanţierile multiple ale unei clase şi de a asigura o metodă unică globală pentru accesarea instanţei. Avantajul faţă de utilizarea unor clase statice sau cu proprietăţi statice este faptul că *Singleton*-ul poate fi derivat şi astfel clienţii îi pot extinde funcţionalitatea fără a fi nevoiţi să modifice clasa existentă.

Diagrama de clase este prezentată în figura 1.

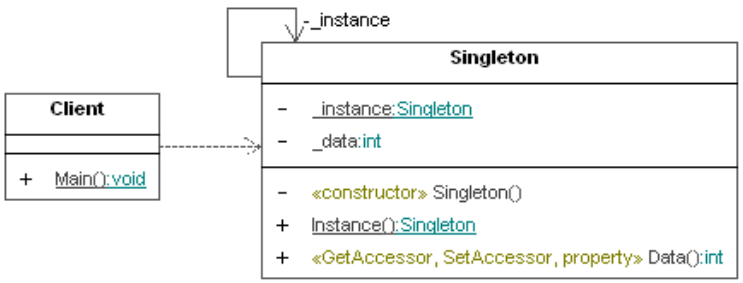

Figura 1. *Diagrama de clase a şablonului Singleton*

Clientul poate accesa poate accesa instanța unică a clasei Singleton utilizând metoda statică Instance.

2.1. Exemplu de implementare

Codul C# corespunzător diagramei UML anterioare este prezentat mai jos.

Clasa Singleton

```
class Singleton
{
    private static Singleton _instance;

    private int _data;  // datele propriu-zise ale clasei

    public int Data
    {
        get { return _data; }
        set { _data = value; }
    }

    private Singleton() // protected dacă avem nevoie de clase derivate
    {
    }

    public static Singleton Instance()
    {
        // Inițializare întârziată ("lazy initialization")
        if( _instance == null )
            _instance = new Singleton();

        return _instance;
    }
}
```

Clientul

```
public class Client
{
    public static void Main()
    {
        // constructorul este privat, nu se poate folosi "new"
        Singleton s1 = Singleton.Instance();
        s1.Data = 5;

        Singleton s2 = Singleton.Instance();
        Console.WriteLine("Rezultat: {0}", s2.Data);    // "Rezultat: 5"

        Console.ReadLine();
    }
}
```

Trebuie precizat că în situațiile în care mai multe fire de execuție pot accesa simultan secțiunea de inițializare, trebuie incluse mecanisme suplimentare de sincronizare.

3. Șablonul creațional *Prototip*

Scopul șablonului *Prototip* este să specifice tipurile de obiecte care pot fi create folosind o instanță prototip și să creeze noi obiecte prin copierea acestui prototip.

Există situații în care instanțierea unui obiect în mod normal, folosind inițializarea stării interne în constructor, este costisitoare din punct de vedere al timpului sau resurselor de calcul. De aceea, dacă este nevoie de mai multe astfel de obiecte în sistem, se poate utiliza un obiect prototip pentru a crea noi obiecte, prin copierea, în loc de calcularea de fiecare dată a valorilor datelor interne. Ideea de bază este utilizarea unei instanțe tipice pentru a crea o altă instanță înrudită.

Șablonul folosește o metodă caracteristică numită `Clone` pentru crearea cópiilor unui obiect. Nu se specifică însă dacă clonarea este *superficială* (engl. "shallow") sau *profundă* (engl. "deep"), aceasta depinde de tipul datelor copiate. Clonarea superficială este mai simplă și se folosește de obicei când câmpurile sunt de tip valoare (tipuri primitive, structuri). Pentru tipuri referință, acest tip de clonare copiază doar referințele și de aceea, în aceste cazuri se poate prefera copierea profundă, care copiază recursiv toate valorile.

De exemplu, dacă obiectul are două câmpuri primitive, `int` și `double`, este suficientă clonarea superficială. Dacă se mai adaugă un câmp vector, `int[]`, clonarea superficială ar copia doar referința: obiectul prototip și copia ar referenția de fapt același vector. Clonarea profundă creează în acest caz doi vectori distincți pentru cele două obiecte.

Una din principalele probleme ale șablonului este legată de tipul clonării, deoarece varianta recomandată depinde de situație.

Diagrama de clase este cea din figura 2.

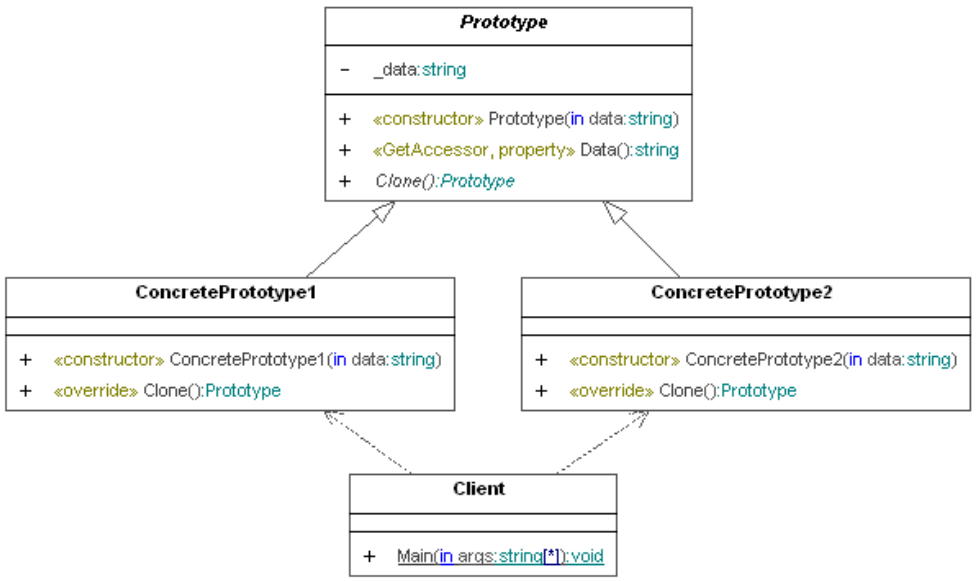

Figura 2. *Diagrama de clase a șablonului Prototip*

3.1. Exemplu de implementare

Codul C# corespunzător diagramei UML anterioare este prezentat mai jos.

Prototipul abstract

```
abstract class Prototype
{
   private string _data;

   public Prototype(string data)
   {
      _data = data;
   }

   public string Data
   {
      get
      {
         return _data;
      }
   }

   abstract public Prototype Clone();
}
```

Prototipurile concrete

```
class ConcretePrototype1 : Prototype
{
   public ConcretePrototype1(string data) : base(data)
   {
   }

   override public Prototype Clone()
   {
      // copie superficială
      return (Prototype)this.MemberwiseClone();
   }
}


class ConcretePrototype2 : Prototype
{
   public ConcretePrototype2(string data) : base(data)
   {
   }
```

```
  override public Prototype Clone()
  {
    // copie superficială
    return (Prototype)this.MemberwiseClone();
  }
}
```

Clientul

```
class Client
{
  public static void Main(string[] args)
  {
    // se creează două prototipuri și se clonează fiecare

    ConcretePrototype1 prototype1 = new ConcretePrototype1("Proto1");
    ConcretePrototype1 clone1 = (ConcretePrototype1)p1.Clone();
    Console.WriteLine("Cloned: {0}", c1.Data);

    ConcretePrototype2 prototype2 = new ConcretePrototype2("Proto2");
    ConcretePrototype2 clone2 = (ConcretePrototype2)p2.Clone();
    Console.WriteLine("Cloned: {0}", c2.Data);

    Console.ReadLine();
  }
}
```

4. Aplicații

4.1. Să se simuleze lucrul cu o imprimantă (figura 3). Fiecărui document îi este asociată o mărime. Imprimanta are o coadă de documente ce urmează a fi tipărite. Când se trimite un document la imprimare, dacă această coadă este vidă, fișierul începe să fie tipărit. Timpul necesar tipăririi este proporțional cu mărimea sa. Când coada imprimantei nu este vidă, documentul este doar introdus în coadă. La terminarea tipăririi unui document, se preia următorul din coadă (dacă există). Implementarea se va realiza utilizând șablonul *Singleton*.

Diagrama de clase a aplicației este prezentată în figura 4.

Indicație: pentru simularea tipăririi se recomandă folosirea unui control Timer, descris în anexa laboratorului 2. Evenimentul Tick este tratat o dată la un interval de timp, specificat în milisecunde de proprietatea Interval. De exemplu, dacă Interval = 500, codul evenimentului Tick va fi executat în mod repetat, de două ori pe secundă. Proprietatea Enabled arată dacă *timer*-ul e activat sau nu (true/false).

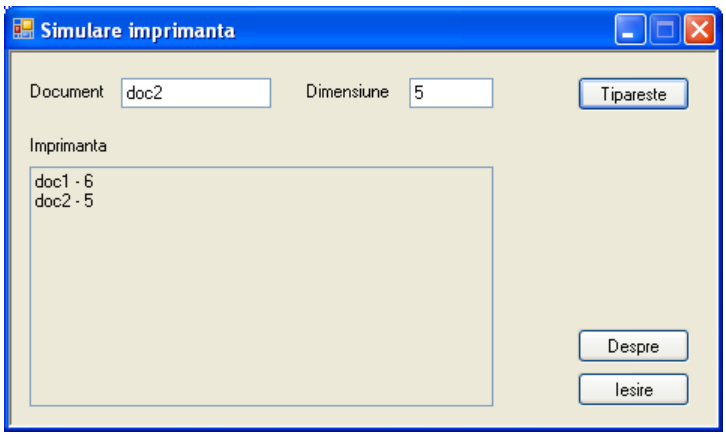

Figura 3. *Exemplu de rezolvare: interfața cu utilizatorul*

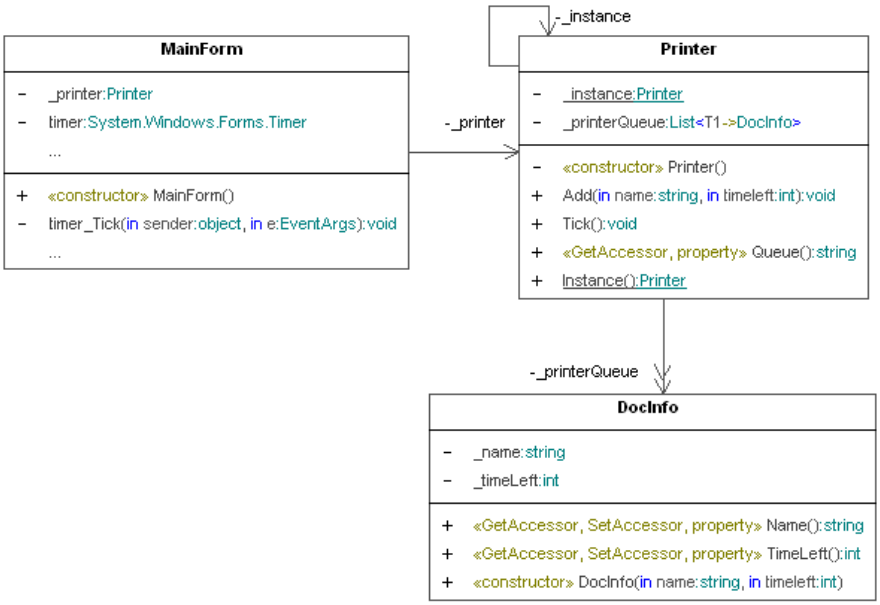

Figura 4. *Exemplu de rezolvare: diagrama de clase*

Proprietatea Queue din clasa Printer returnează conținutul cozii de activități ale imprimantei, pentru a fi afișată ca atare de client (MainForm). În acest fel, clientul nu mai trebuie să depindă de clasa DocInfo, care reprezintă obiectele cu care lucrează intern doar clasa Printer.

4.2. Să presupunem că avem un joc în care utilizatorul împușcă monștri pe ecran (figura 5). Există mai multe tipuri de monștri, fiecare cu propriile caracteristici: imagine, culoare, număr de vieți etc. Pe lângă acestea, fiecare monstru are implementat un modul de inteligență artificială, a cărui inițializare necesită multe resurse.

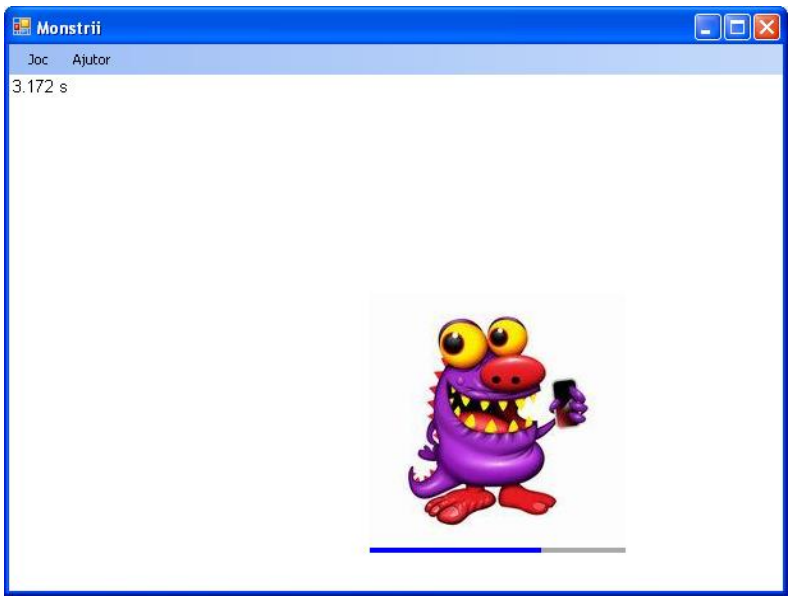

Figura 5. *Exemplu de rezolvare: interfața cu utilizatorul*

Scheletul programului este dat, la fel și o clasă pentru un monstru implementată în *MonsterSprite.dll*, cu structura din diagrama din figura 6.

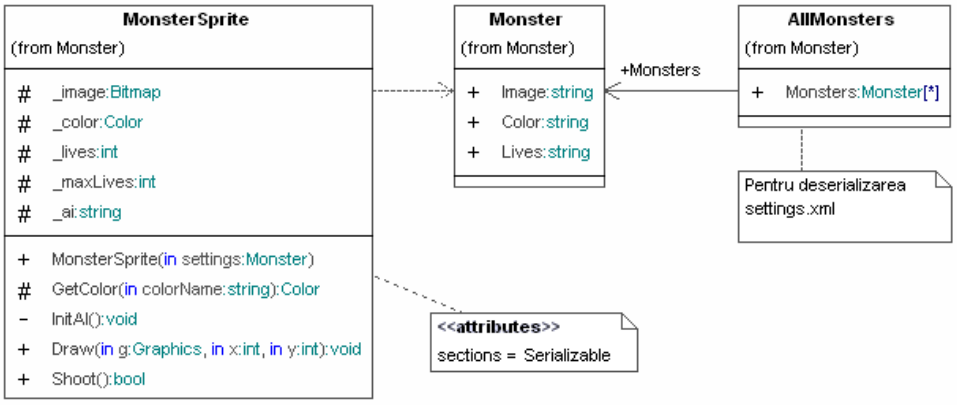

Figura 6. *Diagrama claselor din DLL*

Scopul aplicației este să evitați inițializarea de fiecare dată a modulului costisitor de IA (metoda InitAI), înlocuind instanțierea unui obiect MonsterSprite cu clonarea sa și modificarea caracteristicilor variabile. În acest caz, va exista un PrototypeManager care va asigura crearea monștrilor, înlocuind o instrucțiune de tipul:

```
_mainMonster = new MonsterSprite(_listMonsters[_monsterType]);
```

cu:

```
_mainMonster = _prototypeManager.GetMonster(_monsterType);
```

DLL-ul conținând clasa MonsterSprite se va folosi ca atare, fără modificări.

Un exemplu de fișier cu setări este *settings.xml*.

Indicație: pentru a face rapid o copie profundă a unui obiect, se poate utiliza serializarea într-o metodă de tipul:

```
public static object Clone(object obj)
{
    object objClone = null;
    MemoryStream memory = new MemoryStream();
    BinaryFormatter binForm = new BinaryFormatter();
    binForm.Serialize(memory, obj);
    memory.Position = 0;
    objClone = binForm.Deserialize(memory);
    return objClone;
}
```

În vederea utilizării acestei modalități de copiere, trebuie incluse în proiect *namespace*-urile System.Runtime.Serialization și System.IO, iar clasa trebuie decorată cu atributul [Serializable]. Această metodă generală trebuie particularizată la situația concretă a aplicației.