Demythisation of interfaces in OOP

Dragoş NICOLAU

National Institute for Research and Development in Informatics – ICI Bucharest

dragos.nicolau@ici.ro

Abstract: The current material aims at introducing several considerations designed to clarify the topic of interfaces - a remarkable software development tool, while at the same time a very challenging (burdensome) concept pertaining to Object Oriented Programming. Modern scientific literature includes a substantial number of written materials (tedious and purely descriptive in nature) dedicated exclusively to listing an inventory of the anatomical structure and functional tasks of interfaces, whereas only a negligible percentage of the currently available scientific works would dare to indicate correctly (though chaotically and inconsistently) the operating processes supported by interfaces. The contribution of the present paperwork is to identify and dismantle the most widely spread misconceptions about interfaces; to provide accurate and clear descriptions of the key notions and fundamental mechanisms lying at the core of interfaces activity; to equip the reader with the necessary skills as well as with the full motivation needed to incorporate interfaces into the code-development strategies.

Last but not least, even though this material primarily addresses software developers, we hope that it could also represent a beneficial eye-opener for all individuals willing to capture a good understanding of the seemingly "capricious and hard-to-read" persona of the saga of Object Oriented Programming.

Keywords: Interface, misconceptions, benefit of using, drawbacks of not using, OOP.

Demitizarea interfeţelor în Programarea Orientată Obiect

Rezumat: Lucrarea de faţă are drept scop prezentarea unui ansamblu de consideraţii gîndite (destinate) să clarifice noţiunea de "interfaţă" – un instrument remarcabil în dezvoltarea software, care, însă, riscă să rămînă, în mod oneros şi nedrept, un concept greu inteligibil în cadrul Programării Orientate Obiect. Literatura științifică modernă oferă un număr substanțial de materiale scrise (greoaie și pur descriptive, dedicate exclusiv listării banale a unui inventar de elemente anatomice și de sarcini funcționale ale interfețelor), din care însă doar un procent neglijabil, pînă în prezent, se încumetă să indice corect (deși haotic, incomplet și inconsecvent) modul real în care sînt utile interfețele. Contribuția prezentului material este aceea de a identifica și de a face cunoscute cele mai răspândite concepții greșite despre interfețe; aceea de a oferi descrieri precise și clare pentru noțiunile cheie și pentru mecanismele fundamentale care stau la baza principiului de lucru al interfețelor; cea mai importantă, aceea de a oferi cititorilor (dezvoltatorilor) motivația deplină necesară utilizării interfețelor în strategiile de dezvoltare a codului.

Nu în ultimul rând, chiar dacă acest material se adresează în principal dezvoltatorilor de software, sperăm să oferim un instrument de cunoaştere binevenit pentru oricine este interesat să înţeleagă acest personaj „capricios și greu de pătruns" din saga Programării Orientate Obiect.

Cuvinte cheie: interfaţă, concepţii greşite, avantaje ale utilizării, dezavantaje ale neglijării, POO.

1. Introduction

Main essential notions of Object Oriented Programming (referred to as OOP from this point onward) are straightforward and intuitive: classes are data structures that group functions (called methods), encapsulation is a restriction against excessive spreading of data and functions throughout a project (i.e. confining data to their own class as much as possible) and inheritance is the opportunity of taking for granted a certain amount of pieces of functionality.

It is our opinion that interfaces represent with certainty the single most bizarre and obscure concept of OOP. Despite the abundance of materials (and authors) that approach this topic, it still remains obvious the failure to provide the reader with the extremely needed rationales for the usage of interfaces. Consequently, instead of being regarded as useful tools for good programming practice, interfaces appear doomed to retain the ill-fated position of simply being basically "empty classes to be inherited". Unfortunately, within the scientific community of software developers, the vast majority

https://doi.org/10.33436/v30i4y202006

of authors (of printed books, articles, academic lectures etc) appear to focus solely on describing the operating principles of interfaces (exemplifying the notion of interfaces inside code written in Java or C#), while lamentably failing to promote the actual benefits of the consistent employing of interfaces.

1.1. Related works

For instance, most scientific works share the following traits:

(a) they merely present what interfaces are, while neither revealing the grounds behind building/using interfaces, nor any tangible advantages thereof [1];

(b) they elliptically and inappropriately present isolated, irrelevant cases in which interfaces seem to be of no utility; state that interfaces are useful for multiple inheritance; or present chaotically the role of interfaces [3];

(c) they present no rationale to employ interfaces: they simply assert how to implement interfaces, thus transforming the concept into nothing more than a theoretical, academic notion devoid of any practical utility [4];

(d) introduce interfaces as a new tool aiming to create a generic behavior rather than specific per class behavior; unfortunately, they lack clarity and neglect not only mentioning other (more) important/significant uses, but also asserting the functional identity of interfaces; they happen to insert erroneous and misleading scheme [6];

(e) they "advertise" interfaces as being a "peculiar, specific programming trick" in whose absence the detection of mouse or keyboard actions would become impossible - incidentally a very poor and irrelevant paradigm, as mouse and keyboard actions are truly captured by a certain window procedure, not by interfaces (which in this case function as simple action filters, a technicality imposed by the specific Java code compiler.) [2];

(f) they are generally a tiny bit inconclusive, incomplete & confusing [7].

1.2. Contribution of the present paperwork

These are a modest but **extremely representative** sample of a plethoric collection of papers exhibiting the protocol of using interfaces but practically never mentioning ***the reason, the raison d'être of resorting to this important programming feature***.

Every scholar emphasizes only the functional traits of interfaces either neglecting to underline their purpose or providing the audience with confusing and erroneous statements. Leaving unanswered the following questions wrongfully transforms the concept of interfaces yet into another abstruse academic complication: *why consuming time with understanding them ? what is their finality ?, what is their strong point, if any ?, to serve what technical purpose were they even invented ?,* in our opinion the crucial questions being *what is the benefit of using interfaces ?* and moreover so, *what is the havoc of not using interfaces ?*

As everyone can intuit, practically all the "answers" to these questions are not the most illuminating choice, the vast majority of them underlining – among other more or less convincing / confusing arguments – that interfaces are: *(a) a contract to be fulfilled by implementing some void methods, (b) a chance to enjoy multiple inheritance, (c) a mechanism for programming the detection of mouse / keyboard actions in a Desktop Application* or *(d) a board with sockets where one can plug different electrical appliances* etc.

The **contribution** of the present material is the striving to shad light on the most spread misconceptions, thus eliminating what we consider to be inaccuracies inside the general perspective on the concept, and the attempt to introduce the most important element in deciphering interfaces: incentive. **Although** this paperwork **mainly addresses** software developers, we think that it can be useful to any individual who wishes to deepen the understanding of a whimsical "character" of the saga
of OOP.

From here onward the paperwork is structured as follows:

- general misconceptions on interfaces;
- rationales for using interfaces, supported by tangible examples of OOP code; subservient to the purpose of highlighting the value of interfaces, some cases will illustrate *in antithesis* the outcome obtained by using interfaces *vs* the equivalent outcome attained without the contribution of interfaces;
- discussions;
- conclusions.

2. Main general misconceptions on interfaces

The intellectual frustration nourished by the author and generated by the abundance of misleading "answers" was the main motivation of writing this article. We shall start the journey into the domain of interfaces by trying to display some important misconceptions pertaining to interfaces.

(a) The first main misconception is that: An interface is a contract to be fulfilled by implementing some void methods.

We are of the opinion that interfaces are not a contract *per se*. We dare to disapprove. We think that this is a misconception recurrently imposed and easily accepted. Despite the fact that one can correctly assert that at anatomic level interfaces *behave* like a contract, at functional and "motivational" level interfaces are far from being a contract *per se*. In fact, this widespread myth is factually in a ridiculous contradiction with the semantics of the concept of "contract" offered by real life. Logically, why would anyone desperately seek to get involved in the bonds of contract? Why would anybody "sign" a contract instead of pursuing code development? Is there any perspective of gain in a contract based on compulsory supplementary implementations? Which is the exact goal to accomplish? Is there any profit or convenience in deploying the effort of implementing a list of void methods? What drives the desire to enter an absurd ring of obligations that are time and energy consuming ?

Understanding interfaces suffers even more grievously from the contradiction with the philosophy of OOP (*Object Oriented Programming*) itself. The single real point of OOP is obtaining and maximizing comfort (ease) in the process of developing and managing code. No other real reason has OOP to exist, but comfort. Hence, the sole purpose of entering a contract simply contradicts the idea of comfort. In conclusion, interfaces cannot possibly be the fulfilling of a contract, *per se*.

(b) The second main misconception is that: an interface is a chance to enjoy multiple inheritance.

Again, we dare to disapprove, as we are of the opinion that the purpose of interfaces is not at all multiple inheritance *per se*. Again, we dare to disapprove. Despite the fact that one can correctly assert that at anatomic level interfaces *can incarnate* multiple inheritance, at a purpose-related level interfaces are absolutely not connected with the idea of multiple inheritance.

This misconception comes in contradiction with the significance of the concept of "inheritance" in real life.

In real life, inheriting something is a benefit-bringing event; it is neither a troublesome circumstance, nor an effort committed to fulfilling some severe contract. Inheritance means getting for granted an asset (a profit), not a torment. (Exactly in this spirit, inheritance in OOP is enjoying a whole functionality without any effort.)

This misconception comes in contradiction with the significance of the concept of "multiple inheritance" in real life programming. In spite of the reverence due to C++ , one must admit that multiple inheritance is noxious. It is simply and straightforwardly against Nature.

For example, objects **billy** or **jimmy** are both instances of class.

Human, having the pedigree Human -> Mammal -> Vertebrate -> Animal.

The fact that they simply happen to be engineers, or parents, or tourists or whatever else – does not modify their congenital status. Moreover so, inheriting something that compels the beneficiary to additional work, is again in conflict with the idea of comfort (the main if not the single existential reason of OOP). Objectively speaking, every object *is a single entity* (it is something) and possibly *can do something else*, this being exactly the reason for which the Java compiler and the C# compiler were strictly designed to forbid multiple inheritance.

However, is implementing interface(s) not managing inheritance ? Obviously, yes, it is, but it is managing a stem of ***functional, special, technical, strange, unusual*** inheritance, instead of a stem of ***conceptual, classical, established*** inheritance. Interfaces are a *pseudo*-inheritance, an *artificial* inheritance. Interestingly, this is the intricate inheritance in which the assets are being provided by the inheritor instead of the parent class.

3. The concrete benefit of using interfaces

First and foremost, we consider opportune to underline that interfaces provide ***grouping facilities*** between otherwise ***totally unrelated*** classes.

Even if physically they are abstract classes, interfaces are called as such for reasons of elegance and rigor: abstract classes are ancestors, whereas interfaces are connectors. As it is well-known, if various objects are all of-a-kind and share a common general behavior, then it is *strongly recommended* that they descend from a common base class (abstract class), whereas if they are totally unrelated and must share a common functionality, then it is *obligatory* that they use (implement) an interface.

In the following section we shall present very specific examples of code (C#) capable of illustrating manifestly the role played by interfaces in the actual code development process.

1) When grouping some objects in collections. The objects are totally different, but have a common functionality

In this example we shall witness the first salutary effect of using interfaces. It will show plainly and clearly the difference between a sample of code that does not use interface, even if it should – and a sample of perfectly similar code (serving exactly the same functional purpose) that uses interfaces.

Without interfaces, grouping “genetically separated” (unrelated) objects is counter-intuitive, implying also many tries and type castings, whereas with interfaces the same grouping is intuitive and no more in need of tries and type castings.

Without Interfaces

```
// define the classes
class Avion { public void Print() { Console.WriteLine("Avion FLIES !"); } }
class Birdy { public void Print() { Console.WriteLine("Birdy FLIES !"); } }
class Hellyc { public void Print() { Console.WriteLine("Hellyc FLIES !"); } }
class Rocket { public void Print() { Console.WriteLine("Rocket FLIES !"); } }

// utilize the classes
object[] flota = new object[4]; // --- object[] is anti-intuitive
flota[0] = new Avion();
flota[1] = new Birdy();
flota[2] = new Hellyc();
flota[3] = new Rocket();

for (int i = 0; i <= flota.Length - 1; ++i)
{
       // --- many tries and castings
     if (flota[i] is Avion) ((Avion)flota[i]).Print();
```

```
     else if (flota[i] is Birdy) ((Birdy)flota[i]).Print();
     else if (flota[i] is Hellyc) ((Hellyc)flota[i]).Print();
     else if (flota[i] is Rocket) ((Rocket)flota[i]).Print();
}
```

With Interfaces

```
// define the classes
interface IFlies { void Print(); }
class Avion : IFlies { public void Print() {Console.WriteLine("Avion
FLIES !");} }
class Birdy : IFlies { public void Print() {Console.WriteLine("Birdy
FLIES !");} }
class Hellyc : IFlies { public void Print(){Console.WriteLine("Hellyc
FLIES !");}}
class Rocket : IFlies { public void Print(){Console.WriteLine("Rocket
FLIES !");}}

// utilize the classes
IFlies[] flota = new IFlies[4]; // *** IFlies[] is intuitive ***
flota[0] = new Avion();
flota[1] = new Birdy();
flota[2] = new Hellyc();
flota[3] = new Rocket ();

for (int i = 0; i <= flota.Length - 1; ++i)
{
     flota[i].Print(); // *** no more extra tries and castings ***
}
```

2) Passing an object to a function. The object type can be known only at run-time, but the object belongs to a family with a certain functionality

For example, let us assume that a Presentation, a Sheet, a Text or a Photo needs to be written to the Output Buffer of a Web Page; obviously, we can never know in advance which one of the four aforementioned imaginary features is requested by the remote client at a certain moment.

Without interfaces, passing at random an object coming from "genetically separated" (unrelated) objects is counter-intuitive, implying also many tries and type castings, whereas with interfaces passing also at random the same object is intuitive and no more in need of many tries and type castings.

Without Interfaces

```
// define the classes
class Prezentare { public void Write() { Console.WriteLine("Prezentare written to
OutputBuffer !"); } }
class Textul { public void Write() { Console.WriteLine("Textul written to
OutputBuffer !"); } }
class Sheet { public void Write() { Console.WriteLine("Sheet written to
OutputBuffer !"); } }
class Foto { public void Write() { Console.WriteLine("Foto written to
OutputBuffer !"); } }

class Pagina
{
     // Reference to the object to be written to the Buffer, in order to be sent
     // to the client.
     private object Cuerpo = null; // --- object is counter-intuitive

     public void OnLoad(string RequestFromClient)
     {
          if (RequestFromClient.ToUpper().IndexOf("PPT") != -1) Cuerpo =
               new Prezentare();
          else if (RequestFromClient.ToUpper().IndexOf("SHEET") != -1) Cuerpo =
               new Sheet();
          else if (RequestFromClient.ToUpper().IndexOf("TXT") != -1) Cuerpo =
```

```
                new Textul();
          else if (RequestFromClient.ToUpper().IndexOf("POZA") != -1) Cuerpo =
                new Foto();
          if (Cuerpo == null) return;
          RenderToBuffer(Cuerpo);
     }

     private void RenderToBuffer(object x)
     {
            // --- many tries and castings
            // --- If it is necessary to add more along the way,
            // let's say, Video or Music classes, one will have to
            // record them in the list of matches.
            // The code will result both ugly and prone to errors.
          if (x is Prezentare) ((Prezentare)x).Write();
          else if (x is Sheet) ((Sheet)x).Write();
          else if (x is Textul) ((Textul)x).Write();
          else if (x is Foto) ((Foto)x).Write();
     }
}
```

With Interfaces

```
interface IContinut { void Write(); }
class Prezentare : IContinut { public void Write() { Console.WriteLine("Prez.
written to OutputBuffer !");} }
class Textul : IContinut { public void Write() { Console.WriteLine("Textul
written to OutputBuffer !"); }}
class Sheet : IContinut { public void Write() { Console.WriteLine("Sheet written
to OutputBuffer !"); } }
class Foto : IContinut { public void Write() { Console.WriteLine("Foto written to
OutputBuffer !"); } }

class Pagina{ private IContinut Cuerpo = null; // *** IContinut is intuitive ***

     // ICuerpo is the refrence to the object to be written to the Buffer in
     // order to be sent to the client.
     public void OnLoad(string RequestFromClient)
     {
          if (RequestFromClient.ToUpper().IndexOf("PPT") != -1) Cuerpo =
                new Prezentare();
          else if (RequestFromClient.ToUpper().IndexOf("SHEET") != -1) Cuerpo =
                new Sheet();
          else if (RequestFromClient.ToUpper().IndexOf("TXT") != -1) Cuerpo =
                new Textul();
          else if (RequestFromClient.ToUpper().IndexOf("POZA") != -1) Cuerpo =
                new Foto();
          if (Cuerpo == null) return;
          RenderToBuffer(Cuerpo);
     }
     // *** no more extra tries and castings; elegant, convenient, safe code ***
     private void RenderToBuffer(IContinut x) { Cuerpo.Write(); }
}
```

3) When a method of class List < T > uses a helper method of class < T >

Let us suppose that we design a parameterized **`List <T>`** class that *hosts* all sorts of **`T`**-type elements. The method **`List<T>.Sort()`** will inevitably call a **`T`**-class comparison mechanism. *Without interfaces, using a helper method of a parameter class T by a method of a container class would be clearly impossible, as the creator of the* **`List <T>`** *class would have to inject in-here the comparison method that might be used by any potential downstream developer; this is a case in which using interfaces is vital, mandatory.*

Without Interfaces

In this example, using interfaces is mandatory. Now, here we assume that the general type **`T`** does not implement the **`Icomparable <T>`** interface. Method **`List<T>.Sort()`** would look

like this:

```
public void Sort() // example (1) without interfaces
{
      ...
      ...
      int rez = 0;

       // --- ABSURD! The creator of the List <T> class
       // would have to inject in-here the comparison method that
       // might be used by any potential downstream developer.

      if(T is Whatever) rez = elem[i].Against(elem[i + 1]);
      else if(T is NimporteQuoi) rez = elem[i].CoincideAvec(elem[i + 1]);
      else if(T is Qualcosa) rez = elem[i].DiversoDa(elem[i + 1]);
      else if(T is Ceva) rez = elem[i].ComparaCu(elem[i + 1]);
      ...
}
```

With Interfaces

General type **T** implements interface **IComparable <T>**.

```
// example (1) with interfaces
class Whatever : IComparable <Whatever>,
class NimporteQuoi : IComparable <NimporteQuoi>,
class Qualcosa : IComparable <Qualcosa>,
class Ceva : IComparable <Ceva>

public void Sort() // example (1) with interfaces: the new look of List<T>.Sort()
{
      ...
      ...
      int rez = elem[i].CompareTo(elem[i + 1]); // *** general, elegant, safe ***
      ...
}

// example (2) with interfaces
class Angajat : IComparable<Angajat>
{
      public int Salary { get; set; }
      public string Name { get; set; }

      // IComparable
      public int CompareTo(Angajat other)
      {
            if (Salary == other.Salary) return 0;
            if (Salary < other.Salary) return -1;
            if (Salary > other.Salary) return 1;
            return -2;
      }
}
// use:
List<Angajat> list = new List<Angajat>();
list.Add(new Angajat() { Name = "Steve", Salary = 10000 });
list.Add(new Angajat() { Name = "Janet", Salary = 10000 });
list.Add(new Angajat() { Name = "Andrew", Salary = 10000 });
list.Add(new Angajat() { Name = "Bill", Salary = 500000 });
list.Add(new Angajat() { Name = "Lucy", Salary = 8000 });
list.Sort(); // Uses IComparable.CompareTo()
```

4) When Domenico creates an upstream DLL and gives it to Ercole, who writes a downstream EXE. The DLL contains a non-derivable class, that Ercole will instantiate by the means of a factory and use only through already implemented interfaces

With interfaces, a versatile code results: regardless of the alterations that Domenico will make on his code in the future, the running of the code written by Ercole will remain unharmed. In addition, potential complications are basically removed for Ercole, since he does not have to know what classes will be operated "behind" his own code.

```
// *************************
// * Domenico creates the DLL
public interface IVolante { void Decola();}
public interface IBeveCarburante { void VroomVroom(); }

public class SalaDiProduzione // <== Factory Class Exported to be used Ercole.
{
     // <== exported enumeration to be used by Ercole.
     public enum Aeroplane { ARIETE, LEONE, SAGITTARIO, AERITALIAAIT320,
                             AERMACCHIAM3 };

     // <== Factory Static Function exported to be used by Ercole.
     public static IVolante CreaApparatoDiVolo(Aeroplane tip)
     { return new _Aereo(tip); }
}

// Accessible only inside this DLL, hence only by Domenico.
internal class _Aereo : IVolante, IBeveCarburante
{
     void IVolante.Decola() { Console.WriteLine("Sto decollando !"); }
     void IBeveCarburante.VroomVroom() { Console.WriteLine("Bevo Kerosen !"); }

     // Constructor
     public _Aereo(SalaDiProduzione.Aeroplane tipul)
     {
          // *** Regardless of the alterations that Domenico will make on this
          // code in the future, the running of the code written by Ercole
          // will remain intact (unharmed.
          //
          // *** Ercole does not have to know what classes will be
          // operated by his own code.
          Console.WriteLine("Aeroplano Italiano del tipo = {0}", tipul);
     }
}
//
// ************************************************************************
// * Ercole creates the EXE, after having imported the DLL given by Domenico.
//
// a) Uses the factory to create the airplane
IVolante v =
SalaDiProduzione.CreaApparatoDiVolo(SalaDiProduzione.Aeroplane.LEONE);

// b) Uses the appropriate interface to get the airplane perform the taking-off
v.Decola();

// c) Uses the appropriate interface to keep the airplane flying
IBeveCarburante turbo = v as IBeveCarburante; // Ercole must switch the interface
if (turbo != null) turbo.VroomVroom();
```

5) Code Testing

Interfaces help isolate bugs because they limit the scope of a possible logic error to a given subset of methods.

6) COM interoperability

Interfaces are crucial when importing a DLL developed in VB 6.0 into a project written in C++, and vice-versa. Interfaces are vital when developing a C#.Net DLL to be imported into a project written in VB 6.0 or C++, and vice-versa.

7) Effective Teamwork

Let us suppose that within a software company there be two teams, working on different components that must co-operate (must be coupled). If the two teams sit down on day ONE and agree on a set of interfaces, then starting from day TWO they can go their separate ways and implement their components around those interfaces. Team A can build test that harnesses and simulate the component coming from Team B, for testing, and vice versa. Consequently, the department gains parallel development and fewer bugs.

8) Elastic development

Let us suppose that a developer writes the following method that returns an array after having retrieved some data from a DataBase.

Without interfaces, this code might prove to be rigid: it returns only a `VectorList` type value. Changing at some moment the returned type would result in catastrophic consequences for the downstream code: a tiny code change in the back-end will require adjustments in countless different classes all across the World.

With interfaces, the same code becomes elastic (versatile): when it returns an interface, everything that is pointed to by that interface is easily replaceable. In addition, this case is acceptably illustrated by the metaphor stating that interfaces are "a board with sockets where one can plug different electrical appliances".

```
public VectorList getUsefulData(String query)
{
       //
       //
       // This code is rigid: it returns only a VectorList type value.
       // Changing the returned type would result in catastrophic consequences
       // for the downstream code: a tiny code change in the back-end will
       // require adjustments in many different classes all across the World.
       //
      VectorList rval = new VectorList();

       // magical code that gets the useful data
       // ...

       // ret
       return rval;
}
```

But this code might prove to be rigid. For instance, what if at some point the hypothetic developer finds that is mandatory to use (consequently to return), say, a **`LinkedList`**, or a **`DoublyLinkedList`** instead of the initial **`VectorList`**, as the new container happens to be more efficient for a particular purpose? For the downstream code, the consequences would be catastrophic: a tiny code change in the back-end will require adjustments in countless different classes all across the World. In order to prevent this kind of situation to occur, interfaces are a brilliant solution:

```
public IListA getUsefulData (String query)
{
       //
```

```
        // This code is elastic (versatile): it returns an interface, i.e.ILista,
so
        // everything that is pointed to by IListа is easily replaceable.
        // Obviously, class VectorList, class LinkedList and class
DoublyLinkedList
        // MUST implement interface IListа.
        //
      ILista rval = new LinkedList(); // VectorList(), DoublyLinkedList()... etc

        // magical code that gets the useful data
        // ...

        // ret
        return rval;
}
```

4. Discussions

Drawbacks of Using Interfaces

If one adds a new method to an interface, he/she must track down all implementations of that interface in the Universe and provide them with a concrete implementation of that method, whereas if one adds a new method to an abstract class, he/she has the option of providing a default implementation of it, so all existing code will continue to work without change. It is the dedicated standard rule that when creating an interface, one must be sure that everything be designed correctly and completely the first time. Once that interface is published and goes to the client, changes are extremely difficult to impose. Once the creator changes that interface, everyone else's code breaks.

It's possible to work around the issue presented above by creating a new interface that extends or supplements the original one, but the downstream developer will still have to change his concrete implementations in order to comply with the new interface, instead of - or in addition to - the previous (old) one.

Speed: Interfaces are slow, because they require extra indirection to find the corresponding method in the actual class. Modern JVMs (Java Virtual Machines) are discovering ways to reduce this speed penalty. Unlike interfaces, abstract classes are fast.

Interfaces are not generally used when throwing/catching exceptions (functional errors), in this case classes being the best instrument to incarnate specific exceptions [5].

5. Conclusions

In the vast majority (if not in the quasi-unanimity) of the paper-works indented to introduce (or comment on) elements of knowledge pertaining to Object Oriented Programming - interfaces, neglectfully and academically presented, sadly risk to remain a bizarre, tenebrous topic. Instead, this article aims to emphasize, by the means of concrete sample of what we think to be an elegant code, the benefactor role of interfaces.

The whole role of interfaces in quite simple: basically, they provide grouping facilities between otherwise totally unrelated classes.

Despite the fact that from anatomic point of view implementing interfaces represents operating with (multiple) class inheritance, from functional point of view implementing interfaces do not at all refer to inheritance *per se*: inheritance is *pure parenthood*, whereas interfaces *incarnate available parenthood* in the service of (intended to ensure) *artificial, ad-hoc brotherhood*.

Interfaces do refer to a *specifically functional, strange, unusual inheritance*, instead of a *conceptual, classical, established inheritance*. Interfaces *incarnate* a *pseudo-inheritance*, an *artificial inheritance*, a *sui generis inheritance* in which *assets are provided* by the *inheritor class* instead of

the parent class.

Interfaces do not materialize *filiation or parenthood*, but *brotherhood*. To summarize, interfaces are *available fatherhood* meant to serve *necessary, forced brotherhood*. The metaphor that interfaces are a contract does not offer the best solution to understanding interfaces. The rigorous formulation would be: "*interfaces may be regarded as a mutually profitable contract in that that in order to enjoy coupling totally unrelated objects (but sharing a certain functionality), the developer is compelled to implement every method of a given interface, in every inheritor class*".

Are interfaces a panacea ? Absolutely not. Using interfaces is simply a matter of choice. However, in some critical situations, using interfaces offers the chance to write a simple, elegant, easy to maintain, convenient and safe code / *versatile code*, whereas not using interfaces leads to a luxuriant, prone to errors code / *rigid code*, respectively.

Acknowledgement

This article is the result of the software developing activity that the author has performed within the Project "**RO-SmartAgeing; 2020 phases**".

REFERENCES

1. Antal, Margit (2020). *Object-Oriented Programming in Java*. Univ. Sapientia, Tg. Mureş, 2020, https://ms.sapientia.ro/~manyi/teaching/oop/oop_java.pdf.
2. Bersini, Hugues – *La programmation orientée objet - 4e édition: Cours et exercices en UML2*, Université Libre de Bruxelles, https://www.academia.edu/12016134/La_programmation_orient%C3%A9e_objet_4e_%C3%A9dition_Cours_et_exercices_en_UML2.
3. Frasinaru, Cristian – *Curs de programare în Java*. Univ. Alex. Ioan Cuza, Iaşi, https://ms.sapientia.ro/~manyi/teaching/oop/oop_java.pdf.
4. Germain, James – *Interfaces in Object Oriented Programming Languages*. University of Utah, https://www.cs.utah.edu/~germain/PPS/Topics/interfaces.html.
5. Ivan, Mihaela (2013). *Analiză comparativă a programării orientate pe obiecte în limbajele de programare - ABAP şi Java*. Revista Română de Informatică şi Automatică – RRIA (Romanian Journal for Information Technology and Automatic Control), https://rria.ici.ro/wp-content/uploads/2013/03/07-art.-Mihaela-Ivan.pdf.
6. Keogh, Jim & Mario Giannini, Mario (2004). *OOP Demystified*. McGraw Hill, 2004.
7. Spasojevic, Marinko – *C# Intermediate – Interfaces*, https://code-maze.com/csharp-interfaces/#explicitinterfaceimplementation.

Dragoş NICOLAU este absolvent al Facultăţii de Electrotehnică din cadrul Universităţii "Politehnica" Bucureşti, în anul 1991. În prezent este cercetător ştiinţific gradul III în cadrul Institutului Național de Cercetare-Dezvoltare în Informatică – ICI Bucureşti. Este specializat în dezvoltarea de aplicaţii Web, desktop şi reţea. Este foarte pasionat de zonele puţin explorate ale programării orientate pe obiect. Drept arii de interes, Dragoș Nicolau mai menționează: securizarea reţelelor, securizarea codurilor javascript şi fizica semiconductoarelor. Domnului Dragoș Nicolau îi place să studieze şi să implementeze aplicaţii software bazate pe fire de execuție, algoritmi de criptare/compresie, analiza de imagine şi comunicații reţea. A publicat peste 30 de articole în ţară şi străinătate. Între anii 1997-2002 a desfăşurat activitate didactică la Facultatea de Electrotehnică din Universitatea "Politehnica" din București. Dragoș Nicolau este vorbitor fluent de italiană, franceză şi engleză.

Dragoş NICOLAU has been graduated from The School of Electrical Engineering of Politehnica University of Bucharest, class of 1991. Currently he holds the position of scientific researcher –level III at National Institute for Research and Development in Informatics – ICI Bucharest. His area of expertise include developing web, desktop and network applications. He feels strongly passionate about the "uncharted territories" of Object-Oriented Programming. As his personal fields of occupational interests, Dragoș Nicolau would also mention: securing networks, securing javascript codes, and semiconductor physics. Mr. Nicolau is particularly kin on studying and deploying software based on execution threads, encryption / compression algorithms, image analysis, and network communications. He has published more than 30 articles with nationwide and foreign / international/ outreach. Between 1997-2002 he has been involved into the Academic teaching activity at the Faculty of Electrical Engineering of UPB. Dragoș Nicolau is fluent in Italian, French, and English.