Multiple Inheritance Using Interfaces

Stephen McHenry

A Discussion of the Differences Between Implementation and Interface Inheritance and an Explanation of How To Use Multiple Interface Inheritance

This document describes the differences of implementation inheritance and interface inheritance, and shows how to use multiple interface inheritance. It also provides an example of how to simulate multiple implementation inheritance using multiple interface inheritance and delegation.

Introducing Inheritance

Inheritance is one of the fundamental tenets of the object paradigm. Virtually all experts agree that, unless an OO language supports inheritance, it is not object-oriented. (Languages that support encapsulation, but not inheritance, are generally referred to as “object-based”.) However, there are several choices that the language designer has available to implement inheritance.

Single inheritance means that a class may have at most one direct superclass (that is, the class may have at most one direct ancestor in the class hierarchy). This means that the subclass inherits both the interface and the implementation (more on the terms *interface* and *implementation* later) of its superclass.

Multiple inheritance means that a class may have more than one direct superclass (that is, it may have more then one direct ancestor in the class hierarchy). How much is inherited and why is the subject of the remainder of this paper.

Why Use Inheritance?

There are many reasons various people cite as reasons to use inheritance. Among these are:

- Code Reuse
- Developing abstract interfaces
- Specialization
- Sub-typing

DEVELOPING ABSTRACT INTERFACES

One of the valid reasons to use inheritance is to develop abstract interfaces. An abstract interface is an interface defined on a superclass that specifies the behavior of some number of subclasses. Abstract interfaces are a very powerful way to insulate the system against changes. If all operations in the system are specified in terms of the abstract interface, new subclasses may be added to the (abstract) superclass and no changes are required to the portions of the system that use the abstract interface.

SINGULAR IMPLEMENTATION

Another valid reason to use inheritance is to provide a single implementation point for all operations that are, by the nature of the application, the same. This works best if this singularity is not incidental (that is, today they happen to be the same, but tomorrow they could well be different) but rather inherent in the nature of the application. Thus, they should always be the same.

DEVELOPING FRAMEWORKS

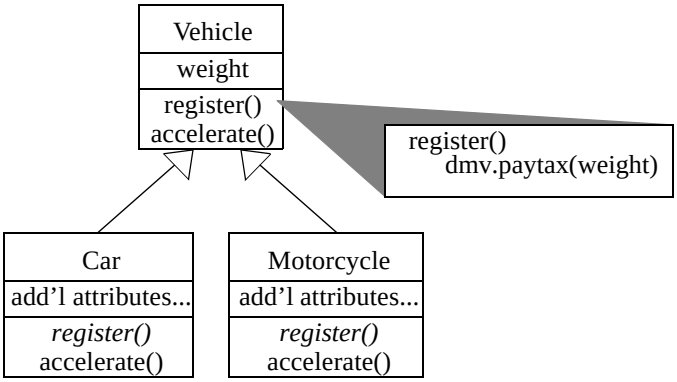
The final valid reason for using inheritance is for developing frameworks. These are cooperating groups of classes that completely specify the nature of a set of applications. Essentially, they provide a solution for an entire application domain. The behavior is customized for each specific application by extending the various base classes and providing (or adding) the behavior required for that particular application domain.

Implementation vs. Interface Inheritance

There are two kinds of inheritance that can be used in the object environment - implementation inheritance or interface inheritance.

IMPLEMENTATION INHERITANCE

Implementation inheritance occurs (simply) when a method is defined and implemented in a superclass and the subclass(es) inherit the method as implemented. They do not override it.

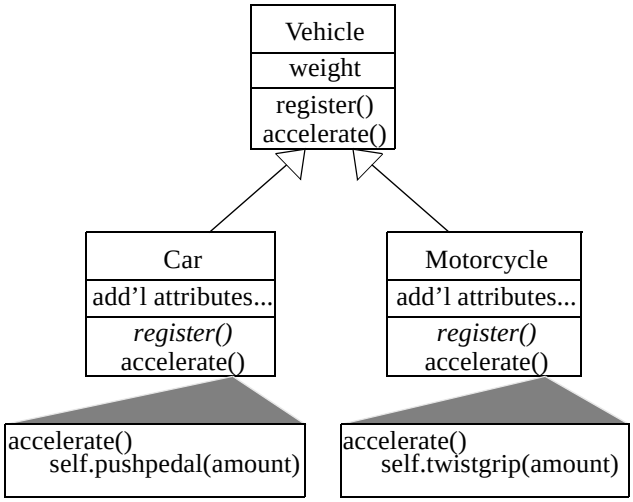


In this example, the superclass vehicle implements the *register* operation which, as shown here, simply uses the *weight* of the vehicle as the input to send a *paytax* message to the dmv (Department of Motor Vehicles) class. Both of the subclasses of Vehicle (Car and Motorcycle) inherit the implementation of paytax defined in Vehicle. (Note: register is depicted in each of the subclasses only to indicate that it is inherited by each. In most diagramming notations, it would not be explicitly depicted - hence, it is in italics.)

This is an example of implementation inheritance. It is used because all vehicles, regardless of their subtype, do exactly the same thing when they must be registered.

INTERFACE INHERITANCE

Interface inheritance occurs when a method is defined in a superclass and overridden in the subclass regardless of whether or not it is implemented in the superclass. With interface inheritance, the subclass is simply inheriting the ability to respond (in a polymorphic way) to exactly the same message as the superclass, or other subclasses. Thus, callers may utilize the methods of the subclass when addressing collections of the superclass in a polymorphic way.



In this example, accelerate is different for Cars and Motorcycles. We have no intention that Motorcycles will accelerate in the same way as Cars. To accelerate in a Car, you must push the pedal down, whereas to accelerate a motorcycle, you must twist the handlebar grip. But the *accelerate* operation is common to both and can be invoked on any kind of Vehicle, regardless of its type.

In this case, the subclasses only need to inherit the interface of *accelerate* from Vehicle. This is so that any member of a collection of vehicles may be sent the accelerate message. However, each subtype must implement the accelerate operation in a way that makes sense for it.

HOW LANGAUGES USE IMPLEMENTATION VS. INTERFACE INHERITANCE

The above description describes the exact nature of interface versus implementation inheritance. But how various systems utilize these concepts is also important to understand. Traditionally, inheritance in OO languages has been classified according to the implementation inheritance provided by the language. That is, when a (super)class is subclassed, all operations defined (and implemented) in the superclass are inherited by the subclass(es). If only one direct superclass is allowed for a subclass, then this is referred to as *single inheritance*. If multiple direct superclasses are allowed for a subclass, then this is has been known as *multiple inheritance*.

In each case, however, when a (sub)class inherited from a superclass, all operations defined and implemented on the superclass are inherited (although they can be overridden). This means that all languages supported implementation inheritance.

Although multiple inheritance often seems handy, it frequently creates problems in the development and maintenance of class hierarchies. It is also significantly more difficult to implement from a compiler perspective than single inheritance. Further discussions of both of these are beyond the scope of this particular paper, however, these are some of the reasons that multiple inheritance support has not found its way into many environments.

More recently, Java introduced a new wrinkle into inheritance. Java allows single inheritance of implementation, but multiple inheritance of interfaces. Thus, a (sub)class may have only one superclass from which it inherits code, it may have multiple superclasses that provide an abstract interface for part of its behavior.A class may *extend*[1] only one class, but it may *implement*[2] multiple others. (This is the type of inheritance also introduced for Forte Version 3.)

Official support for interface inheritance is new. Historically, OO languages have only supported implementation inheritance. Although (as discussed above) it has always been possible to inherit only an interface, languages did not provide a facility to limit the inheritance to only the interface. With interface inheritance, however, a subclass may only inherit the interface; it must provide implementations for each of the methods defined by that interface.

Providing multiple interface inheritance is a compromise solution that should satisfy 98% of the need for multiple inheritance. Language designers have always seen a need for multiple inheritance. It is useful in a small number of situations. However, it is widely misused and often creates designs that are unextendable and unmaintainable. Unfortunately, it was not possible to give multiple inheritance to a select and knowledgeable few, and not give it to the rest at the same time. So, language designers wrestled with a question similar to: “Explosives are useful in a small number of situations in real life. So, do we let everyone have access to explosives or no one?” Most of us, even if we consider ourselves competent to use explosives, would not advocate general access to them. (Fortunately, with explosives, we can entrust them to a few skilled/licensed individuals; not everyone needs to have access to them.)

The reason multiple interface inheritance works so well is that most operations that you would inherit from a second superclass are those that need to be implemented differently for each class that inherits them. We want to use inheritance because we want to include them in collections of the (abstract) superclass and we want to utilize polymorphism in dealing with them. However, even in languages that support multiple implementation inheritance, those methods are often overridden to provide the correct behavior in the subclass.

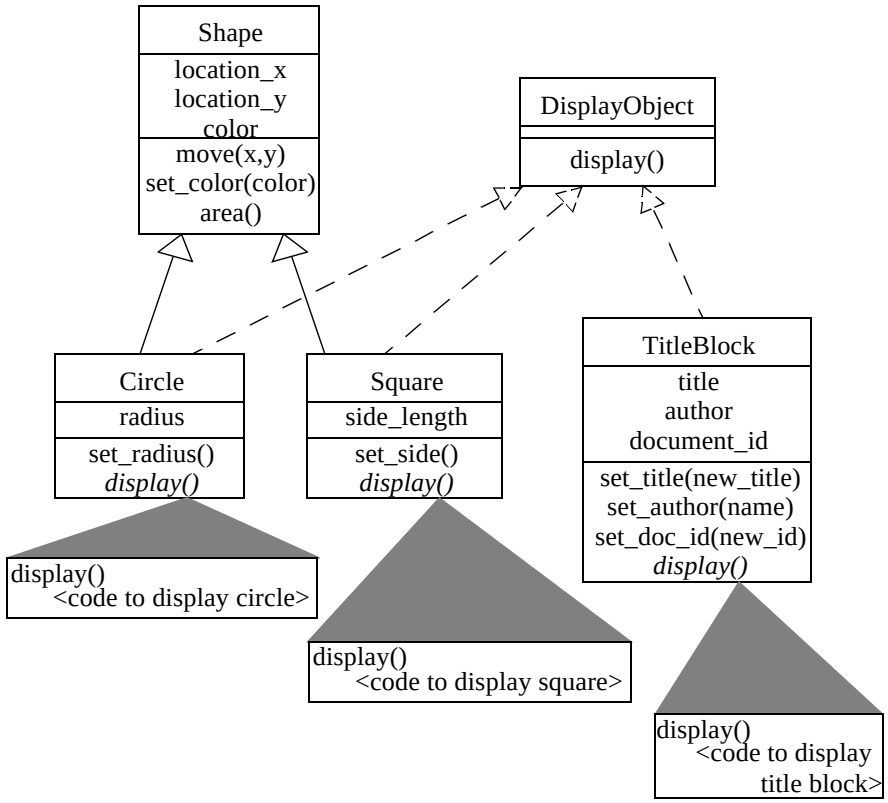
Using Interface Inheritance

Interface inheritance is useful whenever a class is identified that could properly be included into two different base classes. In most situations, one class will clearly be the one that contains the implementations that the class needs to inherit, and the other then becomes a candidate for multiple interface inheritance.

1. *extends* is the Java keyword denoting that the subclass is inheriting the implementation of the superclass.
2. *implements* is the Java keyword denoting that the subclass is inheriting only the interface of the superclass.

INTERFACE INHERITANCE

Consider geometric shapes such as circle, square, triangle, etc. These have certain behavior that makes sense to inherit from some superclass called Shape. However, some of these shapes also can be displayed. If shapes were the only thing that could be displayed, then that behavior could be added to Shape. However, there are other things in the system that we also want to be displayable (for example, the title block).



Each of the Shape subclasses inherits the implementation of Shape (shown by the solid inheritance arrow). Plus, each of them inherits the interface of DisplayObject (shown by the dashed inheritance arrow). TitleBlock also inherits the interface of DisplayObject. Each of the classes that inherit this interface must then provide an implementation for it (in this case, for the display method).

In this example, it was only necessary to inherit from a single interface class. However, it is possible to inherit from multiple interface classes. So, if there was also a requirement to have PrintableObjects, a separate interface could have been inherited for that as well. That, of course, would have required implementation of the methods defined in PrintableObject by each class that inherited from that interface class.

REIMPLEMENTING METHODS

Upon casual observation, it may seem like it is a nuisance to reimplement display for each of the classes that inherit the interface. It would be easier to just provide an implementation in the DisplayObject class and inherit that implementation. However, upon closer inspection, we find that each class that inherits the display operation must do something a little different depending upon the type of the class. So, while this means

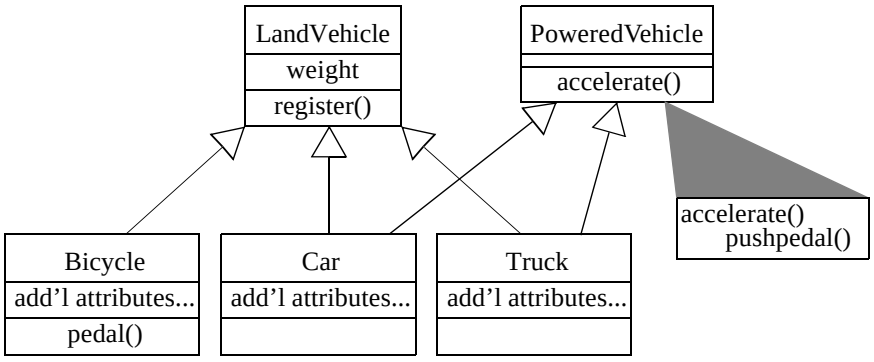
that we can treat collections of DisplayObjects uniformly (by sending them all a display message), it makes no sense to inherit an implementation that must be overridden for each subclass. In this case, interface inheritance is perfectly adequate.

Simulating Multiple Implementation Inheritance with Interface Inheritance

There are times when true multiple implementation inheritance is useful. While it is not possible to achieve this in exactly the same way as with a language that provides true multiple inheritance, there is a design pattern that can be applied to greatly simplify this problem.

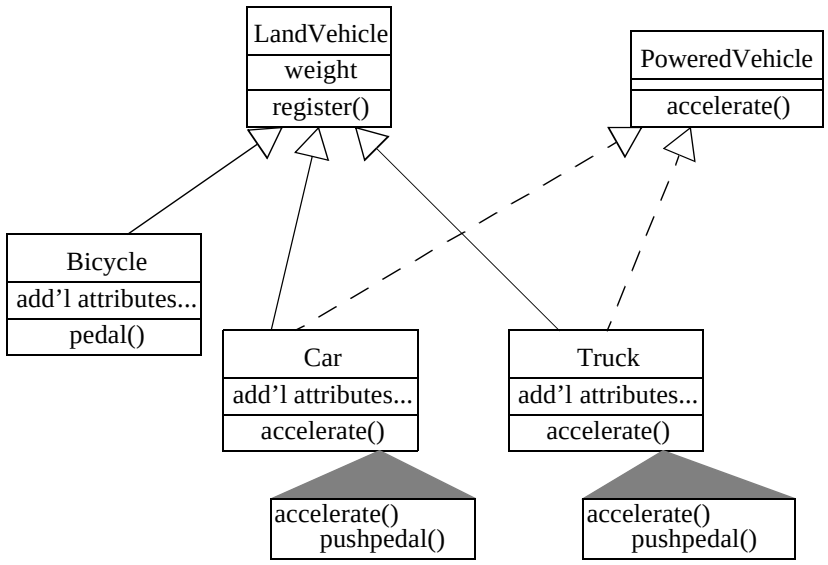
THE PROBLEM

In this example, Cars, Trucks and Bicycles are all subclasses (inheriting the implementation) of LandVehicle. However, Car and Truck also would like to inherit the implementation of PoweredVehicle. The desired structure is shown in the diagram below.



ONE POSSIBLE SOLUTION

Unfortunately, this is not possible if a language only supports single implementation inheritance. Using interface inheritance, as described above, most designers would be led to the solution shown in the following diagram. However, if we really wanted implementation inheritance, this means that these implementations should always be the same. This solution allows one to change independently of the other. The consequence of this is that if changes are being made to the implementation of accelerate in one subclass, the person making the changes must understand that those changes also apply to the other subclass, and make the changes there as well. If they only make the changes in one place, or they make an error in one of the places, the behavior of the classes will diverge (a polite way of saying that a bug has just been introduced).

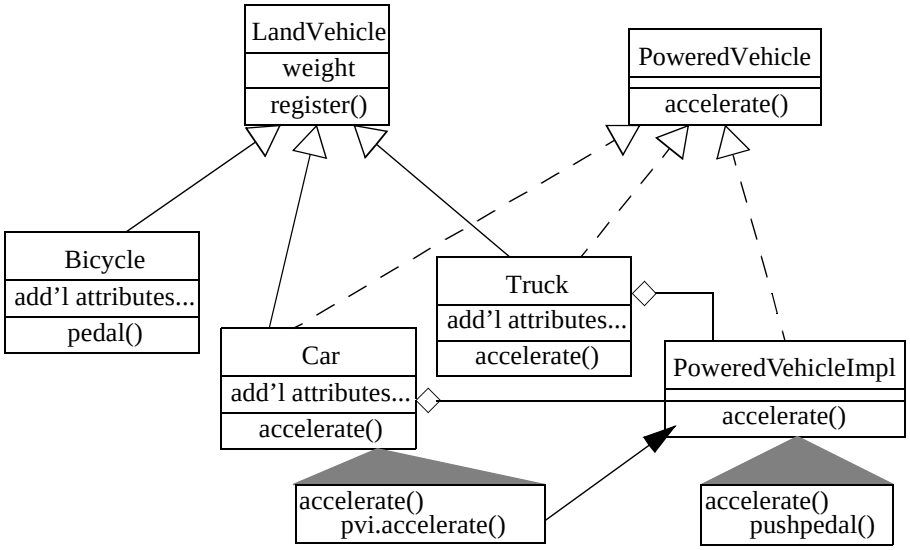


A BETTER SOLUTION

However, using a combination of single implementation inheritance, multiple interface inheritance and delegation, the desired effect can be achieved.

How this is accomplished is shown in the following diagram. An interface class (PoweredVehicle) is created, and both Car and Truck inherit the interface from that class. However, instead of having both Car and Truck contain the full implementation of *accelerate*, another class (PoweredVehicleImpl) is created that also inherits the interface of PoweredVehicle but, in addition, contains the implementation of all the methods defined in PoweredVehicle.

Now, Car and Truck simply create a private local reference to an instance of a PoweredVehicleImpl. Then, both Car and Truck provide implementations of the methods (in this case, *accelerate*) by delegating to the corresponding implementation in their instance of PoweredVehicleImpl.



In the example above, if the *accelerate* message is sent to an instance of Car, this invokes the *accelerate* operation on its private instance of PoweredVehicleImpl (in this case, passing a reference to *self* so that PoweredVehicleImpl can invoke methods in it if necessary.)

In the diagram above, only the implementation for *accelerate* in Car is shown, but the implementation in Truck would be identical.

Using this solution, if the way that Cars and Trucks accelerate changes, there is only one place to make that change, in PoweredVehicleImpl.

DISCUSSION

The first solution is perhaps a little easier to understand. However, it doesn't really capture the intention that Car and Truck are always intended to be the same. The second approach, while slightly more complicated, does capture that intent.

In the future, if another type of vehicle (GolfCart) were added that had a voice command system instead of a pedal that was pushed, then the accelerate method on GolfCart could simply be implemented with the requisite behavior and not delegate its accelerate to PoweredVehicleImpl. Also, if the behavior of Car changed (due to some unforeseen change in the nature of the application), the implementation of accelerate in Car could be changed to the new way without impacting the other subtypes (Truck, and any other siblings it might have by then).

Conclusions

Multiple interface inheritance provides a very significant extension to the capabilities of a single inheritance language. In most cases, it is sufficiently powerful to satisfy the needs of the application. Also, it does it in a way that cannot be misused by those who do not understand proper inheritance principles. And, for those situations that require true multiple implementation inheritance, there is now a pattern available to even help with that situation.