Interface Inheritance

Introduction

This paper will present Interface Inheritance in the context of Object Oriented Programming. Advantages and disadvantages of using Interface Inheritance and last but not least how this concept can be made useful for code reuse.

In the first part terms that are used in this paper will be defined to remove any ambiguity on the used terminology. In the second part the concept will be presented in detail followed by the last part of this paper. In the third part the concept will be identified in specifications and programming languages.

It is presumed that the reader has at least basic knowledge regarding the Object Oriented Paradigm. The following section will not define every object oriented concept.

Definitions

A **class** is nothing more than a template that is used to create objects. A class specifies the attributes which will form the state of each instance and method implementations or instance behaviour (Wikipedia contributors, Class (computer programming) 2014).

An **object** is an instance of a class, any operation that an object can perform is made out of a name, input objects and a possible result object. The name, inputs and output of an operation is called the **signature**[1] of the operation. All operation signatures that an object exposes can be referred to as the **object's interface**[2] (Gamma, et al. 1994).

A **type** is a name used to denote a particular interface. One can say that an object is of multiple types (implements multiple interfaces) or that multiple distinct objects are of the same type (they all implement the same interface) (Gamma, et al. 1994).

The **implementation** of an object is given by its **class** which specifies the objects attributes and behaviour (Gamma, et al. 1994).

A type is a **subtype** of another type if the former contains all the operations specified by the later. This is also referred as **inheritance**, a type **inherits** another type if the former contains the later (Gamma, et al. 1994).

When a class inherits another class this means that the former contains both the attributes and method implementations of the later. Inheritance between classes is also referred as **implementation inheritance**. When a class contains an interface it is said that the class **implements** or **realizes** that interface. **Interface inheritance** relates to **interfaces** inheriting other **interfaces** (Gamma, et al. 1994).

[1] Some programming languages like Java or C# do not include the result object as part of the operation signature.
[2] The interface of an object does not imply that the respective object implements only one interface.

Interface inheritance and its benefits

Motivation

There was a time before the concept of interface as a type existed. The only way to define a type in Object Oriented Programming was through a class. Classes could be inherited allowing a form of code reuse however there were times when a class required to inherit more than one class in order to avoid code duplication. Inheriting multiple classes has proven that it can be problematic. Along side method implementations classes specify how instances are represented in memory. Figure 1 shows a simple class using UML notation.

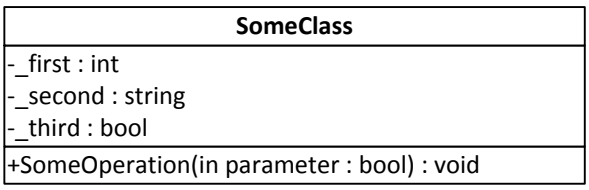

Figure 1: A simple class.

In this example *SomeClass* dictates that each instance will have exactly three attributes which will form the object representation in memory. In this case there are no problems with object representations.

However if multiple implementation inheritance is used then how instances are represented in memory can become a problem. Figure 2 shows the diamond problem which comes with multiple class inheritance (Wikipedia contributors, Multiple inheritance 2014).

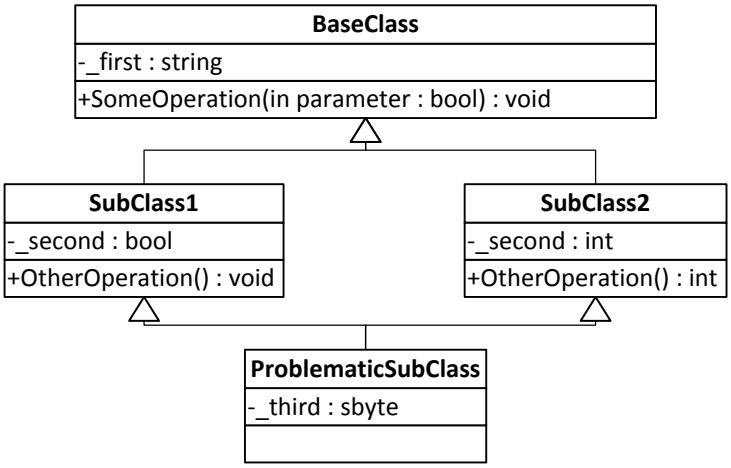

Figure 2: The diamond problem.

At first the problem may not be obvious because there is nothing explicit in the diagram (or code since in most, if not all, programming languages inherited attributes are not specified a 2nd time in the derived class) that suggests that there is any issue with this kind of inheritance.

Each derived class has all attributes and method implementations from its base class, this means that *SubClass1* and *SubClass2* both have an extra *_first* attribute not shown in their attribute lists. This is no big deal so far, the problem comes with *ProblematicSubClass* because this class inherits all attributes from *SubClass1*, alongside *_first* from *BaseClass*, as well as all attributes from *SubClass2*, and again *_first* is among these attributes. The issue is around *_first*, should this be a duplicate attribute? Should it be the same even though it comes from different inheritance paths? Should the developer be able to decide when inheriting?

Even though in some languages the developer can specify when inheriting multiple classes whether the same attributes coming from a common indirect base class[3] should or

[3] Indirect base classes are base classes of classes that are inherited. In figure 2 *ProblematicSubClass* has *BaseClass* as an indirect base class.

should not be duplicates it does not entirely resolve the problem. What if *BaseClass* would have not one but multiple attributes and in *ProblematicSubClass* only a part of them should be duplicates? This obviously breaks encapsulation through the entire inheritance graph because each subclass must know how its base classes are implemented and how their attributes are used within method implementations to be able to make a correct decisions.

Clearly multiple implementation inheritance can result in some complicated problems that would rather trade the benefits of object oriented code for its reuse. However if the base classes does not declare any attributes then there would be no problems with how objects are represented in memory, only the behavior is inherited which can be made dependent upon other methods (e.g.: factory method (Gamma, et al. 1994)). The only problem with this approach would be method signature collisions when two base classes define two methods that have the same signature or they differ through their result. In this case the subclass must decide which one is the implicit or default implementation. Figure 3 shows a restructured class diagram from figure 2.

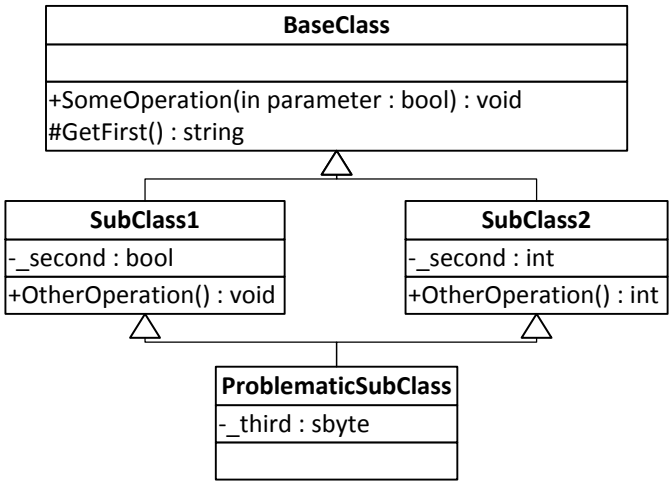

Figure 3: A diamond problem solution.

This can solve the diamond problem, however it is not feasible in real world applications where there are a lot more than a few classes. To apply this solution one must inspect each diamond-like inheritance because there are only required four classes to create the issue however instead of *SubClass1* or *SubClass2* there can be entire inheritance chains or other diamond-like inheritance graphs. Analyzing the particular case where multiple inheritance does not cause real issues one can notice that the lack of attributes saves the day. If one would restrict that if a class inherits multiple classes then indirect base classes must not have any attributes except for at most one inheritance path then there would be no diamond problem even though a class inherits multiple classes. Figure 4 shows a restructured diagram for figure 3 with a more clean approach.

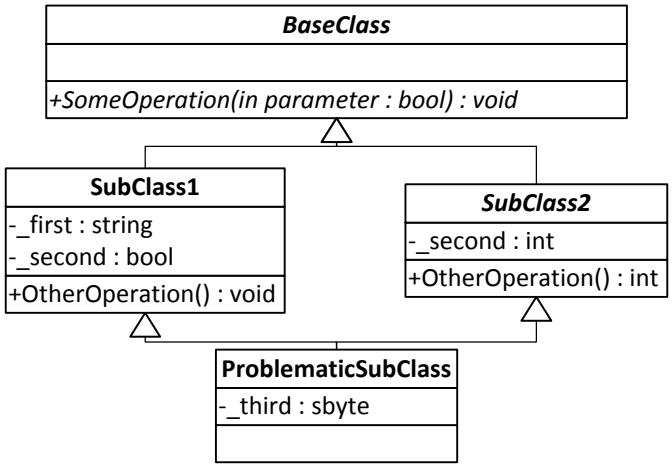

Figure 4: Diamond problem solution with abstract classes.

In this case SubClass1 defines the *_first* attribute in order to avoid the diamond

problem and both *BaseClass* and *SubClass2* are abstract because *SomeOperation* may require the *_first* attribute in order to be implemented. If so then the operation is abstract. *ProblematicSubClass* receives the implementation of *SomeOperation* through *SubClass1* and in the end has the same interface and may have the same behavior as in figure 1 without having memory representation issues. Notice that *SubClass2* also has an attribute and there is no representation problem because within *ProblematicSubClass* instances there would be two "sub instances", one of type *SubClass1* and the other of type *SubClass2* and each holds the attributes defined in their corresponding classes. There are no "common" attributes because *BaseClass* does not define any.

Through this solution the problem is easier to deal with however the one constant in software development tells that change is unavoidable (Freeman, et al. 2004). This means that base classes that once were attribute free may be required to have attributes in order to meet new or updated requirements. In order to avoid the problem to reappear the one changing the class definition must make sure that through the change the derived classes do not have more than one indirect base class with attributes, an impossible task in the case of libraries because client code that uses those libraries is not usually open source and even if it was it is titanic work to check every client code. The Open-Closed principle guides developers the write classes closed for change and open for extension however in production this is not always possible at least due to time pressure (Freeman, et al. 2004).

In conclusion any presented solution has one flaw, it makes the developer responsible for inheriting classes that do not cause the diamond problem. All those checks take time and is a job better suited for a computer rather than a human. Machines are great at iterating and checking concrete things such inheritance chains. However both presented solutions strive towards attribute-free indirect base classes. In other words base classes that only specify operations and maybe have some of them implemented without using any attributes. There is an implicit concept within the solution and should be made explicit like Eric Evans suggests in Domain Driven Design: "Make implicit concepts explicit" (E. Evans 2003)[4]. The implicit concept within all solutions is the concept of Interface as a type.

Interfaces define operations through signatures only, no implementation and no attributes. Interfaces are free from how implementing objects are represented in memory. They only specify what an object can do and offer no details of how they do it. Interfaces may hint the attributes of an object through getter and setter methods however the underlying class is free to define whether those values are held in memory through attributes or are calculated each time the value is requested.

This is the great advantage of interface inheritance over implementation inheritance. Because interfaces are restricted to not being able to define attributes it is guaranteed that multiple interface inheritance will never lead to the diamond problem. If a class can inherit at most one class and any number of interfaces and interfaces can inherit any number of other interfaces then it is guaranteed that the diamond problem will never happen[5]. Compilers and

[4] In this case the domain is software development through Object Oriented Programming.
[5] This happens in all CLI compliant languages (such as C#, Visual Basic .NET), Java and other languages.

interpreters can easily check these constraints and at the same time the code becomes more obvious and clear to any developer because interface types are explicitly stated and it is not required for the developer to infer from a class definition (which may change over time).

Figure 5 shows the solution in figure 4 restructured using interfaces.

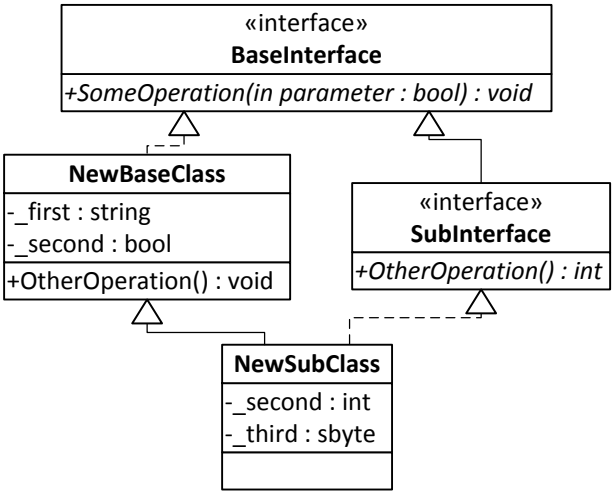

Figure 5: Diamond problem solution with interfaces.

One of the tradeoffs of using interfaces and implementing them instead of using implementation inheritance is that attribute declarations and method implementations are the responsibility of the implementing classes. This can become a source of duplicate code because when two classes implement multiple interfaces and for at least one of them they offer the exact same implementation, when change comes it must be made in two places instead of one.

A simple solution to this tradeoff is to move the common implementation into a 3rd class and the first two classes would aggregate an instance of the last. The code which would be duplicated in both classes will simply delegate to the aggregated instance. When change comes it will happen only in one place and it will affect both classes. Figure 6 shows how this can be done.

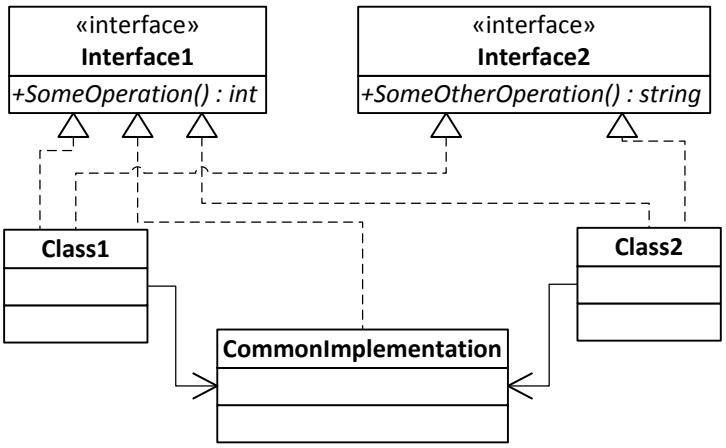

Figure 6: Common interface implementation.

The advantage of not causing the diamond problem in case of interface hierarchies is a great tool in the set of any developer. Interfaces are abstract and not bound by implementation details like classes. *"Program to an interface not an implementation"* (Gamma, et al. 1994) is one of the principles presented by Gang of Four and it is a very solid one. By programming to interfaces there is no dependency on the implementation. Instances are truly like black boxes because when consuming an interface the only concern is what that object can do and not how. This is great because it favors encapsulation and communication through messages (method calls).

Small or large interfaces?

Now knowing the benefits and drawbacks of using interfaces the next step is towards designing them. Would it be better if interfaces are small rather than large and why. SOLID object oriented design principles recommend through the Interface Segregation Principle that no client should be made to depend upon operations it does not use (Martin 2002).

Splitting large interfaces into smaller ones that contain closely related operations is usually a good guideline because at a moment in time a class will implement that interface and according to the Single Responsibility Principle the implementing class should have only one responsibility (Martin 2002).

Having a large interface may imply that the implementing class will have multiple responsibilities to carry out. To keep close to the principle that class will have to delegate some of its implementation to other classes where each will have one responsibility making the large one more of a "pass through" implementation. By analyzing the smaller classes each having a responsibility an interface can be extracted which will be part of the larger one. Either way to follow the Single Responsibility Principle a grouping of related operations will be made making the Interface Segregation implicit rather than explicit. Anyone who implements the large interface will have to go through the operation grouping process again and may find that some of the methods will be duplicates of another implementation.

Having small interfaces makes the design flexible. Instead of realizing a 2nd time the same large interface just to change the implementation of a few closely related operations, which leads to duplicate code, it is better to have a few extra but smaller interfaces. When a change is required only a part of the smaller interfaces will be implemented differently leaving the existing implementation to be reused. Figure 7 displays an interface segregation example.

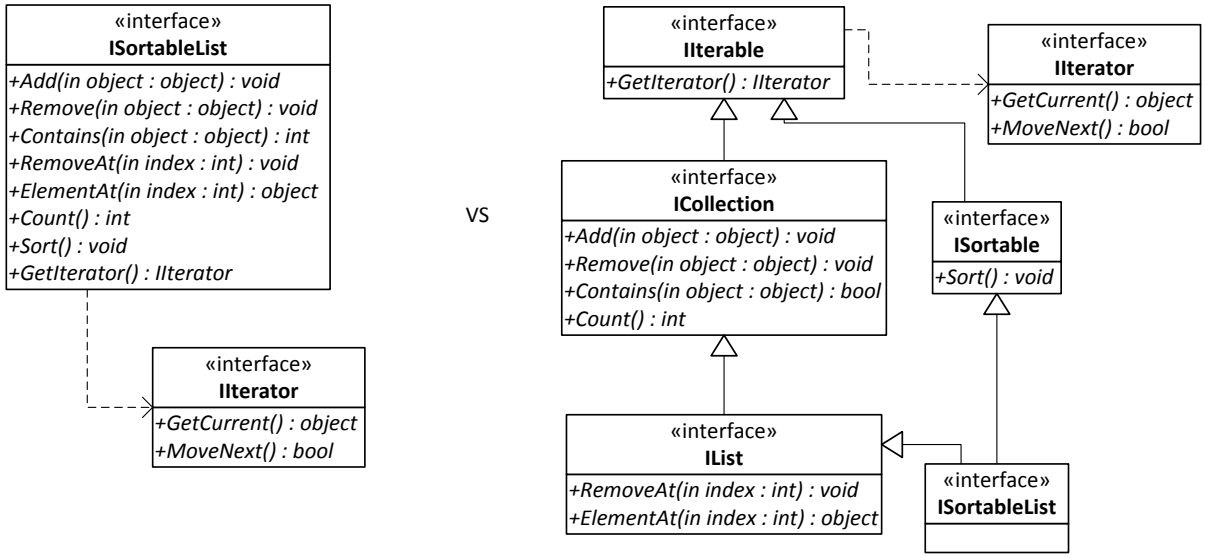

Figure 7: Segregating collections

On the left side there is a large interface representing a sorted list's interface. On the left side there is the same interface only that some of the operations which are list specific are declared by the *IList* interface and the sorting capability comes from the *ISortable* interface. The interface hierarchy on the right is more flexible and reusable than the one on

the left because each interface is specific to a type of collection. All collections are iterable and some operations may ask just for an iterable object because other collection specific operations are not of interest. If for each collection there would be an interface and each interface would declare a *GetIterator()* method then the method requiring an IIterable would be overloaded with each interface. Also when a new collection is defined a new overload should be added. Another approach to the problem would be to define an adapter to the *IIterable* (Gamma, et al. 1994) interface.

Another advantage for using IIterable rather than specific collection interfaces is that the client code does not have the chance to alter the collection.

If an *ISortableList* exists then undoubtedly there must an IList since the former is just a particular case of the later.

A list of observations about why the interfaces on the right are better than the ones on the left can go on for pages however that is not the scope of this paper.

Code reuse

Previously a form of code reuse was used in the case of implementation inheritance where a class extending another was reusing what was already implemented, however there are issues when a class inherits multiple classes to reuse code. There are other ways of reusing code not just by inheriting implementations.

Liskov's Substitution Principle defined as follows "Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S, where S is a subtype of T" says that any object of a base type (e.g.: T) can be substituted by an object of a subtype (e.g.: S) (Liskov and Wing 1994).

Besides implementation reuse through inheritance one can reuse code by writing methods that work with base type parameters meaning that any subtype can be used as actual parameters making the existing method reusable. .NET's Language Integrated Query is a great example of this mechanism. Almost all LINQ operators are defined on *IEnumerable<T>* where this interface is the base interface of any collection meaning that one can use LINQ operators on any collection including ones defined by the user that implement *IEnumerable<T>*. Besides this .NET also provides a way to implement all these operators by a custom query engine allowing custom interpretations for each operator. Entity Framework offers a custom implementation of each LINQ operator for translating a query into SQL syntax and then executing it on a relational database.

Besides base type parameters one can declare attributes of a base type allowing their values to be substituted by any subclass, the Strategy design pattern is an example of this reuse technique (Gamma, et al. 1994).

In conclusion interface inheritance is a less problematic way of specifying inheritance graphs compared to multiple class inheritance covering code reuse through object composition rather than implementation inheritance and also promoting dependency upon abstractions instead of concrete implementations.

Interface inheritance in the real world

Theory is theory and practice is something else. In this section the concept will be presented in the context of specifications: CORBA [6] and quite popular programming languages: Java and C#.

Interface inheritance in CORBA

Within this specification interfaces are abstract types which can contain definition of other types (act like namespaces), declare attributes which are actually getter and optionally setter operations, declare events, exceptions and of course operations.

When it comes to interface inheritance CORBA is very permissive. Besides allowing multiple inheritance between interfaces, which is expected, it allows redefinition of members that is allowing a derived interface to use a different name for an operation or attribute. Renaming of inherited members comes in handy when there are naming collisions and the extending interface needs to clarify which is which.

The example in figure 8 illustrates multiple interface inheritance using IDL[7].

```
interface Interface1 {
      typedef int Integer;
      Integer SomeOperation();
}
interface Interface2 {
      typedef long Integer;
      Integer SomeOtherOpration();
}
interface Interface3 : Interface1, Interface2 {
      void Operation(Interface1::Integer parameter);
}
```

Figure 8: multiple interface inheritance in CORBA.

In the above example Interface3 extends both Interface1 and Interface2 however both create the same alias for different numeric type. When Interface3 tries to use the type name Integer it must specify the alias from which base interface is being used.

Keep in mind that the language CORBA uses is only a specification and not an actual programming language. There are several mappings from this specification to programming languages such as C, C++ or Java.

[6] CORBA stands for Common Object Request Broker Architecture, a specification rather than a programming language, library or framewrok developed and maintained by the Object Management Group.
[7] IDL stands for Interface Definition Language, a standard used by CORBA to define interfaces.

Interface inheritance in Java

Java is a popular object oriented programming language that appeared in 1995 and developed by James Gosling at Sun Microsystems. At the moment Java is owned by Oracle since Sun was bought by them in 2009-2010.

Most of the language syntax is derived from C and C++ however it is not like any of the two. Like other languages, Java strives to abstract the underlying platform and go by the principle “Write once, run anywhere” which is quite true. A lot of today machines can run Java. The secret is that Java applications are compiled into standard Java Bytecode which can be interpreted by Java Virtual Machines and executed by the underlying machine.

To write Java compliant languages is easier than writing languages that compile directly to machine code like C or C++. In order to do this one must only write a compiler that generates equivalent Java Bytecode and then it can be executed by the JVM.

The Java programming language allows declaration of interfaces by using the interface keyword. An interface can contain only public operations and constants (final attributes). An interface can inherit any number of other interfaces. A class can inherit at most one class and implement any number of interfaces however in the Java world the signature of a method does not include the result type of the method. This has side effects upon interface realization because when two methods, each in one interface, differs only through return type and they are not compatible with one another, meaning that one is not a subtype of the other, then a class that is willing to implement both interfaces will generate a compile time error. Figure 9 illustrates this issue.

```
interface Interface1 {
      public void SomeMethod();
}
interface Interface2 {
      public Object SomeMethod();
}
class SomeClass implements Interface1, Interface2 {
      public Object SomeMethod(){
            return null;
      }
}
```

Figure 9: Interface realization issue in Java.

In this example both interfaces define a method with the same signature. The problem comes when trying to implement both interface in the same class. Because *void* is not compatible with *Object*, or any other type, this is a dead end scenario. The two interfaces can never be implemented in the same class. A solution to this problem is writing an adapter to one of the interfaces and have *SomeClass* implement the other. The adapter will take an instance that implements the other interface, preferably *Interface2* since it is easier to not return a result rather than magically generate one.

The adapter solution is not always this straight forward. The code in figure 10 illustrates a more complicated scenario.

```
interface Interface1 {
      public void Method1();
      public Object Method2();
}
interface Interface2 {
      public Object Method1();
      public void Method2();
}
class SomeClass implements Interface1, Interface2 {
      // …
}
```

Figure 10: Complicated interface realization in Java issue.

In this example the adapter solution will not work the same way. Each interface requires an adapter that delegate's calls to a class that implements both methods (can be *SomeClass*) that have a result. Figure 11 shows the solution for this issue.

```
interface Interface1 {
      public void Method1();
      public Object Method2();
}
interface Interface2 {
      public Object Method1();
      public void Method2();
}
class SomeClass {
      public Object Method1(){ /*...*/ }
      public Object Method2(){ /*...*/ }

      public Interface1 AsInterface1()
            { return new Interface1Adapter(); }
      Public Interface2 AsInterface2()
            { return new Interface2Adapter(); }
      //...
      private class Interface1Adapter implements Interface1 {
            public void Method1(){ SomeClass.this.Method1(); }

            public Object Method2(){ return SomeClass.this.Method2();}
      }

      private class Interface2Adapter implements Interface2 {
            public Object Method1(){ return SomeClass.this.Method1();}

            public void Method2(){SomeClass.this.Method2(); }
      }
}
```

Figure 11: Signature collision adapter solution.

In this solution *SomeClass* does not implement any of the interfaces because both methods need to return a result and instead exposes two extra methods that allows the current instance to be used as either interface.

Sadly not only *void* has this issue. Any conflicting signatures can cause this problem. The return types in the examples were *void* and *Object* however conflicting signature methods can have other result types including user defined types. Finding a workaround to the issue can be more complicated that simply omitting to return the result.

Java along other features has the possibility of creating anonymous types from interfaces. Instead of writing a class one can simply use the name of the interface and implement all its methods inline however there is no support for an anonymous type to implement multiple interfaces. This is not really a big issue because the method that creates the anonymous type is contained in a class and classes can have any number of nested private types. To make an anonymous type implement multiple interfaces one can define a nested private interface that extends all of the wanted interfaces, figure 12 illustrates the use of interface inheritance to achieve this goal.

```
class SomeClass {
	private interface MyInterface extends Interface1, Interface2 {}

	private MyInterface _myInterface = new MyInterface(){
		public String Method1(){
			return "1";
		}

		public Integer Method2(){
			return 1;
		}
	};

	public Interface1 Get1(){
		return _myInterface;
	}
	public Interface2 Get2(){
		return _myInterface;
	}
}
```

Figure 12: Implementing multiple interfaces in an anonymous type.

Interface inheritance in COM and WinRT

COM stands for Component Object Model and is a standard introduced by Microsoft in 1993 and it allows inter-process communication. The basic building block of COM are components which expose a public interface and have a private implementation. Any component must provide an interface that is separate from the implementing class thus hiding any implementation details from the user. COM is also a neutral language architecture, it allows programs written in different languages to bind together and form a final application. Because of its component oriented design one can write a component in C++ while another can write a different component in Visual Basic that consumes the former.

One of the draw backs of COM is that there is no implementation inheritance. In order to reuse the implementation of an existing class one must aggregate an instance of the existing class into the new one and delegate calls to it. Having the restriction of defining interfaces for each class can somewhat help in this matter. Multiple interface inheritance is possible allowing the reuse of existing interfaces to define new ones.

Interfaces in COM can declare operations and properties which are getter and setter operations. A class may implement any number of interfaces.

A new architecture based on COM and developed by Microsoft is the new WinRT made available since Window 8. This new programming model solves drawbacks that COM had such as complete object oriented support. Each WinRT component definition is stored in a Window Metadata file that exposes the types within that component that can be used and extended, not necessarily in the same programming language. The key of WinRT is the Window Metadata which is similar to .NET Assemblies which enables the use of Reflection to inspect the types contained by a WinRT Component (Wikipedia contributors, Windows Runtime 2014).

Unlike its predecessors, WinRT, is suited not only for desktop application but for ARM processors as well. One can write an application that can run on a mobile phone, on a server or on a desktop (Wikipedia contributors, Windows Runtime 2014).

Besides being object oriented there is also support for generics, a feature that COM lacks. Also being fully compliant with existing C++ and JavaScript code. A developer using native C++ libraries can continue to do so when using WinRT. The only restriction is that when using WinRT components the developer needs to use language extensions to interact with the platform (Wikipedia contributors, Windows Runtime 2014).

Interfaces in WinRT can declare properties which are getter and setter methods, operations, indexers and events. Interfaces can inherit any number of other interfaces while classes can implement any number of interfaces and inherit at most one class. Most of the capabilities of interface inheritance in WinRT are inspired from .NET which is described in the next section.

Interface inheritance in .NET Framework

.NET Framework is Microsoft's direct competitor to Oracle's Java. .NET Framework is a Common Language Infrastructure (CLI) implementation that allows multiple languages to interoperate by Common Intermediate Language (CIL). The working horse behind .NET Framework is the Common Language Runtime (CLR) which takes CIL and produces machine code that gets executed by the underlying platform. The CLR corresponds to the Java Virtual Machine (JVM) while the CIL corresponds to the Java Bytecode.

Each program written for .NET is compiled into an Assembly which can be a .DLL file or .EXE file. Unlike classic DLLs, Assemblies can be strongly named to avoid problems such as DLL Hell (Wikipedia contributors, DLL Hell 2014). Like the registry, .NET Framework has a Global Assembly Cache (GAC) where strongly named assemblies can be registered and referred directly from any .NET application without probing.

An Assembly also contains CIL which is obtained by compiling a CLI compliant language such as C# or Visual Basic .NET. CIL is a less user friend language that in some ways is similar to the Assembly Language, however it has high level features such as namespaces, modules, classes, interfaces and generics.

The following examples will be written in C# since this is a language that was conceived for the .NET Framework. C# is a multi paradigm language, primary object oriented.

C# allows definition of interfaces which can declare operations, properties, events and indexers. Interfaces can inherit multiple interfaces while classes can implement any number of interfaces and inherit at most one class. Structs can only implement interfaces, one cannot define hierarchies of structs.

Unlike Java, C# does not have result type inheritance covariance that is when a class extends another it cannot change the result type to one of its subtypes for an overridden method. To overcome this limitation C# allows shadowing of methods. One can shadow a method from a base class allowing the shadowing method to have a different result type or even access modifier. While shadowing is done by defining a new method, the old one is still accessible by using a reference of the base class type. Figure 13 shows an example of a class shadowing a method from its base class.

```
class Base {
      Public void Method(){
}
}
class Derived : Base {
      new public int Method(){
            return 0;
      }
}
```

Figure 13: Shadowing example.

This is a useful feature when it comes to interface inheritance because interface hierarchies may change the result type of methods declared in base interfaces to a more

specific one. This happens with the *IEnumerable<T>* interface where the *GetEnumerator()* from *IEnumerable* class is shadowed to return an *IEnumerator<T>* instead of *IEnuemrator* instance type. Besides being generic, the *IEnumeator<T>* interface extends *IEnumerator* and *IDisposable* interface which can be used in *using* statements.

Besides shadowing a great feature of C# is that it allows an implementing class to realize interfaces either implicit or explicit. With Java a class cannot implement two interfaces that each have a method with the same signature and incompatible return types. This is not the case of C#. When a class implements such interfaces it must decide which operation is the implicit one and which one is the explicit one. Figure 14 illustrates a class implementing two interfaces with conflicting signatures.

```
interface Interface1 {
      object Method();
}
interface Interface2 {
      void Method();
}
class SomeClass : Interface1, Interface2 {
      public object Metod(){
            return null;
      }
      void Interface2.Method(){
      }
}
```

Figure 14: Implicit and explicit implementations.

In this example *SomeClass* explicitly implements *Method()* from *Interface2* to avoid compilation errors. While it is a good idea to implicitly implement as much as possible it is not a requirement. Both interfaces can be explicitly implemented however this has the side effect of not making the interface operations visible using references of the implementing class type. Explicitly implemented operations are hidden from the class scope.

Interfaces can be generic. A generic interface can have any number of generic arguments and each can be covariant, contravariant (variant) or invariant.

Covariant generic arguments allow implicit conversions between generic type instances using a more generic type as a generic argument. Figure 15 shows an example on generic argument covariance.

Covariance refers to operation results. It is intuitive because if a generic parameter is used only as output type for declared operations then there is no problem in using a reference of a base type rather than the concrete type. This blends well with the substitution principle mentioned earlier. One can request an interface having a covariant generic argument of a base type and instead the client code will provide an instance with the generic argument of a subtype. It is perfectly valid because the subtype can substitute its base type.

```
interface CovariantInterface<out T> {
      T GetValue();
}
class SomeClass<T> : CovariantInterface<T> {
      public SomeClass(T value) { _value = value; }
      public T GetValue() { return _value; }
      private T _value;
}
class Program {
      static void Main(){
            CovariantInterface<string> someVariable
                  = new SomeClass<string>("Hey!");
            CovariantInterface<object> someOtherVariable
                  = someVariable;
            System.Console.WriteLine(someVariable.GetValue());
            // Will write: Hey!
      }
}
```

Figure 15: Covariance example.

The opposite of covariance is contravariance. Which is a little less intuitive than covariance because it refers to generic types that are used as method parameters, as input. Contravariant arguments allow a reference of a generic interface to use generic arguments of derived types of the one it was instantiated with. Figure 16 illustrates an example.

```
interface ContravariantInterface<in T> {
      void WriteValue(T value);
}
class SomeClass<T> : ContravariantInterface<T> {
      public void WriteValue(T value){
            System.Console.WriteLine(System.Convert.ToString(value));
      }
}
class Program {
      static void Main(){
            ContravariantInterface<object> someVariable
                  = new SomeClass<object>();
            ContravariantInterface<string> someOtherVariable
                  = someVariable;
            someOtherVariable.WriteValue("Hey!");
            // Will write: Hey!
      }
}
```

Figure 16: Contravariance example.

The interesting part about contravariance is that it allows generic arguments to be changed with more specific ones rather than more general ones. This is valid because in the end for any input parameter one can substitute a base type instance with a more specific type

instance. If the *WriteValue()* method takes an object as parameter then it is safe to provide a string which is a subtype of object.

While this feature is great, it works only with reference types (classes) and does not work with value types (structs and enums) even though value types derive from object, one cannot use object as a substitute generic argument for value type bound covariant arguments. This doesn't work the other way around either. One cannot substitute an object contravariant generic argument with a value type. This happens because behind the scenes when converting from value type to object or any reference type such as an interface that the value type implements, the CLR actually boxes the value from the *Stack* into a reference type. After boxing the reference type variable does not refer to the value on the *Stack* but a copy in the *HEAP*. This is why generic argument variance does not work with value types.

There are two solutions to this limitation. One would be to define a proxy that changes the generic argument from a TSource where TSource is a value type to a TDestination where TDestination is a reference type.

The other solution would be to use interface inheritance. Define one interface that works with objects and is not generic and define a 2nd interface with the same name but generic extending the 1st one. Shadow operations that change the result type and this way, regardless of the generic argument, any reference can be safely casted to the base interface which is not generic. Figure 17 illustrates this solution.

```
interface CovariantInterface { object GetValue(); }
interface CovariantInterface<out T> : CovariantInterface
      { new T GetValue(); }
class SomeClass<T> : CovariantInterface<T> {
      public SomeClass(T value) { _value = value; }
      public T GetValue() { return _value; }
      object CovariantInterface.GetValue() { return GetValue(); }
      private T _value;
}
class Program {
      static void Main(){
            CovariantInterface<int> someVariable
                  = new SomeClass<int>(1);
            CovariantInterface someOtherVariable = someVariable;
            System.Console.WriteLine(someOtherVariable.GetValue());
            // Will write: 1
      }
}
```

Figure 17: Overcoming variance limitations.

Generic interfaces are not limited to only one variant generic parameter. An interface can have any number of variant or invariant generic parameters. Also the .NET type system takes into account the number of generic arguments when resolving the type name. Generic arguments are part of the type name.

Bibliography

Albahari, Joseph, and Ben Albahari. *C# 5.0 in a Nutshell: The Definitive Reference.* 5th. O'Reilly, 2012.

Evans, Benjamin J., and David Flanagan. *Java in a Nutshell.* 6th. O'Reilly, 2014.

Evans, Eric. *Domain-driven Design: Tackling Complexity in the Heart of Software.* 1st. Addison-Wesley, 2003.

Freeman, Eric, Elisabeth Robson, Bert Bates, and Kathy Sierra. *Head First Design Patterns.* 1st. O'Reilly, 2004.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* 1st. Addison-Wesley, 1994.

Liskov, Barbara H., and Jeannette M. Wing. "A Behavioral Notion of Subtyping." *ACM Trans. Program. Lang. Syst.* (ACM), November 1994: 1811 - 1841.

Martin, Robert C. *Agile Software Development: Principles, Patterns and Practices.* 1st. Prentice Hall, 2002.

Miles, Russ, and Kim Hamilton. *Learning UML 2.0.* 1st. O'Reilly, 2006.

Object Management Group. "CORBA 3.3." *Object Management Group.* n.d. http://www.omg.org/spec/CORBA/3.3/.

Wikipedia contributors. *.NET Framework.* December 19, 2014. http://en.wikipedia.org/w/index.php?title=.NET_Framework&oldid=638720050.

—. *Class (computer programming).* November 12, 2014. http://en.wikipedia.org/w/index.php?title=Class_(computer_programming)&oldid=633467621.

—. *DLL Hell.* June 14, 2014. http://en.wikipedia.org/w/index.php?title=DLL_Hell&oldid=614150870.

—. *Multiple inheritance.* December 9, 2014. http://en.wikipedia.org/w/index.php?title=Multiple_inheritance&oldid=637302972.

—. *Windows Runtime.* November 29, 2014. http://en.wikipedia.org/w/index.php?title=Windows_Runtime&oldid=635938427.