# Implied Interface

**Authors :** Alex Shindich, Curt Hagenlocher

## Intent

Establish an interface abstraction implicitly by using functional polymorphism in the cases where the use of explicit interface constructs is either unavailable or would cause excessive refactoring of object-oriented code.

## Forces

- The most successful libraries of generic algorithms impose minimal requirements on their subjects. All such requirements must be explicitly defined in order for the library to be widely used.
- The explicit interfaces provide a structured way of defining the exact parameters of the collaboration between the algorithm and its subjects. In contrast, the implied interfaces are defined only by algorithm's implementation.
- There is a need to minimize the time it takes to develop generic algorithms.
- Changes to the explicit interfaces result in a big refactoring effort to the client code that is solely aimed at making the client code compliant with the new look of the old interfaces.
- Some OO programming languages and legacy systems do not support the notion of explicit interfaces, but use the functional polymorphism to define an interface abstraction.

## Motivation

There are many ways of implementing an interface abstraction. Most commonly known implementations make use of language or technology features to explicitly express interfaces. A well-known example of such a language is Java, and a good example of such a technology is COM. The languages that support functional polymorphism offer a very powerful alternative for implementing interface abstraction. Among such languages are Lisp, Perl, Python, JavaScript, and C++ (templates).

Most programmers make use of the Implied Interface pattern without even realizing it. In fact, one could argue that when applied to a particular language, the pattern turns into a language-specific idiom. The inherent simplicity of this pattern causes an unfortunate side effect – implied interfaces are rarely acknowledged as such and therefore are almost never documented.

The purpose of this paper is to promote the notion of the Implied Interface to a first-class pattern. It is the authors' hope that the implied interfaces will be better documented once the existence of the pattern is widely acknowledged.

Suppose that we are interested in developing an algorithm or library of algorithms that would operate unchanged on a vast majority of existing code, and thus appeal to a large group of potential users.

In decreasing order of generality, such an algorithm might operate on

1. Any object
2. Any object that implements a specific, explicit interface
3. Any object derived from a specific abstract base class
4. An object of a specified concrete class

The corresponding disadvantages of these targets are

1. The algorithm is limited to simple container classes. Type safety may be lost.
2. Existing code must be modified to support the algorithm by adding the interface to each class that wishes to use the algorithm, or by using the interface or class adapters
3. Existing code must be modified to support the algorithm as in case 2. In languages such as Java that do not support multiple inheritance, extensive refactoring may be required.
4. The algorithm is not generic.

Let's analyze the solution with an explicit algorithm interface and a class adapter.

## Class Adapter Solution

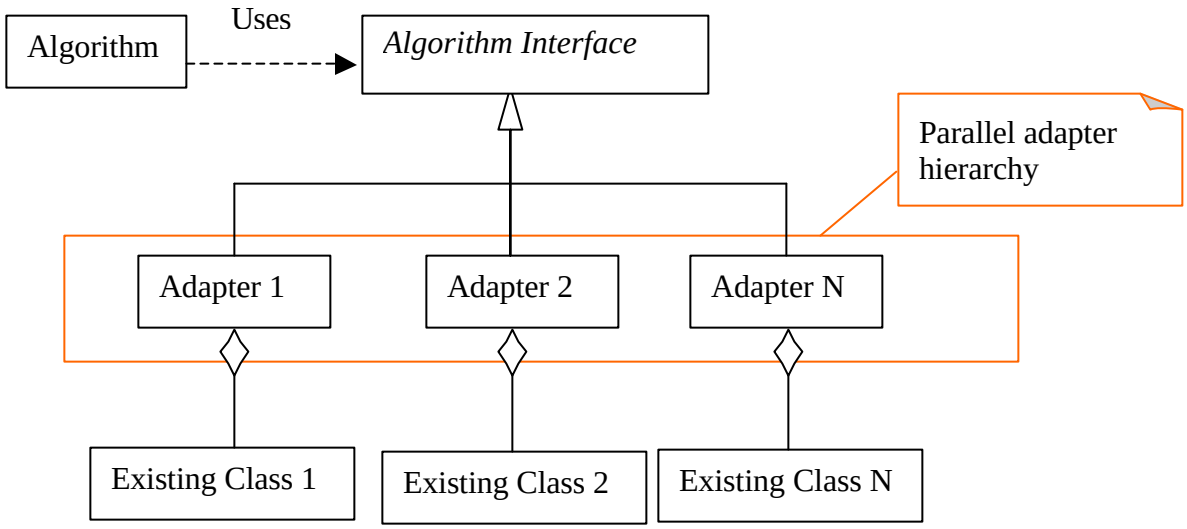


**Fig 1.a**

As we can see from the class diagram (Fig 1.a), the use of class adapter will lead to creation of a parallel hierarchy of adapters. This solution does not work with

the forces that disallow extensive modifications to existing class hierarchies, and demand to minimize the development time.

An alternative way of implementing the Algorithm Interface is to use an Interface Adapter (Fig 1.b). Unfortunately this would require that existing classes had a common base interface, which would preclude a large number if existing/legacy systems from being able to use our algorithm implementation.

**Interface Adapter Solution**

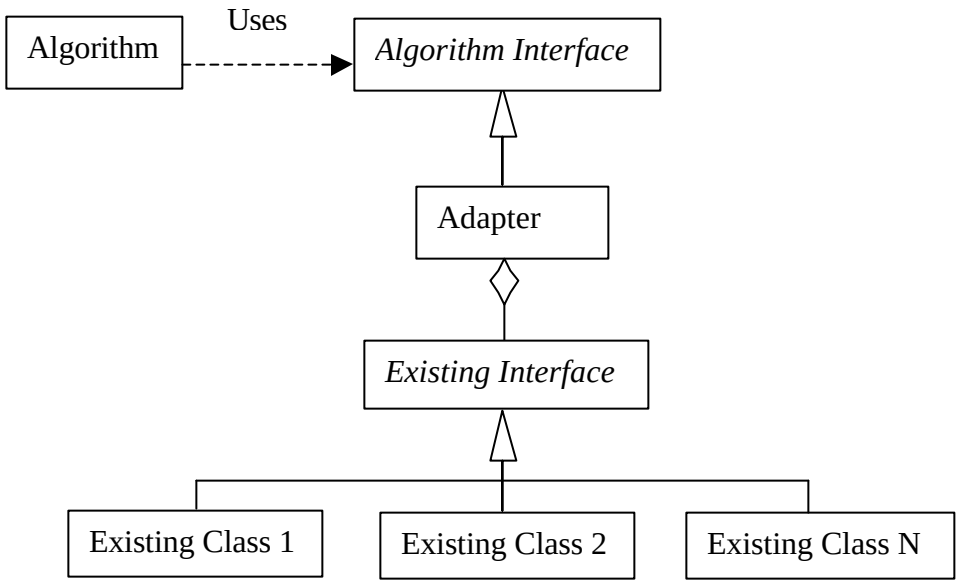


**Fig 1.b**

Around 1990, the state of the art in C++ class libraries was the NIH Class Library[1]. This included a number of useful containers, each of which required that the objects being contained be a subclass of a fairly wide `Object` class. This is an example of category 3, above. Despite the high quality and usefulness of the code, it did not see widespread usage at least in part because of its incompatibility with existing and new code that did not derive from the same `Object` class.

We can achieve greater success by recognizing that there is a fifth possibility that fits between the first two in its level of generality. We do this by taking advantage of *functional polymorphism*. *Functional polymorphism* is a construct that allows writing generic algorithms that can operate on **any** data type that implements methods and properties used in the algorithm. Functionally polymorphic methods and properties are bound by name. Depending on the language, the binding is performed either at compile time or at runtime.

[1] Keith E. Gorlen, Sanford M. Orlow and Perry S. Plexico, *Data Abstraction and Object-Oriented Programming in C++*, John Wiley & Sons, Ltd., Sussex, England, 1990.

By 1996, the state of the art in C++ libraries was the Standard Template Library – STL. STL, now in widespread use, takes advantage of the template features of C++ to implement generic algorithms. An STL class or algorithm operates on another class T. In doing so, it defines an *implied interface* for the target class. In the simplest cases, this implied interface consists only of publicly accessible constructor, copy constructor and destructor. Changes to the target class are required only in very rare cases; in most of these, a generic adapter class can be used.

Because we want widespread use of our code, we will learn from this history, and use functional polymorphisms with implied interfaces to implement the algorithms. In order to improve our chances of success, we will document all the requirements that our library imposes on its clients. In particular, we will define all the implicit interfaces that it uses.

As an additional benefit, the future cost of changing the implicit interfaces that our library uses will be insignificant in comparison to the refactoring costs we would incur had we used explicit interfaces. Consider the following example:

a. A company is trying to define a vendor-independent native COM interface to middleware servers.
b. The authors of the interface considered snapshotting as one of the basic features that any middleware vendor would support. And indeed, the first vendor of choice supports the snapshot capability.
c. At a later point, the company decides to add support for another middleware product, but the vendor of the new product doesn't support the snapshot feature.
d. Since the COM object for the new middleware product will not support snapshotting, the authors decide to split the existing interface into two – the base interface that doesn't support snapshotting and the snapshot interface. An alternative solution would require implementing the snapshot method that doesn't do anything, but that would violate Liskov's rule.
e. The code that doesn't make use of the snapshot feature now needs to be refactored to use the base interface.

As it can be seen from the above example, changes to the explicit interfaces result in a huge refactoring effort aimed exclusively at satisfying the new look of the old interfaces. If the authors chose implied interface, such as OLE's IDispatch, over the explicit COM interfaces, the refactoring of the client code would be limited to the areas that make use of the snapshotting function.

# Applicability

Use the Implied Interface pattern when:

- The language supports the notion of functional polymorphism
- The language does not support the notion of explicit interfaces

- The use of explicit interfaces results in an excessive refactoring effort
- There is a need to develop generic algorithms that can operate on a wide range of existing data types

Do not use this pattern if checking for compliance with an explicit interface is important.

# Structure

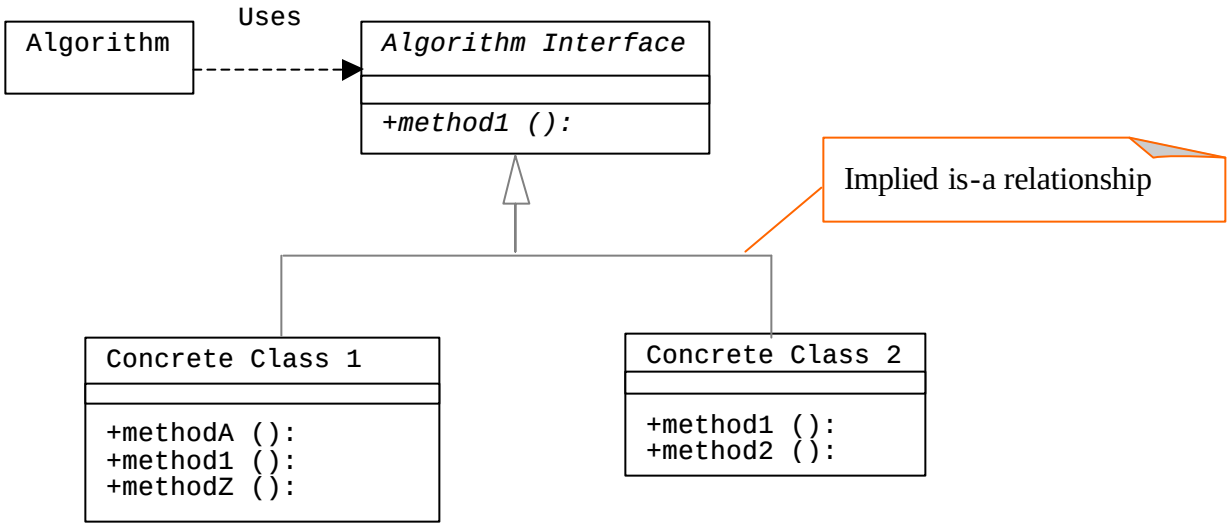


**Fig 2**

The generalization arrows are grayed out because they are implied. In reality, the concrete classes do not derive from a common interface.

# Participants

- **ImpliedInterface**
  - Defines an interface abstraction
  - Creates a contract. The contract says that the object will implement the functions of the implied interface.
- **Concrete classes**
  - Implement methods defined in the implied interface. Not all of the methods have to be implemented. Clients will still be able to use implemented methods.
  - If an object violates the interface contract, the client is well within its rights to throw whatever exception it wants or to flag an error during the compilation.
- **Algorithm**
  - Makes use of the objects that implement methods of the implied interface

# Collaborations

- Clients invoke implied interface methods on the instances of concrete classes using method binding by name. No Java-like typecasting to the interface reference is needed; thus, the implied interface itself need not be completely implemented.

# Consequences

The Implied Interface pattern has several benefits and drawbacks:

1. Introduces an interface abstraction that is useful for design purposes
2. Allows polymorphic access to the objects that implement methods of the implied interface but are otherwise totally unrelated
3. Makes it easy for objects to support multiple interfaces by simply implementing the methods of such interfaces
4. Reduces the clutter of the derivation hierarchy
5. Allows a class to implement the interface only partially
6. The use of functional polymorphism minimizes the amount of unnecessary refactoring associated with the interface changes
7. It is not always possible to safely check interface compliance
8. Depending on the language/technology, the implementation of this pattern may lead to increased executable image size (C++ templates) or inflict performance penalties (if functional polymorphism is implemented via an extra level of indirection)

# Implementation

In the languages that have interface support, it is customary to check if an object implements a particular interface. This check can happen at compile-time (for strongly typed languages) or at run-time. This check is not typical when using an implied interface. While many languages allow discovering whether or not a specific method is present (usually through introspection), the pattern is to assume that it does exist, and to trap the error to handle the case where it does not.

The implementation strategy is extremely simple.

1. *Defining an interface*. There is no formal syntax for implied interface definition. Using your favorite documentation method, document the methods that make up the interface. For example:

```
<?xml version="1.0" ?>
<Interface  name="ILivingCreature"
            doc="Defines an interface to a living
          creature">
  <Method   name="eat"
            retvaltype="void"
            doc="Consumes food to produce the energy for
            the living creature"/>
  <Method   name="makeNoise"
            retvaltype="void"
```

```
            doc="Makes creature-specific noise. "/>
 </Interface>
```

2. *Modifying concrete classes to conform to the interface*. Simply implement the methods documented in step 1. It is not necessary to implement all the interface methods but only the implemented methods will be available to the client code.

   Python example:

```
>>> class Dog:
      """"""

      def eat (self):
            """eat () -> void
            Eats bones.
            """

            print "Bones are yummy!"
```

   Class Dog partially implements ILivingCreature interface because it only implements eat method.

   An equivalent definition in C++ would look like:

```
class Dog
{
public:
      void eat () const
      {
            std::cout << "Bones are yummy!" << std::endl;
      }
};
```

   Note that method eat is not virtual.

3. *Making use of the interface*. The client code simply calls the methods on the instances of the concrete classes. A language-specific error occurs if a concrete class does not implement a particular method.

   Python example:

```
>>> d = Dog ()
>>> d.eat ()
Bones are yummy!
>>> d.makeNoise ()
Traceback (most recent call last):
  File "<pyshell#56>", line 1, in ?
    d.bark ()
AttributeError: Dog instance has no attribute 'makeNoise'
>>>
```

   **Note**: Python uses introspection to bind methods and properties at runtime. This makes all Python algorithms generic. In C++ generic algorithms are written using templates.

   C++ example:

```
template <class T> void feedCreature (T & const creature)
{
      creature.eat ();
}

feedCreature (Dog());
```

Output:

```
Bones are yummy!
```

Trying to compile the following code would result in compile error:

```
template <class T> void hearCreature(const T & creature)
{
      creature.makeNoise();
}

hearCreature(Dog());

--------------------Configuration: Dog - Win32 Debug-------------
Compiling...
Dog.cpp
D:\Dog\Dog.cpp(16) : error C2039: 'makeNoise' : is not a member
of 'Dog'
        x.cpp(4) : see declaration of 'Dog'
        x.cpp(24) : see reference to function template
instantiation 'void __cdecl hearCreature(const class Dog &)'
being
compiled
Error executing cl.exe.

dog.exe - 1 error(s), 0 warning(s)
```

# Sample Code

The sample code is going to illustrate the solution to the problem presented in the motivation section.

The goal is to develop a library of generic mathematical algorithms. The library should have an algorithm for computing a sum of elements of an arbitrary sequence. The algorithm should work with a vast majority of existing types, and impose minimal requirements on the algorithm's subjects.

The first step is to define an implied interface that sum algorithm will operate on.

```
<?xml version="1.0" ?>
<Interface  name="IAddable"
            doc="Defines an interface for adding two
          objects of the same type">

  <Method   name="operator +"
            retvaltype="unknown type"
            doc="Adds two objects of the same type">

      <Parameter  name="that"
                  type=" unknown type"
```

```
                        inout="in"
                        doc="The value to be added to the
                  value contained in this instance."/>
          </Method>

        </Interface>
```

The next step is to write the algorithm.

```
>>> def sum (sequence):
      if not len (sequence):
            raise "Non-empty sequence expected."
      return reduce (lambda a, b: a+b, sequence)
```

As you can see, the implementation in Python is straightforward, because the language has built-in support for generic sequences.

The C++ implementation of our algorithm will rely on STL's iterators.

```
template <class Iterator, class T>
void sum (Iterator begin, Iterator end, T & out)
{
      if (begin == end)
      {
            throw exception ("Non-empty container expected.");
      }
      out = *begin;
      for (Iterator iter = ++begin; iter != end; ++iter)
      {
            out = out + *iter;
      }
}
```

The only thing left at this point is to make use of the algorithm.

```
>>> nums = (1,2,3,4,5,6,7,8,9)
>>> sum (nums)
45
>>> strs = ['1','2','3','4','5','6','7','8','9']
>>> sum (strs)
'123456789'
```

An equivalent example in C++ would look like

```
      std::vector<int> nums;
      for (int i = 1; i < 10; ++i)
      {
            nums.push_back (i);
      }
      int result = 0;
      sum (nums.begin (), nums.end (), result);
      std::cout << result << std::endl;

      std::set<std::string> strs;
      for (i = 1; i < 10; ++i)
      {
            std::stringstream stream;
            stream << i;
            strs.insert (stream.str ());
      }
      std::string strresult;
      sum (strs.begin (), strs.end (), strresult);
      std::cout << strresult.c_str () << std::endl;
```

```
Output:
45
123456789
```

As the examples above demonstrated, the sum algorithm was used successfully with a set of existing types that had an implementation of "+" operator.

# Known Uses

The Implied Interface pattern allows polymorphic treatment of objects that are not members of the same class hierarchy; two totally unrelated objects could implement methods with the same names (the exact restrictions vary depending on the implementation language; for instance C++ also requires that method signatures match.) to indicate that they implement the same implied interface. STL is a perfect example of a library that makes use of the implied interface. This technique is also very common in Python.

This pattern is also used in conjunction with Windows DLLs. Each DLL can export a number of functions that are not grouped. DLL's users may choose to only use a subset of the exported functions. More importantly, the client may use two DLLs polymorphically as long as they both export a function with the same name and signature. (In practice, there are more restrictions then simply exporting the same function name from the DLL. Implementations of both functions must use the same calling convention, the same name-mangling scheme, the same version of C++ runtime library, etc. There are other restrictions that are not mentioned here.)

There is an example of Implicit Interface usage that came about during the early days of COM. The (original) COM control spec defined a certain number of "standard" properties to be accessed through automation. These properties were given standard DISPIDs. For instance, DISPID_FONT is -512. This made it possible to set the font without going through the vtbl by using the IDispatch interface with a DISPID of -512.

# Related Patterns

Many patterns that rely on the notion of interfaces can be implemented using the Implied Interface pattern.
Also see The Abstract Class Pattern.

# References


- The Abstract Class Pattern, Bobby Wolf (http://jerry.cs.uiuc.edu/~plop/plop97/Proceedings/woolf.pdf)
- Design Patterns, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

# Acknowledgments

We would like to thank Jim Stern and John Liebenau for their comments on the pattern. Special thanks go to our PloP shepherd Hans Wegener for his help during the revision of this paper.