Spremanje stanja programa

Paradžiković, Kristijan

Sveučilište Josipa Jurja Strossmayera u Osijeku

Fakultet primijenjene matematike i informatike

Sveučilišni prijediplomski studij Matematika i računarstvo

Spremanje stanja programa

Završni rad

Mentor:

izv. prof. dr. sc. Domagoj Matijević

Kandidat:

Kristijan Paradžiković

Komentor:

dr. sc. Luka Borozan

Osijek, 2023

Sadržaj

1 | Uvod

Spremanje stanja programa je izuzetno važan koncept koji omogućuje pohranu i razmjenu podataka, prijenos podataka između sustava i integraciju s drugim servisima. Mogućnost očuvanja trenutnog stanja aplikacija omogućuje korisnicima da se vrate točno tamo gdje su stali i osigurava postojanost podataka. Jedan od ključnih aspekata ovog procesa je odabir prikladnoga formata za spremanje tih podataka.

U ovom radu objasnit ćemo implementaciju i primjenu C# aplikacije za spremanje stanja programa pomoću XML-a. XML (Extensible Markup Language) je jezik koji se koristi za pohranu i prijenos podataka te je lako čitljiv ljudima i računalima. XML format omogućuje pohranu višerazinskih i složenih objekata te podržava hijerarhijsku organizaciju podataka. Također, XML osim oznaka omogućuje i dodavanje atributa kako bismo što bolje mogli opisati podatke. Sve to ga čini izuzetno pogodnim za ovaj projekt.

Osim XML formata, postoji niz drugih formata i pristupa za spremanje stanja programa. Jedna od popularnijih alternativa je JSON (JavaScript Object Notation). JSON je format koji se često koristi u web aplikacijama i REST API-ima. Nudi čitljivu sintaksu i dobru podršku u većini programskih jezika. Osim toga, binarni formati kao što je BSON (Binary JSON) također nude brže čitanje i pisanje, ali obično nisu tako ljudski čitljivi kao XML ili JSON.

Kod projekta dostupan je na:
`https://github.com/KristijanParadz/ProgramStateSaver`

2 | Osnovni pojmovi

2.1 C# compiler i .NET okruženje

Pri izgradnji aplikacije C# compiler prevodi izvorni kod u takozvani common intermediate language (CIL). Računalo ne može izvršavati CIL, on je zamišljen kao međukorak između izvornoga koda i strojnoga jezika. Za izvršavanje koda u .NET okruženju brine se takozvani common language runtime (CLR). CLR koristi just-in-time compiler kako bi preveo CIL u strojni jezik tijekom izvođenja programa.

Slika 2.1: Prevođenje koda

2.2 Klase i objekti

Klasa je osnovni pojam u objektno-orijentiranom programiranju i služi kao predložak za kreiranje objekata. Objekti su instance klase. U klasi se nalaze member članovi koji predstavljaju podatke i ponašanje klase. Postoji deset vrsta membera koji se mogu pojaviti u klasi. Za ovaj rad bitni su fieldovi i propertyji jer oni predstavljaju podatke koje treba spremiti u XML. Fieldovi su varijable koje su deklarirane u bloku klase, a propertyji su metode klase kojima se može pristupiti kao fieldovima. Ova aplikacija omogućuje spremanje i rekonstrukciju objekata proizvoljnih klasa.

```
1 namespace ProgramStateSaver
2 {
3     internal class Complex
4     {
5         public double RealPart;
6         public double ImaginaryPart { get; set; }
7     }
```

```
8 }
```

Listing 2.1: Primjer klase Complex s fieldom RealPart i propertyjem ImaginaryPart

2.3 Nasljeđivanje

Nasljeđivanje je koncept u objektno-orijentiranom programiranju koji omogućuje ponovno korištenje, proširivanje i modifikaciju već postojećih klasa. U ovom projektu postoji klasa *Saveable* u kojoj je definirana sva potrebna logika za spremanje i rekonstrukciju objekata. Sve klase čiji objekti trebaju imati mogućnost spremanja samo trebaju naslijediti klasu *Saveable*.

2.4 Atributi

Atributi omogućuju dodavanje metapodataka u aplikaciju. Metapodaci sadrže informacije o tipovima podataka i member članovima tipa. Atribute kreiramo kao klase koje nasljeđuju klasu *Attribute*. Atributima možemo osigurati dodatne informacije o elementu kojega dekoriramo atributom. Elementi aplikacije koje možemo dekorirati atributom su klase, metode, fieldovi, propertyji... U ovoj aplikaciji koristit ćemo atribute pri odabiru podataka koje želimo spremati tako što ćemo fieldove i propertyje koje želimo spremati dekorirati atributom *Save*.

2.5 Refleksija

Refleksija je ključna značajka C#-a za ovaj projekt. Refleksija je koncept koji omogućuje pristup metapodacima tijekom izvođenja programa. To nam omogućuje dohvat i manipulaciju tipova podataka, fieldova i propertyja, atributa, dohvaćanje i postavljanje vrijednosti fielda odnosno propertyja, analizu generičkih tipova... Jedini nedostatak refleksije je taj što zbog dodatnih operacija pretraživanja metapodataka u vremenu izvođenja neke metode iz refleksije koje se često koriste, a nisu dovoljno optimizirane, mogu usporavati program.

2.6 Tipovi podataka

U C#-u tipovi podataka se najčešće dijele na value types i reference types. Za potrebe ovoga rada ta podjela nije previše bitna pa ćemo napraviti vlastiti podjelu na jednostavne i složene tipove. Jednostavnima ćemo zvati sve one kojima vrijednost odmah možemo zapisati u XML unutar jednoga taga, a to su: Boolean, Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64, IntPtr, UIntPtr, Char, Double, Single, String, Decimal, DateTime. Sve ostale tipove zvat ćemo složenima. Složeni tipovi su kolekcije koje mogu biti generičke i negeneričke. U generičkoj kolekciji svi elementi moraju biti istoga tipa, za razliku od negeneričkih gdje mogu biti

proizvoljnoga tipa. Ova aplikacija podržava spremanje i rekonstrukciju sljedećih generičkih kolekcija: Dictionary<>, SortedList<>, List<>, HashSet<>, SortedSet<>, Tuple<>, Stack<>, Queue<> i sljedećih negeneričkih kolekcija: Array, ArrayList, Stack, Queue, Hashtable, SortedList.

2.6.1 Array

Array je neprekidan niz bajtova u memoriji fiksne duljine i jedna je od najjednostavnijih kolekcija.

2.6.2 Tuple

Tuple je specifična struktura fiksne duljine kojoj su podaci nepromjenjivi. Najčešće se koristi kao povratni tip funkcije ako funkcija treba vratiti više parametara.

2.6.3 Stack i Queue

Stack i Queue su jednostavne strukture implementirane pomoću Arrayja. Stack radi po principu last-in, first-out odnosno element koji je zadnji dodan je prvi element koji će biti obrisan. Operacija za dodavanje u stacku naziva se push, a operacija za brisanje pop. Queue je struktura slična stacku, ali radi po principu first-in, first-out odnosno element koji je prvi dodan će biti prvi i obrisan. Operacija za dodavanje u queueu naziva se enqueue, a operacija za brisanje dequeue.

2.6.4 List i ArrayList

Lista je također implementirana pomoću Arrayja. Ona je dosta naprednija struktura od stacka i queuea jer omogućuje dodavanje i brisanje bilo gdje u listi, sortiranje, reversanje, pronalazak elementa... ArrayList je struktura identična listi osim što je negeneričkа tj. možemo reći negeneričkа verzija liste.

2.6.5 Strukture implementirane pomoću hash tablica

Sljedeće strukture o kojima ćemo govoriti implementirane su pomoću hash tablica. Hash tablica je struktura koja koristi matematičke funkcije (hash funkcije) za raspršivanje elemenata po arrayju. Prednost ove strukture nad listom je ta što omogućuje dodavanje, brisanje i pretraživanje elemenata u konstantnom vremenu, dok je pretraživanje u listi linearno.

Dictionary je struktura koja sadrži parove ključeva i vrijednosti. Ključevi moraju biti jedinstveni (ne smiju se ponavljati).

Hashtable je struktura identična Dictionaryju osim što je negeneričkа tj. možemo reći negeneričkа verzija Dictionaryja.

HashSet je struktura ekvivalentna skupu u matematici tj. ne sadrži duplikate elemenata. HashSet možemo promatrati kao Dictionary koji ima samo ključeve odnosno nema vrijednosti.

2.6.6 SortedSet

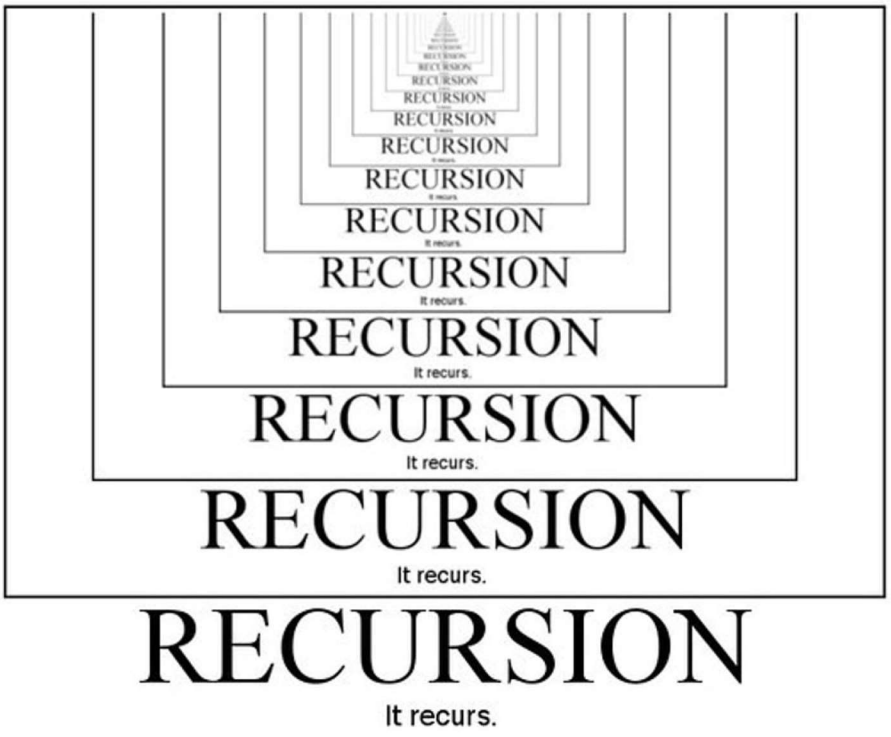
SortedSet je struktura koja sadrži jedinstvene elemente kao u HashSetu, ali sortirane. Implementiran je pomoću binarnoga stabla pretraživanja.

2.6.7 SortedList

SortedList je struktura koja sadrži parove ključeva i vrijednosti sortiranih po ključu. Implementirana je pomoću dva arrayja Keys i Values kojima se može pristupiti.

2.7 Rekurzija

Rekurzija je postupak u kojemu funkcija poziva samu sebe direktno ili indirektno. Direktno znači da funkcija poziva samu sebe u implementaciji, a indirektno da funkcija fun1 poziva neku drugu funkciju fun2 koja onda opet poziva fun1. Funkcija u ovom procesu naziva se rekurzivna funkcija. Rekurzivna funkcija rješava problem tako da ga podijeli na manje potprobleme. Kod rekurzivnih funkcija bitno je dati bazni uvjet kada rekurzija terminira kako se funkcija ne bi beskonačno pozivala. U ovom projektu koristit ćemo indirektne rekurzivne funkcije za čitanje i spremanje višerazinskih (ugniježđenih) kolekcija.



Slika 2.2: Rekurzija(iz [5])

3 | Spremanje i čitanje podataka

3.1 Pisanje i čitanje XML-a u C#-u

Za pisanje po XML-u koristit ćemo ugrađeni *XmlWriter*, a za čitanje XML-a *XmlReader* iz namespace-a *System.Xml* u C#-u.

3.2 Proces spremanja

Proces spremanja odvija se u sljedeća tri koraka.

1. Dohvat fieldova i propertyja klase pomoću funkcija *GetFields* i *GetProperties* iz refleksije (iz *System.Reflection* namespace-a).
2. Filtracija samo onih fieldova i propertyja koji su dekorirani atributom *Save*.
3. Iteracija po svim fieldovima i propertyjima i poziv metode za spremanje.

3.3 Proces čitanja

Proces čitanja odvija se u sljedeća dva koraka.

1. Dohvat fieldova i propertyja klase pomoću funkcija *GetFields* i *GetProperties* iz refleksije (iz *System.Reflection* namespace-a).
2. Prolazak čitača XML datotekom i za svaki XML tag koji predstavlja field ili property poziv metode za čitanje.

3.4 Spremanje jednostavnih podataka

Kao što smo već ranije spomenuli, jednostavne podatke možemo spremiti unutar jednoga XML taga što je vidljivo na primjeru 3.1.

```
1 internal class Person : Saveable
2     {
3         [Save]
4         public string FirstName;
5         [Save]
```

```
6           public string LastName;
7           [Save]
8           public int Age { get; set; }
9           [Save]
10          public bool IsMarried { get; set; }
11          public Person(string firstName, string lastName, int age,
    bool isMarried)
12          {
13              FirstName = firstName;
14              LastName = lastName;
15              Age = age;
16              IsMarried = isMarried;
17          }
18      }
19  Person person = new Person("John", "Doe", 27, true);
20  string filePath = Path.Combine(projectRoot, "xml/person.xml");
21  person.WriteXML(filePath);
22  /*
23  <?xml version="1.0" encoding="utf-8"?>
24  <Person>
25    <FirstName>John</FirstName>
26    <LastName>Doe</LastName>
27    <Age>27</Age>
28    <IsMarried>true</IsMarried>
29  </Person>
30  */
```

Listing 3.1: Primjer spremanja objekta klase Person koja sadrži samo jednostavne podatke (dva fielda FirstName i LastName i dva propertyja Age i IsMarried)

3.5 Čitanje jednostavnih podataka

Kako su svi podaci zapisani u XML-u u obliku stringa, kod čitanja je najvažnije znati kako interpretirati pročitane podatke odnosno kako ih pretvoriti u pravi tip. Kod jednostavnih podataka to je vrlo lako jer točno znamo kojega tipa je field ili property kojega trenutno čitamo.

3.6 Spremanje složenih podataka

Složeni podaci mogu biti višerazinski (ugniježđeni) i zato je za njihovo spremanje ključan koncept rekurzije. Ove podatke spremamo tako da iteritamo po kolekciji i pozivamo rekurzivnu metodu za spremanje. To nam omogućuje da ako je podatak opet složen, opet će se pozvati metoda za spremanje na taj podatak i tako sve dok ne dođe do jednostavnoga podatka kojega direktno upisuje u XML. To je bazni slučaj ove rekurzije tj. kada dođe do jednostavnoga podatka rekurzija terminira.

3.6.1 Spremanje generičkih kolekcija

Primjeri spremanja generičkih kolekcija mogu se vidjeti na 3.2.

```
<!-- Primjer spremanja List<int> s tri elementa : [1, 2 ,3] -->
<ListOfInt>
    <Int32>1</Int32>
    <Int32>2</Int32>
    <Int32>3</Int32>
</ListOfInt>

<!-- Primjer spremanja Stack<int> s dva elementa : [4, 5] -->
<GenericStack>
    <Int32>4</Int32>
    <Int32>5</Int32>
</GenericStack>

<!-- Primjer spremanja Queue<int> s tri elementa : [6, 7, 8] -->
<GenericQueue>
    <Int32>6</Int32>
    <Int32>7</Int32>
  <Int32>8</Int32>
</GenericQueue>

<!-- Primjer spremanja HashSet<int> s tri elementa : [9, 10, 11] --
    >
<HashSet>
    <Int32>9</Int32>
    <Int32>10</Int32>
    <Int32>11</Int32>
</HashSet>

<!-- Primjer spremanja SortedSet<int> s tri elementa : [12, 13, 14]
    -->
<SortedSet>
    <Int32>12</Int32>
    <Int32>13</Int32>
    <Int32>14</Int32>
</HashSet>

<!-- Primjer spremanja Tuple<int, string> : (1, test) -->
<GenericTuple>
    <Int32>1</Int32>
    <String>test</String>
</GenericTuple>

<!-- Primjer spremanja Dictionary<int, string> s dva elementa : [ 1
    : test1, 2 : test2 ] -->
<GenericSortedList>
  <KeyValuePair>
    <Int32>1</Int32>
    <String>test1</String>
  </KeyValuePair>
  <KeyValuePair>
    <Int32>2</Int32>
    <String>test2</String>
  </KeyValuePair>
</GenericSortedList>

```

```
<!-- Primjer spremanja SortedList<int, string> s dva elementa : [ 3 : test3, 4 : test4 ] -->
<GenericSortedList>
  <KeyValuePair>
    <Int32>3</Int32>
    <String>test3</String>
  </KeyValuePair>
  <KeyValuePair>
    <Int32>4</Int32>
    <String>test4</String>
  </KeyValuePair>
</GenericSortedList>

<!-- Primjer spremanja viserazinskoga podatka List<List<int>> s dva elementa : [ [1, 2, 3], [4, 5, 6] ] -->
<Matrix>
    <List>
        <Int32>1</Int32>
        <Int32>2</Int32>
        <Int32>3</Int32>
    </List>
    <List>
        <Int32>4</Int32>
        <Int32>5</Int32>
        <Int32>6</Int32>
    </List>
  </Matrix>
```

Listing 3.2: Primjeri spremanja generičkih kolekcija

3.6.2 Spremanje negeneričkih kolekcija

Kao što smo već rekli, kod negeneričkih kolekcija elementi kolekcije ne moraju nužno biti istoga tipa tj. pri čitanju ne znamo kojega tipa je element koji čitamo. Zato je zajedno s vrijednosti za svaki element kolekcije bitno zapisati i tip elementa koji dohvatimo pomoću refleksije. Ovdje ćemo upotrijebiti XML atribute tako da ćemo za svaki tag u XML-u koji predstavlja element kolekcije još zapisati atribut *type* u koji ćemo pohraniti tip podatka. Primjeri spremanja negeneričkih kolekcija mogu se vidjeti na 3.3.

```
<!-- Primjer spremanja Arraya s tri elementa : [1, 2 ,3]. U Arrayju su svi elementi istoga tipa tako da mozemo u parent element zapisati tip. -->
<Array type="System.Int32">
    <Int32>1</Int32>
    <Int32>2</Int32>
    <Int32>3</Int32>
</Array>

<!-- Primjer spremanja Stacka s tri elementa : [test1, 1, 2] -->
```

```
<Queue>
    <String type="System.String">test1</String>
    <Int32 type="System.Int32">1</Int32>
  <Int32 type="System.Int32">2</Int32>
</Queue>

<!-- Primjer spremanja Queuea s tri elementa : [test3, 3, true] -->
<Queue>
    <String type="System.String">test3</String>
    <Int32 type="System.Int32">3</Int32>
    <Boolean type="System.Boolean">true</Boolean>
</Queue>

<!-- Primjer spremanja ArrayListe s tri elementa : [5, 6, test] -->
<ArrayList>
      <Int32 type="System.Int32">5</Int32>
      <Int32 type="System.Int32">6</Int32>
      <String type="System.String">test</String>
</ArrayList>

<!-- Primjer spremanja Hashtablea s dva elementa : [test11 : false,
    10 : test10] -->
<HashTable>
    <KeyValuePair>
        <String type="System.String">test11</String>
        <Boolean type="System.Boolean">false</Boolean>
    </KeyValuePair>
    <KeyValuePair>
        <Int32 type="System.Int32">10</Int32>
        <String type="System.String">test10</String>
    </KeyValuePair>
  </HashTable>

<!-- Primjer spremanja SortedListe s dva elementa : [7 : test7 , 8
   : 9] -->
<NonGenericSortedList>
    <KeyValuePair>
        <Int32 type="System.Int32">7</Int32>
        <String type="System.String">test7</String>
    </KeyValuePair>
    <KeyValuePair>
        <Int32 type="System.Int32">8</Int32>
        <Int32 type="System.Int32">9</Int32>
    </KeyValuePair>
</NonGenericSortedList>

<!-- Primjer spremanja viserazinskoga podatka ArrayList s tri
   elementa : [ true, ArrayList[1, 2, test2], Queue[1, test1] ] -->
<ArrayList>
    <Boolean type="System.Boolean">true</Boolean>
    <ArrayList type="System.Collections.ArrayList">
        <Int32 type="System.Int32">1</Int32>
        <Int32 type="System.Int32">2</Int32>
        <String type="System.String">test2</String>
    </ArrayList>
  <Queue type="System.Collections.Queue">
```

```
62          <Int32 type="System.Int32">1</Int32>
63          <String type="System.String">test1</String>
64      </Queue>
65  </ArrayList>
```

Listing 3.3: Primjeri spremanja negeneričkih kolekcija

3.7 Čitanje složenih podataka

Kao što smo koristili rekurziju za spremanje složenih podataka, tako ćemo ju koristiti i za čitanje. Podatke ćemo čitati tako da prvo pročitamo otvoreni tag. Zatim čitamo podatke i za svaki tag pozivamo rekurzivnu metodu za spremanje sve dok ne dođemo do zatvorenoga taga. Ako je čitač u međuvremenu naišao na složen podatak opet će se pozvati metoda za spremanje i taj princip će nam osigurati učinkovito spremanje višerazinskih podataka.

3.7.1 Čitanje negeneričkih kolekcija

Kako smo pri zapisivanju negeneričkih kolekcija za svaki podatak kao atribut upisali tip podatka, sada ćemo točno znati koji je tip podatka kojega trenutno čitamo i znat ćemo ga interpretirati.

4 | Optimizacija

Kako smo već ranije rekli da refleksija zna biti dosta spora, pokušat ćemo minimizirati pozive na refleksiju. Za mjerenje vremena izvođenja koda koristit ćemo *Stopwatch* iz namespace-a *System.Diagnostics* te ćemo usporediti vremena izvršavanja refleksije s našom implementacijom.

4.1 GetFields i GetProperties

Prvo što možemo napraviti je reducirati pozive metoda za dohvaćanje fieldova i propertyja. To ćemo napraviti tako da fieldove i propertyje dohvatimo samo pri instanciranju prvoga objekta te klase i cacheamo ih. Pri svakom spremanju/čitanju objekta te klase već ćemo imati spremljene fieldove i propertyje tako da će spremanje/čitanje biti dosta brže.

```
1 var watchFirst = Stopwatch.StartNew();
2 Artificial artificialFirst = new Artificial("John", "King", 28);
3 watchFirst.Stop();
4
5 var watchSecond = Stopwatch.StartNew();
6 Artificial artificialSecond = new Artificial("John", "King", 28);
7 watchSecond.Stop();
8
9 Console.WriteLine(watchFirst.Elapsed / watchSecond.Elapsed);
```

Listing 4.1: Usporedba brzine instanciranja prva dva objekta klase Artifical. Pri instanciranju prvoga objekta dohvaćaju se fieldovi i propertyji klase

Redni broj izvođenja	Rezultat
1	515
2	603
3	471
4	759
5	532

Tablica 4.1: Prikaz rezultata izvođenja koda 4.1 pet puta

4.2 GetValue i SetValue

GetValue i SetValue su funkcije iz refleksije za dohvat odnosno postavljanje vrijednosti fieldova i propertyja. Ove funkcije trebaju nam pri spremanju/čitanju svakoga fielda i propertyja svakoga objekta. GetValue i SetValue nisu dovoljno optimizirane i zato ćemo kreirati vlastite funkcije za ovu svrhu i zvat ćemo ih getteri i setteri. Opet ćemo iskoristiti istu tehniku tako da pri instanciranju prvoga objekta kreiramo gettere i settere za svaki field i property te ih cacheamo. Pri kreiranju gettera i settera koristit ćemo se konceptom compiled lambda izraza. Izrazi (expressions) su čvorovi u stablu izraza (expression tree), strukturi koja definira kod i omogućuje vizualizaciju izvornoga koda. C# omogućuje kreiranje izraza pa tako možemo kreirati vlastite gettere i settere kao lambda izraze. Da bismo mogli koristiti te funkcije još na izrazu moramo izvršiti Compile metodu kako bi se stablo izraza prevelo u CIL.

```
1 var watchLambda = Stopwatch.StartNew();
2 getter(artificial);
3 watchLambda.Stop();
4
5 var watchReflection = Stopwatch.StartNew();
6 field.GetValue(artificial);
7 watchReflection.Stop();
8
9 Console.WriteLine(watchReflection.Elapsed / watchLambda.Elapsed);
```

Listing 4.2: Usporedba brzine gettera kreiranoga pomoću lambda izraza i metode GetValue iz refleksije za field Matrix tipa List<List<int> > objekta klase Artificial

Redni broj izvođenja	Rezultat
1	14
2	14
3	13
4	12
5	14

Tablica 4.2: Prikaz rezultata izvođenja koda 4.2 pet puta

4.3 Usporedba sa C# serializerima

C# nudi ugrađeno rješenje za spremanje i čitanje podataka u XML formatu. Postoje dvije mogućnosti, a to su *XmlSerializer* i *DataContractSerializer*. *XmlSerializer* je dosta ograničen po pitanju tipova koje podržava, npr. ne podržava ako se u klasi nalazi Dictionary, Stack ili Queue. *DataContractSerializer* je po tom pitanju bolji, ali i nešto sporiji.

```
1 var watchOur = Stopwatch.StartNew();
2 person.WriteXML(filePath1);
3 watchOur.Stop();
4
5 var watchXmlSerializer = Stopwatch.StartNew();
6 xmlSerializer.Serialize(stream1, person);
7 watchXmlSerializer.Stop();
8
9 var watchDataContractSerializer = Stopwatch.StartNew();
10 dataContractSerializer.WriteObject(stream2, person);
11 watchDataContractSerializer.Stop();
12
13 Console.WriteLine(watchXmlSerializer.Elapsed / watchOur.Elapsed);
14 Console.WriteLine(watchDataContractSerializer.Elapsed / watchOur.
    Elapsed);
```

Listing 4.3: Usporedba brzine spremanja metode WriteXML iz klase Saveable, metode Serialize iz XmlSerializera i metode WriteObject iz DataContractSerializera na objektu klase Person

Redni broj izvođenja	Rezultat linije 13	Rezultat linije 14
1	5	39
2	6	46
3	5	33
4	6	43
5	4	33

Tablica 4.3: Prikaz rezultata izvođenja koda 4.3 pet puta

5 | Korištenje programa

Program se koristi tako da svaka klasa koja treba imati mogućnost spremanja mora nasljeđivati klasu Saveable. Zatim se fieldovi i propertyji koji se žele spremiti dekoriraju atributom Save. Atributu Save se također u konstruktor može proslijediti proizvoljno ime koje će field ili property imati u XML-u. Klasa Saveable sadrži dvije javne metode WriteXML i ReadXML. Obje metode imaju jedan parametar path u koji treba postaviti putanju do filea u koji će se pisati odnosno iz kojega će se čitati XML.

```
/*Primjer klase za spremanje: U klasi Complex nalaze se 2
    propertyja RealPart i ImaginaryPart.Zelimo spremiti oba pa ih
    dekoriramo atributom Save. Takoder umjesto ImaginaryPart u XML-u
     zelimo da pise Imaginary pa proslijedimo Imaginary u
    konstruktor atributa.*/
namespace ProgramStateSaver
{
    internal class Complex : Saveable
    {
        [Save]
        public double RealPart { get; set; } = 2;
        [Save("Imaginary")]
        public double ImaginaryPart { get; set; } = 3;
    }
}
```

Listing 5.1: Primjer klase za spremanje

```
/*Primjer kreiranja i spremanja objekta klase Complex u file
    complex.xml u folder xml u root folderu projekta: U string
    projectRoot dohvatimo putanju do root foldera projekta, kreiramo
     instancu klase Complex i u string filepath pomocu metode Path.
    Combine definiramo krajnju putanju do filea (u ovom slucaju file
     complex.xml spremamo u folder xml). Nakon toga na instanci
    pozovemo metodu WriteXML i proslijedimo filepath.*/
Complex complex = new Complex();
string projectRoot = Directory.GetParent(Directory.
    GetCurrentDirectory())!.Parent!.Parent!.FullName;
string filePath = Path.Combine(projectRoot, "xml/complex.xml");
complex.WriteXML(filePath);
```

Listing 5.2: Primjer spremanja objekta klase Complex

```xml
<!-- Izgled complex.xml filea u koji je spremljena instanca klase Complex s propertyjima RealPart vrijednosti 2 i ImaginaryPart vrijednosti 3-->
<?xml version="1.0" encoding="utf-8"?>
<Complex>
  <RealPart>2</RealPart>
  <Imaginary>3</Imaginary>
</Complex>
```

Listing 5.3: Izgled XML datoteke u koju je spremljen objekt klase Complex

```
/*Primjer rekonstrukcije objekta klase Complex spremljenoga u file complex.xml u folder xml u root folderu projekta: Kako sada u complex.xml fileu imamo spremljen objekt klase Complex, sada cemo napraviti rekonstrukciju. Kreiramo instancu klase Complex, postavimo RealPart i ImaginaryPart na 17 i ispisemo u konzolu kako bismo se uvjerili da su se vrijednosti promijenile. Kao i kod zapisivanja, opet cemo u string filepath postaviti putanju do filea (iz kojega cemo citati XML). Nakon toga na instanci pozovemo metodu ReadXML i ponovno ispisemo vrijednosti kako bismo se uvjerili da su se promijenile. */

string projectRoot = Directory.GetParent(Directory.GetCurrentDirectory())!.Parent!.Parent!.FullName;
Complex complex = new Complex();
complex.RealPart = complex.ImaginaryPart = 17;
Console.WriteLine($"Real part: {complex.RealPart} Imaginary part: {complex.ImaginaryPart}");
string filePath = Path.Combine(projectRoot, "xml/complex.xml");
complex.ReadXML(filePath);
Console.WriteLine($"Reconstructed Real part: {complex.RealPart} Reconstructed Imaginary part: {complex.ImaginaryPart}");
```

Listing 5.4: Primjer rekonstrukcije objekta klase Complex

```
Real part: 17 Imaginary part: 17
Reconstructed Real part: 2, Reconstructed Imaginary part: 3
```

Slika 5.1: Prikaz konzole nakon pokretanja koda 5.4

Literatura

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein , *Introduction to Algoritms, 4Ed*, MIT Press, 2001.

[2] M. Rahman , *C# Deconstructed: Discover how C# works on the .NET Framework*, 2014.

[3] A. Turtschi, J. Werry, G. Hack, J. Albahari , *C# .Net Web Developer's Guide*, 2002.

[4] Web izvor dostupan na `https://learn.microsoft.com/en-us/dotnet/csharp`

[5] Web izvor dostupan na `https://algodaily.com/categories/recursion`

[6] Web izvor dostupan na *"https://www.geeksforgeeks.org/introduction-to-recursion-data-structure-and-algorithm-tutorials"*

Sažetak

U ovom radu predstavljen je C# alat za spremanje stanja programa tj. spremanje i rekonstrukciju objekata proizvoljnih klasa u XML formatu. Pruženi su detaljan uvid, analiza i primjeri procesa spremanja i čitanja podataka kao i ključnih koncepata koji omogućuju učinkovitost, fleksibilnost i brzinu ovoga alata. Demonstrirane su i optimizacijske metode koje izuzetno pojačavaju performanse programa kao i usporedba performansi programa s alternativnim rješenjima. Detaljno je na primjeru objašnjeno kako se koristiti ovim alatom.

Ključne riječi

C#, program, stanje, XML, refleksija, spremanje, čitanje, rekurzija, optimizacija

Saving program state

Summary

This paper presents a C# tool for saving program state, i.e. saving and reconstructing objects of arbitrary classes in XML format. Detailed insight, analysis and examples of the process of saving and reading data as well as key concepts that ensure the efficiency, flexibility and speed of this tool are provided. Optimization methods that greatly enhance the performance of the program were also demonstrated, as well as a comparison of the program's performance with alternative solutions. The usage of this tool is explained in detail through a practical example.

Keywords

C#, program, state, XML, reflection, saving, reading, recursion, optimization