Urmans Tamm

Multiple Inheritance

Review paper for Programming Languages Research Seminar

Tartu 2012
# Table of Contents

Abstract ......................................................................................................................................................... 3  
Inheritance and multiple inheritance ........................................................................................................ 4  
  Overview of problems related to MI ..................................................................................................... 5  
  MI in existing languages ...................................................................................................................... 7  
Behaviors ..................................................................................................................................................... 9  
Composition .............................................................................................................................................. 9  
Delegation .................................................................................................................................................. 11  
Subtyping ................................................................................................................................................... 12  
Flavors and mixins ..................................................................................................................................... 13  
Traits ......................................................................................................................................................... 16  
Conclusion ............................................................................................................................................... 17  
Reference .................................................................................................................................................. 18
Abstract

“Multiple inheritance is good, but there is no good way to do it.” [Cook 1987]

With the usage of multiple inheritance there have always been a certain amount of uncertainty and doubt. The main reason is that MI is hard to implement efficiently so designs could take advantage of it. Despite a lot of criticism MI is generally considered a useful feature in languages that provide it and frequently requested in the ones that don’t. Situations where the usage of MI would be inevitable rarely exist and there’s always an alternative way through various other techniques and behaviors that programmers often resort to. Implementation of MI in the right way might make the architecture even better if used correctly but misusages tends to lead to some serious problems that are hard to be tracked down and fixed.

The paper is divided into parts where the author tries to give an overview of the main concept of MI, problems related to it and how MI or its workarounds are implemented. Also what are the alternatives for MI? Flavors, mixins and traits are going to be investigated. How they are implemented in various languages and also some problem situations explained to clarify their working principles.
Inheritance and multiple inheritance

Inheritance is commonly regarded as the feature that distinguishes object-oriented programming from other modern programming paradigms, but researchers rarely agree on its meaning and usage.[Taivalsaari 1996]. A general, but not very precise, definition of inheritance would be that it is a way in OOP programming to reuse code of existing objects. Simple, non-OOP languages lack this kind of feature for being designed in way where a program is seen as a list of tasks instead of interaction between encapsulated objects. Single inheritance is the simplest form of inheritance; it allows a class to inherit from (at most) one superclass becoming the sub-class of its parent. Such limitation also makes single inheritance inadequate for expressing classes that share features not inherited from their common parent because they don’t have access to them. This gives way to a problem of duplicated features because some methods and characteristics need to be redefined in the sub-classes. A common way to avoid this kind of code duplication is to implement certain methods “high enough” in the class hierarchy that they are available to all the classes where they are actually required and thus can be inherited. Duplication problem gets solved but results in inappropriate hierarchy, i.e a parent class gets a load of methods that have nothing to do with the containing class.[Schärli 2006]

To deal with this kind of limitations and problems researchers have developed various forms of inheritance over the years. This brings us to multiple inheritance which is one of the more controversial topics in OO-languages design. The core concept of multiple inheritance is that a class can have more than one parent class to be inherited from. Instead of duplicating code or placing it into classes where it actually doesn’t belong it can be inherited from various sources. MI is often useful for implementing complex OO-structures where it might be complicated to arrange all interfaces and implementations in a single tree-like hierarchy. That way a class is able to support and reuse the implementations of multiple interfaces from different class hierarchies that can be unrelated to each other.
Overview of problems related to MI

The idea of MI dates back to mid-eighties but despite the amount of time passed multiple inheritance has not achieved wider acceptance in OO-languages. [Taivalsaari 1996] Although MI is often useful it also introduces quite a few complications. Classes play two competing roles. First of all a class can be defined as a generator of instances, encapsulating all the functionality, and after that having the secondary role of being a unit of reuse. Therefore a class must contain a minimal set of features to be used together in a sensible way. These two roles tend to conflict. For instance one may need to write wrapper methods for specific tasks within the given class that can be difficult to factor out as reusable classes later on. Because of that inheritance can very easily be misused or even abused. MI adds an additional level of complexity to it by allowing multiple classes to be inherited from.

Besides single and multiple inheritance there exists one special form of inheritance called mixin inheritance that is going to be covered later. Each of these forms provides different answers to following problems:

1. decomposition: how we decompose a software base into suitable units of reuse
2. composition: how we compose these units to obtain a class hierarchy suitable for our application domain

[Schärli 2006]

While decomposing software i.e trying to create suitable units of reuse using MI techniques a situation of duplicate wrappers may result. Assume that class A contains methods read and write to provide unsynchronized access to some data. Now if the access to the data needs to be synchronized a new class is created for the task called SyncA. Suppose that there exists another class B that also needs synchronized access to the same data. So first of all we create a new class SyncB and to reuse code more efficiently we factor the code used for synchronizing access to a separate class called SyncReadWrite. Now by using multiple inheritance we let both SyncA and SyncB extend it. Situation gets hazy if the methods defined in SyncReadWrite are using the super keyword to call methods of read and write to turn to corresponding methods in parent classes of A and B. Because super-sends are statically resolved in most forms of MI such as C++ and Eiffel the super-sends in
SyncReadWrite methods would resolve to calls to its own parent class and not to A or B. A workaround for it would be removing the super-calls and making methods in SyncReadWrite abstract implementing them in SyncA and SyncB. Unfortunately this creates just duplicate code and makes the usage of class SyncReadWrite pointless.[Schärli 2006] Previous situation is illustrated in Figure 1. Situations like this can be quite easily be solved using mixins as we can see later on.

![SyncReadWrite Diagram](image)

*Fig 1:* A multiple inheritance class diagram presenting an incorrect attempt to factor out a synchronization wrapper. “acquireLock” and “releaseLock” are used to set locks on data during concurrent access to them. [Schärli 2006]

In the terms of composition the usage of MI can lead to *conflicting features*. One of the most serious of these is the ambiguity that arises when these conflicting features are inherited via different paths. This creates a situation that is sometimes called the “diamond problem” or “the fork-join-inheritance”[Sakkinen 1989]. The scheme of the diamond problem can be seen in Figure 2.
Fig 2: A diamond class inheritance diagram. Here class D inherits from classes B and C that themselves are the siblings of class A. Now if D calls a method defined in class A and does not override it while classes B and C have overridden it differently from which class is the implementation going to be inherited?

MI in existing languages

C++ is known for its support of multiple inheritance. For ambiguity and the diamond problem special virtual classes are used. This kind of technique is also known as virtual inheritance. A virtual class means that any object that belongs to the virtual base class becomes a direct base for the derived class and all other classes that derive from it. If in Fig. 2 classes B and C are virtually derived from class A then A becomes a direct base class of B, C and thus also class D. Now if one calls a method from class A in class D without overriding it the call would not be ambiguous anymore.

Java has disallowed MI for making language too complex and ultimately creating more problems than it fixes. Something similar to MI is created by using interfaces and abstract classes. An interface in Java is a class that groups methods with empty
bodies, an abstract class differs by that it provides partial implementation leaving it to subclasses to complete the implementation. Ambiguity is avoided because there is always only one implementation to a specific method if a class implements an interface or extends an abstract class.
Behaviors

Composition

When talking about composition we have to keep in mind that there exist two types of composition. Function composition is a mechanism to combine simple functions into more complicated ones by passing the result of a function to the next one as its argument. Say we have two functions $f$ and $g$ as in $z=f(y)$ and $y=g(x)$ then composing them means we get $z=f(g(x))$. This kind of act is supported by all programming languages.

Object composition is a way to combine simple objects or data types into more complex ones. An object itself in an OO-programming language can be thought of as a composition by encapsulating certain distinctive features. Inheritance can be described as “is-a” relationship: employee “is-a” person. An example code is given in Fig 3.

```java
class Person {
    String Title;
    String Name;
    Int Age
}
class Employee : Person {
    Int Salary;
    String Title;
}
```

*Fig 3: The Employee inherits from Person.*

Composition at the same time presents “has-a” or “uses-a” relationship: employee “has-a” or “uses-a” person, i.e employee has and can use the characteristics that a person has. This also implies that the child cannot exist without the context of the parent. An example code is given in Fig 4.
class Person {
    String Title;
    String Name;
    Int Age;

    public Person(String title, String name, String age) {
        this.Title = title;
        this.Name = name;
        this.Age = age;
    }

    void printName() {
        print(„Person.name“);
    }
}

class Employee {
    Int Salary;
    private Person person;

    public Employee(Person p, Int salary) {
        this.person = p;
        this.Salary = salary;
    }

    void printName() {
        person.printName();
    }
}

Fig 4: The Employee class has a Person object. It does not inherit from it but instead gets an object passed to it and uses its features.

Object composition can be used as an alternative to inheritance, because by assembling objects new and more complex functionality can be obtained. Inheritance has a short-coming by early binding the implementation. By inheriting from a super-class we have locked the sibling into a particular implementation used by the parent. Although we can override methods in the sub-class certain aspects can only be changed in the parent class. With composition we can late bind the implementation by taking the Person object used in the example as a constructor object and provide the implementation later. For that objects being composed should have well-defined interfaces. By using object composition instead of inheritance each class can be more focused on its task and by that the class hierarchy can be kept small.[ Gamma, Helm 1994] So if we are talking about code re-use composition should be preferred over inheritance. A downside of composition is that when using inheritance the super-class can call the methods of sub-classes but this isn’t so with composition.
Delegation

Delegation is a way to make composition as powerful for reuse as inheritance.[Gamma, Helm 1994] The re-usability and extensibility in architecture is achieved through a concept of delegating a task over to another part of the program, a class or an object, to do the work. This is analogous to subclasses deferring requests to parent classes. But with inheritance, an inherited operation can always refer to the receiving object for example through the “self” keyword in various languages. To achieve the same with delegation the receiver passes itself to the delegate to let the delegated operation refer to the receiver.[Gamma, Helm 1994] A wrapper method for example can be called a form of delegation. Delegation is often referred to as consultation or aggregation in modeling.

Delegation tends to be useful and more flexible in some situations like:

1. Given classes A and B need to be used in a matter where A is derived from B and one wants to enhance A, but A is final and can no further be sub-classed.
2. If an object can play many roles in a system the combination of sub-classing may grow exponentially but using delegation would result in fewer classes.
3. If an object needs to be a different sub-class of the parent class on specific occasions it is better to use delegation that being more dynamic while inheritance is strictly a static relationship.

The main advantage of delegation is that it makes it easy to compose behaviors at run-time and even to change the way they’re composed while class inheritance is defined statically at compile-time. On the other hand the disadvantages of delegation include larger number of objects in memory and an extra level of object indirection that makes the code more difficult to read and understand. The use of delegation also forces user to write a series of small methods that do nothing more than just forward responsibility for a method to a delegate while such work would be automated by MI thus lessening mistakes that might slip in with such repetitive methodology.[Viega, Tutt 1998] Delegation is a good design pattern only when it simplifies more than it complicates.[Gamma, Helm 1994]

In Fig 4 when calling Employee.printName() “Person.name” gets printed because class Employee delegates the method printName() to a given object of class Person.
Subtyping

Subtyping is a form of polymorphism where object $A$ is considered to be the subtype of object $B$ if it can appear everywhere $B$ is requested: $A$ can be treated as an instance of $B$. For that in different programming languages $A$ doesn’t always need to be derived from $B$, it just has to be *substitutable*. Vice versa even if $A$ is the subclass of $B$ it doesn’t automatically mean that it also is a subtype of $B$. For example consider the situation where a “Square” class is derived from the “Rectangle” class. Here the “Square” object expects the height and width parameters to be equal and if the “Rectangle’s” setter method sets their values independently the results can be unexpected. This kind of behavior was first introduced by Liskov in 1987 and therefore named *Liskov’s substitution principle* afterwards.

In Java language to define a class $A$ as a subtype of class $B$ then it has to extend $B$ or implement it if $B$ is an interface. Here we have actually two choices to choose between depending what we want to achieve:

- **implementation inheritance** – extending a parent class for mostly just re-using its code and overriding it to introduce new functionality where needed.
- **interface inheritance** – an interface is a type of abstract class that doesn’t provide any implementation. So if class $A$ implements an interface class $B$ it has to provide its own implementation for the methods and properties defined in $B$. We see the effects of polymorphism in action when we pass the object $A$ as an argument to a method that takes a parameter of type $B$. As object $A$ has the same method and property specifications as $B$ it should be compatible with the context where $B$ is being used.
Flavors and mixins

Inheritance allows us to choose specific superclass or classes when using MI and extend or to specialize them. However we would get much higher reuse if we could specify subclasses in a way that we can apply them to different super-classes as needed. Flavors, being the first solution to this kind of needs, were early predecessors of mixins developed by A.Moon in the late seventies and early eighties as an extension for the Lisp programming language being small incomplete implementations of classes that could be “mixed in” at arbitrary places in the class hierarchy. From that the term “mix-in” and the style of programming was introduced. [Moon 1986]

Mixins are subclasses used to specialize the behavior of a variety of parent classes. That is achieved by defining new methods in the mixin class that perform some actions and then call the corresponding parent methods. Mixins classes just describe reusable pieces of functionality that can be attached to other classes using multiple inheritance. That is a class or an object can inherit its functionality from one or more mixins and by inheriting we mean a way of collecting functionality. For this kind of behavior they are often referred to as abstract subclasses. Mixins are abstract classes for the reason that they cannot be instantiated. [Bracha and Cook] Some of the functionality of mixins can be provided using interfaces like in Java language with the limitation that an interface can only specify what the class must support and not providing an implementation for it. The reasons to use mixins are that they extend existing classes in new areas without having to edit their source code and ease the creation of new classes by providing a set of functionalities that can be combined when needed. Yet do mixins come with their own limitations.

Mixins are composed linearly: all mixins used by a class are inherited one at a time. In Figure 5 we have a situation where two mixin classes are applied to the “MyRectangle” class. The implementations of the method “asString” in the mixins first call the parent method via the “super” keyword and after that print some specific info about their own state. Implementation of the “serializeOn” function is something similar. This kind of linear hierarchy is the main source for the lack of control over the composition of mixins.
Let’s say that for some reason we want to serialize our class so that the “rgb” value appears before the “borderWidth”. This means that “MColor” needs to be applied before the “MBorder”. This also means that the order of the “asString” methods are changed and the color attributes get printed before the border attributes which we might not want. So if we need to customize how the features are composed we need to modify the mixins involved by potentially breaking all the other clients using them. [Schärli 2006]

Features defined in mixins appearing later in the inheritance hierarchy override all the features of earlier mixins that are named identically. This gives way to fragility problem. Adding a new method to a mixin can mess up the previous behavior in other parts of the system and it may be problematic to reestablish the original state. Suppose that the “MBorder” mixin does not define a method “asString”. This means that the implementation used in “MyRectangle” will be the one specified in “MColor”. Let’s say that later the corresponding method is added to “MBorder”. Because of the total ordering this implicitly overrides the previous implementation provided by “MColor” and to achieve the original behavior of “MyRectangle” class one needs to change more of the mixins involved in the composition. [Schärli 2006]

Fig 5: Lack of control in composition of mixins. [Schärli 2006]
Mixins can be composed the same way as functions to create *composite mixins* first introduced by Bracha in 1992. This means that result of a mixin is the argument for the next. So in Figure 5 we can combine the two mixins to a composite mixin “MColorAndBorder” and use that to define the “MyRectangle” class. As composite mixins also provide only a linear composition this doesn’t help us much and previously described problems occur during the definition of the composite mixin. The only way would be altering the source code of mixins by breaking them into smaller units, creating so called “glue code”. For example we can take the parts where color name and border width get printed and place them into separate mixins giving them unique names so they wouldn’t be overridden later on. But by doing that our hierarchy grows bigger and makes the program comprehension much more difficult having the glue code scattered around over many different entities.
Traits

Traits are purely units of reuse that provide a way for unrelated classes to share code. They are presented as an alternative to both multiple inheritance and mixins and have been gathering wider popularity lately. Languages that support traits include for instance PHP and Smalltalk. When in need of factoring out code in classes instead of using inheritance it can simply be put inside a trait. Traits are a set of methods that implement behavior being completely divorced from any class hierarchy at the same time. [Schärli 2006] By also providing implementation not just type signatures traits principally differ from interfaces. An important aspect about traits is that they have no state variables. Besides providing a set of methods traits also require some for parameterization.

To make it more clear let’s consider an example of drawing graphical objects. Each object can be decomposed into two aspects: geometry and the way it’s drawn on canvas. For that let’s create a trait called “Drawing” that has a method “draw” for drawing the given object. To use this method properly we first need to define the bounds of the circle. For that we define the corresponding method in the “Circle” class and pass it as a parameter to the “Drawing” trait.

Traits can be composed in similar manner as mixins. The main difference lies in the way it is achieved. Opposed to mixins’ linear composition traits can be composed in arbitrary order. Combining two or more traits comes with a possible side-effect in naming-conflict if we provide identically named methods. [Schärli 2006]

When talking about MI the most principle difference is in the inheritance part. A trait is not inherited from but rather included or “mixed-in” thus becoming a part of “this class” – context of the class that uses trait’s characteristics.
Conclusion

In conclusion it can be said that there isn’t always a way to tell which approach should be preferred over another. Multiple inheritance can prove itself useful and more flexible in situations where a strict tree-like inheritance hierarchy lacks possibilities to make code re-use more efficient. On the other hand there are always plenty of other options to be used depending on the specific programming languages. Mixins, traits, abstract classes and interfaces were covered in this paper. Yet they all come with their own limitations as we saw. Some studies have suggested to give up using multiple inheritance altogether for it being too complex. For that and the lack of knowledge multiple inheritance can often be misunderstood and therefore should rather be avoided.
Reference


[4] “Traits: A mechanism for fine-grained reuse” Schärli 2006

[5] "Design Patterns - Elements of Reusable Object-Oriented Software," Gamma, Helm 1994
