# Parameterized Inheritance vs. Multiple Inheritance

Amit Patel Vitaly Shmatikov John Mitchell

Department of Computer Science
Stanford University
Stanford, CA 94305-9045
{amitp, shmat, mitchell}@cs.stanford.edu
Phone: (650) 723-8634, Fax: (650) 725-4671

13 Feb 1997

Abstract

In this paper, we demonstrate that multiple inheritance can be expressed as a combination of single inheritance, parameterization, and interface subtyping. We propose a language construct called "phunctor" as an alternative to multiple inheritance. Phunctors combine single inheritance and constrained parameterization to provide required functionality in the contexts where multiple inheritance is used, while retaining the benefits of static type checking. We argue that using parameterized inheritance instead of multiple inheritance leads to more robust class hierarchy designs. We discuss how phunctors can be implemented efficiently and show that phunctors facilitate several common design patterns.

## Contents

# 1 Introduction

Our goal in this paper is to evaluate an alternative to multiple inheritance which is based on combining single inheritance, parametric polymorphism, and interface subtyping. This objective is particularly important for languages such as Java that allow subtyping of class interfaces to be independent of class implementation inheritance but do not provide support for multiple inheritance. In this case, if some form of parameterization is added to the language, then multiple inheritance functionality can be obtained "for free" with an appropriate hierarchy design. We are also interested in languages such as C++ that support parameterization and multiple inheritance but not multiple subtyping, since we believe that adding multiple subtyping to such languages would obviate the need for multiple inheritance.

We start by explaining the difficult issues related to multiple inheritance, and then propose a language construct called "phunctor" that provides the benefits of multiple inheritance without its drawbacks. A phunctor is a class definition parameterized over the base class. We give examples of phunctor usage, showing how common contexts where multiple inheritance is employed can be implemented using parameterized inheritance instead and what advantages are to be reaped from this approach. We emphasize the fact that parameterized inheritance supports well-structured class hierarchy designs and is preferable to multiple inheritance from the software engineering viewpoint. Some examples use C++, and others use OBSTACL, an object-oriented extension of ML which is being developed at Stanford. OBSTACL syntax is very straightforward, and the examples use only well-known language constructs such as `class`, `method`, etc.

We compare phunctors with similar constructs in other languages such as C++ templates, ML functors, and CLOS mixins, and discuss type checking and implementation issues. Finally, we demonstrate how several common design patterns can be rewritten without multiple inheritance using phunctors only.

# 2 Multiple inheritance

Multiple inheritance is one of the more controversial topics in object-oriented language design. In this section, we describe situations in which multiple inheritance is useful, and demonstrate some of the problems caused by it.

## 2.1 Advantages of multiple inheritance

Multiple inheritance is often useful for implementing complex object-oriented structures in which it is difficult to arrange all required interfaces and implementations in a single tree-like hierarchy. A class derived from more than one base class is capable of supporting multiple interfaces and reusing interface implementations from different, possibly unrelated class hierarchies.

### 2.1.1 Multiple implementation inheritance

Multiple inheritance is useful when a class has to combine properties that are implemented in different class hierarchies. For example, an edit dialog window may inherit from either `XWindow` or `CursesWindow` (system-specific implementations of an abstract `Window` class), and also from `Editor` (implementation of basic editing functionality). A more detailed example of this nature can be found in [6].

### 2.1.2 Multiple interface inheritance

Objects of some classes may need multiple interfaces if they are to be used in different contexts. For example, an Editor could have a Stream interface so that the contents of the edit buffer can be accessed as a stream of characters. This idea is formalized in [9] as the Adapter pattern.

If the programming language allows multiple interface subtyping independent of implementation inheritance, then multiple inheritance is not needed in order for a class to support more than one interface. This is one of the reasons why multiple inheritance is absent in Java [1].

Another example is provided by Smalltalk where mixin multiple inheritance would express the structure of the collection hierarchy more clearly [5].

## 2.2 Multiple inheritance problems

Even though it is often useful, multiple inheritance introduces quite a few complications. The main issues are the ambiguous methods problem, and the problem of redundant base class method calls in the diamond hierarchy, both of which are explained below. As a result, many developers shun multiple inheritance, and some language designers eliminate it altogether.

### 2.2.1 Ambiguous methods

Consider a class `EditDialogWindow` derived from two base classes, `Editor` and `XWindow`, both of which have a method called `draw()`. Essentially, this means that `EditDialogWindow` inherits two methods with the same name. If `EditDialogWindow` does not define a new `draw()` method that overrides both `Editor.draw()` and `XWindow.draw()`, then `EditDialogWindow.draw()` is ambiguous.

There are two possible ways to treat this situation:

- Ambiguous methods are considered a static error, and signaled as such by the compiler. A variation of this approach is to only signal an error if a call is made to `EditDialogWindow.draw()` since it cannot be resolved.

- Give precedence to either `Editor.draw()`, or `XWindow.draw()` to ensure that all calls to `EditDialogWindow.draw()` are unambiguous. Linearization is one of the possible ways to determine the precedence order.

#### 2.2.2 Diamond inheritance

Suppose that class `ScrollableEditor` has a single ancestor `Window` which is common to more than one of its base classes. For example, `Editor` and `ScrollableWindow` are derived from `Window`, and `ScrollableEditor` is derived from both `Editor` and `ScrollableWindow`. In this case, `Window`'s data members may appear in `ScrollableEditor` either once, or twice. If only one copy of `Window` is included in `ScrollableEditor` (i.e., `Editor` and `ScrollableWindow` are *virtual base classes* of `ScrollableEditor` in C++ terminology), then the so called "diamond" (or shared) inheritance takes place. Diamond inheritance leads to a subtle problem when a `Window` method is called by several of the derived classes' methods. Consider the following example:

```
class Window /* W */
   method draw() = ... erase the interior ...
end;
```

```
class EditWindow /* EW */
   inherits Window;
   method draw() =
      Window.draw();
      ... draw text ...
end;
```

```
class ScrollableWindow /* SW */
   inherits Window;
   method draw() =
      Window.draw();
      ... draw scrollbar ...
end;
```

```
class ScrollableEditor /* ESW */
   inherits EditWindow;
   inherits ScrollableWindow;
   method draw() =
      EditWindow.draw();
      ScrollableWindow.draw();
end;
```

We use the abbreviations $W$, $EW$, $SW$, and $ESW$ to refer to `Window`, `EditWindow`, `ScrollableWindow`, and `ScrollableEditor` respectively.

The code of $ESW$.`draw()` is perfectly reasonable from the $ESW$ implementor's viewpoint: it simply calls the methods inherited from $ESW$'s parents and adds some behavior of its own. However, the result is that $W$.`draw()` is called twice which most likely was not intended by the programmer. The window interior will be erased after the contents of the edit buffer have been drawn by $EW$.`draw()`, and the window will appear blank.

## 2.3 Multiple inheritance in existing languages

### 2.3.1 C++ approach: rely on the programmer

In general, C++ relies on the programmer to specify the semantics of multiple inheritance. The programmer can refer to any inherited method using base class name as a prefix, thus avoiding ambiguity [8]. C++ takes the view that name conflicts between inherited methods are program design flaws that have to be addressed explicitly by the programmer [6]. Therefore, C++ does not provide a rule for resolving ambiguities. Attempts to call an ambiguous method are signaled as errors at compile-time. The standard solution is to define a new method with the same name in the derived class, and in the body of this method call the conflicting inherited methods in some order. This puts the burden of specifying the order on the programmer. A similar approach in Eiffel [11] requires renaming in the case of ambiguous methods. Note that in C++, if conflicting methods are inherited from base classes but not actually called by the derived class's clients, no error is signaled [8].

C++ does not provide a good solution to the problem of redundant method calls in diamond inheritance. The approach suggested by the designer of the language [16] essentially requires the implementor of $ESW$.`draw()` to explicitly call special "helper" methods of all ancestor classes without relying on calls to inherited `draw()` methods. This in some sense defeats the purpose of inheritance since a class implementor has to know the class hierarchy beyond his class's immediate parents and directly refer to methods implemented in ancestors. This kind of design is very sensitive to changes in the class hierarchy and makes implementing class libraries very difficult. The code that uses the library depends not only on the terminal classes, but also on the entire class hierarchy, which makes client-transparent changes to the library impossible.

### 2.3.2 CLOS approach: linearize

CLOS [15] and other object-oriented descendants of Lisp solve the ambiguous methods problem by linearization. The listing order of base classes is considered significant, and methods from the classes that are listed earlier take precedence over methods with the same name from the classes that are listed later. This approach is also sensitive to changes in the class hierarchy [14], and sometimes leads to unexpected results. Some of the problems with the CLOS linearization algorithm are fixed in Dylan [3].

### 2.3.3 Java approach: avoid multiple inheritance

Early object-oriented languages such as Simula67 [7] and Smalltalk-76 [10] had only single inheritance. Since the experience with multiple inheritance accumulated by C++ users does not lead to any definite conclusions, the designers of Java decided that the complications introduced by multiple inheritance outweigh the benefits, and did not include it in the language [1].

It is worth noting, however, that Java does provide a mechanism for multiple *interface subtyping*. In the rest of the paper, we intend to demonstrate that common multiple inheritance designs can be expressed using single inheritance, interface subtyping, and parameterization. Java currently does not provide any form of parametric polymorphism, although this may change in the near future [2, 13].

## 3 Parameterized inheritance

In this section, we consider an alternative to multiple inheritance based on the parameterized mixin idea. *Mixins* are a programming idiom first introduced in the Flavors system [12]. A mixin is a lightweight base class responsible for a particular property. By inheriting from the main base class $C$ and several mixins, it is possible to create various modifications of $C$ and "mix in" new properties as needed. The mixin style of programming is sometimes cited as the justification for multiple inheritance [17].

The idea of using linearized mixins as an alternative to multiple inheritance is not new. Bracha and Cook [4] demonstrated that mixin-based inheritance can viewed as a generalization of both single and multiple inheritance. We are interested in the software engineering implications of this result. We consider various contexts in which multiple inheritance is used, and demonstrate that they can be reimplemented with parameterized inheritance. Parameterized inheritance provides additional benefits such as explicit base class ordering, parameter constraints, and the ability to add a property to a class more than once.

### 3.1 Phunctors

We are interested in a parameterized form of mixins that can be type checked at compile-time. We call the proposed language construct *phunctor* (this stands for parameterized inheritance functor).

A phunctor is a parameterized class definition. It can be viewed as a compile-time operator that takes a single parameter, which corresponds to the base class in conventional inheritance, and derives a new class from it. The difference with conventional single inheritance is that the same phunctor can be applied to different classes. See section 6 for the explanation of similarities and differences between phunctors and C++ templates, ML functors, and CLOS mixins.

A phunctor definition consists of the following two parts:

Parameter. Phunctor definition includes a parameter name and associates a constraint with it. Only classes satisfying this constraint may be passed as parameters to the phunctor. The constraint is a signature constraint: it lists which methods a class must have so that the phunctor can be applied to it. Constraints play a particularly important role in parameterized inheritance since they express the interface requirements on the parameter and enable static type checking of derived classes.

Class definition. This is a conventional class definition. The phunctor parameter is used as the base class. Therefore, all public and protected methods and data members defined in the parameter are inherited by the phunctor.

Given a class definition $C$, it is possible to automatically convert it into the phunctor form by defining a phunctor $P$ that has the same methods and data members as $C$, and applying $P$ to the empty class $E$ (or the root of the class hierarchy). The original class $C$ is equivalent to $P(E)$. If $C$ has a base class $B$, then $C$ is equivalent to $P(B)$.

### 3.2 Examples of phunctor usage

The best way to illustrate phunctor usage is to re-implement with phunctors several code fragments employing multiple inheritance. Examples in the following sections are separated based on how multiple inheritance is used in them. Following each example is a rule for transforming similar designs employing multiple inheritance to equivalent designs using phunctors. After the examples we give rules for transforming class hierarchies using multiple inheritance into class hierarchies using phunctors.

### 3.2.1 Multiple Interface Inheritance

Multiple inheritance is often used when a single class must support more than one interface. Some languages, such as Java and OBSTACL, support this use but not other uses of multiple inheritance. Multiple interfaces in Java allow the programmer to declare that a class satisfies one or more interfaces. Structural subtyping in OBSTACL allows a type to have several potentially unrelated supertypes. For example, a class might implement both the `Editor` and `Stream` interfaces, so that its instances can serve both as an editor window and as a stream of characters. To do this, it can can inherit from an `Editor` window class ($E$) and then add any code required to support the `Stream` interface:

```
class StreamableEditor /* SE */
  inherits Editor;
  implements Stream; /* second supertype */
  method read() = getText();
  method open() = lockEditBuffer();
  method close() = unlockEditBuffer();
end;
```

This class is declared to implement `Stream`, so objects of `StreamableEditor` can be used in functions that require `Stream`s. When those functions call a `Stream` method, it will be redirected to the appropriate `Editor` method by the forwarding functions.

We may want to do the same for subclasses of `Editor` such as `HTMLEditor` ($H$) and `LaTeXEditor` ($L$). Without multiple inheritance, it is not possible to inherit the forwarding functions from `StreamableEditor`, so these methods must be reimplemented:

```
class StreamableHTMLEditor /* SH */
  inherits HTMLEditor;
  implements Stream;
  method read() = ...;
  method open() = ...;
  method close() = ...;
end;
```

```
class StreamableLaTeXEditor /* SL */
  inherits LaTeXEditor;
  implements Stream;
  method read() = ...;
  method open() = ...;
  method close() = ...;
end;
```

If multiple inheritance is available, `StreamableHTMLEditor` can inherit from both `StreamableEditor` and `HTMLEditor`. It inherits the forwarding functions from `StreamableEditor` and the HTML editing extensions from `HTMLEditor`. Similarly, `StreamableLaTeXEditor` can be implemented without reimplementing the forwarding functions. The classes that adapt one class to use a different interface are called *Static Adapters* in [9]. A class that requires a different interface can inherit its implementation from one class and also inherit from an adapter class.

Alternatively, we could use phunctors. The adapter code is put into a phunctor and then added whenever needed:


Figure 1: Multiple Inheritance hierarchy compared with Phunctor hierarchy

```
phunctor StreamAdapter(W: Editor) /* S */
   inherits W;
   implements Stream;
   method read() = getText();
   method open() = lockBuffer();
   method close() = unlockBuffer();
end;
```

```
class StreamableEditor =
   StreamAdapter(Editor);
class StreamableHTMLEditor =
   StreamAdapter(HTMLEditor);
class StreamableLaTeXEditor =
   StreamAdapter(LaTeXEditor);
```

The multiple inheritance hierarchy and the phunctor hierarchy are shown in figure 1. Classes are shown as circles and phunctors are shown as triangles. A phunctor application is shown as a phunctor attached to either a class or to another phunctor application.

Even in the case where a class needs only to support multiple interfaces but not to inherit code from more than one parent class, multiple inheritance can be useful to inherit the code needed to forward messages. Using phunctors is more convenient than multiple inheritance in this case because a new class does not have to be created – classes such as `StreamAdapter(HTMLEditor)` and `StreamAdapter(LaTeXEditor)` can be used directly.

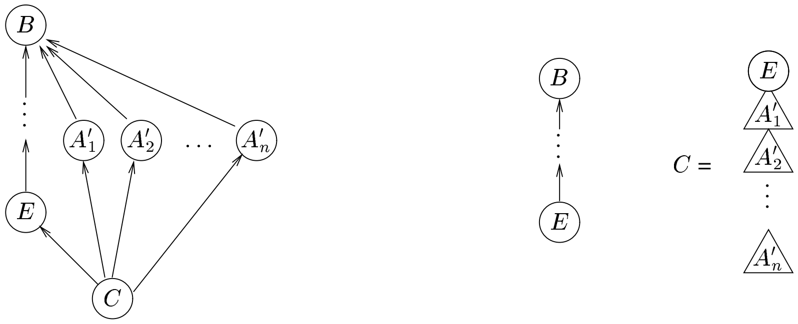



Figure 2: A set of adapters using multiple inheritance and using phunctors

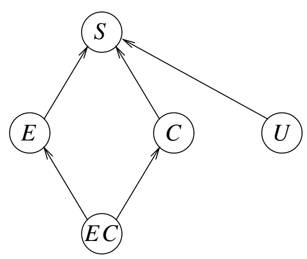
In general, whenever a class $C$ must support multiple interfaces $A_1, A_2, \ldots, A_n$ and inherit code from base class $E$, multiple inheritance is used with adapter classes $A'_i$ that translate the interface of some class $B$ to the interface $A_i$. Class $C$ inherits from $E$ and $A'_1, A'_2, \ldots, A'_n$. Given this hierarchy, phunctors can be used instead to create adapters $A'_i$ to take any class supporting the interface of $B$ and produce a class that also supports interface $A_i$. Then class $C$ is written as $A'_1(A'_2(\ldots A'_n(E)\ldots))$. Phunctors $A'_i$ can also be reused with other classes that support $B$'s interface.

An alternative to adding interfaces to class $C$ is creating a separate object with one or more interfaces. That object can forward all requests to the original object, after translating from the $A_i$ interface to the $B$ interface. This approach is described as *Dynamic Adapters* in [9]. Whenever a translation object is needed for some class $B$, a forwarding class $F$ can be written for $B$'s interface. This class has one private field, a pointer to an instance of $C$. It also defines methods to forward requests to that instance.[1] The phunctors $A'_i$ are applied to the forwarding class, producing $F' = A'_1(A'_2(\ldots A'_n(F)\ldots))$. All messages sent to instances of $F'$ are translated from the $A_i$ interface to the $B$ interface, then forwarded to an instance of class $B$, which in this case is an instance of class $E$.

#### 3.2.2 Multiple Implementation Inheritance

Multiple inheritance is also used when code, and not just an interface, must be inherited from more than one class. Often this code is a feature that is "mixed in" to another class. For example, a stream class may be extended with features such as compression or encryption. Less obvious features include object locking and logging capabilities.[2] The class implementing the feature (a "mixin"), and its methods either replace or add functionality to the code in the parent class. For example, the `read` method in the `EncryptedStream` class would read from the stream by calling the parent's `read` method, and then decrypt the data. The `write` method would encrypt the data and then pass it on to the `write` method in the parent stream class.

```
class EncryptedStream /* E */
   inherits Stream;
   method read() =
      data = super:read();
      /* decrypt data */
      /* return data */
   method write(data) =
      /* encrypt data */
      super:write(data);
end;
```




```
class CompressedStream /* C */
   /* Similar to EncryptedStream */
   inherits Stream;
   method read() = ...
   method write(data) = ...
end;
```

```
class UUEncodedStream /* U */
   /* Similar to EncryptedStream */
   inherits Stream;
   method read() = ...
   method write(data) = ...
end;
```

To use more than one feature, multiple inheritance can be used to inherit from the classes that have implemented the feature:

[1] A forwarding class may perform additional functions, such as sending messages to another address space, providing protection for the original object, or handling reference counting. See the *Proxy* pattern in [9] for details.

[2] For example, a mixin could override each method to write information to a log file and then call the original method. This mixin could be added during the debugging process to and removed once the debugging phase is complete.

```
class EncryptedCompressedStream / * EC */
  inherits EncryptedStream;
  inherits CompressedStream;
end;
```

The class hierarchy above shows the "diamond" shape that results. The class on top (Stream in this case) is inherited by the class on the bottom (`EncryptedCompressedStream`) through more than one path.

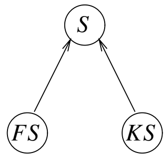
It is not clear, however, that `EncryptedCompressedStream` works as expected. There is no way for the `Compressed.read` method to know that it should call its *sibling*, `Encrypted.read`, rather than its parent, `Stream.read`. Some languages provide a "super" call that implements the required sibling call in this case and the parent call in other cases. If this feature is not available (e.g., in C++), the classes must be designed to implement the `read` method with a helper method. The helper method performs the actual work, and the `read` method calls the helper method(s) in the appropriate order, as described in 2.2.2. For `EncryptedCompressedStream`, the `read` method would call the helper methods in `Stream`, `CompressedStream`, and then `EncryptedStream`. This new `read` method cannot re-use the inherited `read`s to do part of its work. This leads to a great deal of code duplication: every method that is affected by the mixin must be split into a helper method and a control method, the control methods must be reimplemented (not inherited) in every class, and the subclasses must be updated any time one of the mixin control methods change (since the code is duplicated and not inherited).

Even if the language provides a suitable "super" call mechanism, the methods may not be called in the desired order. Some languages, such as CLOS and Dylan, provide automatic linearization; in those, the programmer must be aware of the algorithm used for linearization so that he or she can ensure that compression occurs before encryption, and encryption occurs before uuencoding. Other languages, such as C++, do not provide linearization, and instead, each method that requires linearization has to be written again in the class that inherited several mixins. Languages with multiple inheritance offer many different ways of handling the situtation with multiple mixins, but they either require extra code to be written for every combination of mixins or are error-prone in the sense that changes to the class hierarchy may lead to unexpected changes in behavior.

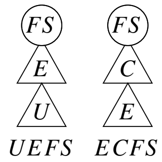
With phunctors, each mixin becomes a phunctor that can be applied to one of the non-mixin classes. In our example, `Encrypt`, `Compress`, and `UUEncode` would be phunctors that could be applied to any `Stream` class:

```
type AnyStreamType = {| with:
        method read: unit → array of α;
        method write: array of α → unit;
        method eof: unit → boolean |}
```

```
phunctor Encrypt( S:AnyStreamType ) /* E */
   method read() =
        data = super:read();
        /* decrypt data */
        /* return data */
   method write(data) =
        /* encrypt data */
        super:write(data);
end;
```



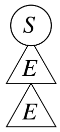

```
phunctor Compress( S:AnyStreamType ) /* C */
   method read() = ...
   method write(data) = ...
end;

phunctor UUEncode( S:AnyStreamType ) /* U */
   method read() = ...
   method write(data) = ...
end;
```




```
class FileStream; /* FS */
class KeyboardStream; /* KS */
class UUEncodedEncryptedFStream = UUEncode(Encrypt(FileStream)); /* UEFS */
class EncryptedCompressedFStream = Encrypt(Compress(FileStream)); /* ECFS */
val ueKStream = UUEncode(Encrypt(KeyboardStream)).new()
```

Instead of writing code for each combination of mixins, the phunctor approach allows us to name each combination without writing any code. It also allows us to use a combination of mixins without naming it, such as for `ueKStream` above. Phunctors give us control over the order in which the mixins are assembled. We can make sure that the user uuencodes an encrypted version of a compressed stream, rather than a compressed encrypted uuencoded stream.[3] In addition, phunctors are not limited to being applied once. A stream could be encrypted *twice* by using `Encrypt(Encrypt(Stream))`. This would not be possible with most multiple inheritance systems, because classes (such as `EncryptedStream`) cannot have copies of themselves as siblings.

```
class DoubleCrypt =
    Encrypt(Encrypt(Stream));
```




The advantages of phunctors over multiple inheritance in this example are the ability to control the exact order of feature addition, the ability to use a combination of features without writing any additional code, and the ability to apply a feature more than once.

In general, whenever features $A_1, A_2, \ldots, A_k$ are needed to add functionality to class that inherits $B$, they can be implemented as phunctors that take as input any class that satisfies the interface of class $B$. Any class $E$ that had $C, A_{i_1}, A_{i_2}, \ldots, A_{i_n}$ as parents and added new fields

[3]UUencoding is needed when we need to transmit a file in ASCII. Encrypting an ASCII stream would lose its 7-bit property, so it is pointless to encrypt after uuencoding. Similarly, compression can be used well when there are patterns in the file. Encrypting the file destroys any patterns, so compression should be done before encryption.

and methods is first split into two classes $E$ and $E'$. Class $E'$ inherits from $C, A_{i_1}, A_{i_2}, \ldots, A_{i_n}$ but adds no fields or methods. Class $E$ inherits from $E'$ and adds the fields and methods. With phunctors, class $E'$ is replaced by the phunctor application $A_{i_1}(A_{i_2}(\ldots A_{i_n}(C)\ldots))$, and class $E$ inherits from it.

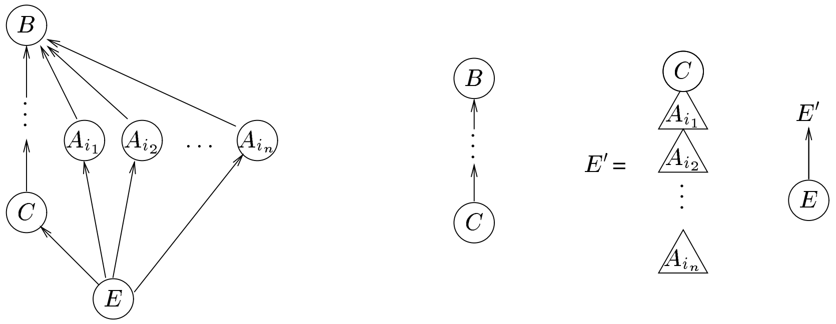



Figure 3: A multiple inheritance hierarchy with mixins can be transformed into one using mixin phunctors

The *Decorator* design pattern described in [9] is similar to mixins, but applies to features that can be added or removed dynamically. Decorators can be created with phunctors $A_1, A_2, \ldots, A_k$ and the forwarding class $F$ described in section 3.2.1. A decorator for a single feature $i$ would be $A_i(F)$. With phunctors, one can also combine multiple decorators statically to produce a new decorator, by applying a phunctor $A_j$ to a decorator class. For example, $A_j(A_i(F))$ forms a decorator from the composition of $A_i$ and $A_j$. The use of forwarding classes makes it possible to produce both static mixins and dynamic decorators from the same phunctors.

#### 3.2.3 Arbitrary Hierarchies

The examples above show how common contexts using multiple inheritance can be implemented using only phunctors and single inheritance (assuming that interface subtyping is available in the language). In general, we believe that the most fruitful approach is to design class hierarchies with phunctors in mind rather than trying to convert an existing multiple inheritance hierarchy into the phunctor form. The main reason is that the order of phunctor application should be determined by design requirements (e.g., compression should be applied before encryption) rather than an algorithm that considers only the listing order of base classes.

However, if the need should arise, an arbitrary class hierarchy using multiple inheritance can be converted into a sequence of phunctor applications using a monotonic superclass linearization algorithm such Dylan's proposed C3 algorithm [3]. For any leaf class $A$, all of its ancestors $A_1, \ldots, A_n$ are linearized from the most specific to the least specific. Then for all $i$, class $A_i$ is turned into a phunctor whose parameter constraint is inferred as specified in section 4.1.1 based on how the inherited methods are used in the class body. Finally, $A$ is defined to be the result of a sequence of phunctor application $A_1(A_2(\ldots A_n(E)\ldots))$ where $E$ is the empty class or the root of the class hierarchy.

# 4 Type checking phunctors

## 4.1 Constraint on phunctor parameter

The constraint on the phunctor parameter is either a list of typed method signatures (the `with:` clause), or a class name. If the latter form is used, parameterized inheritance becomes equivalent to conventional single inheritance. Notice that in the streams example in section 3.2.2, `AnyStreamType` is just a name for the signature list and not a specific class name.

### 4.1.1 List of method signatures

If the phunctor is intended to be applied to more than one class, the constraint on the phunctor parameter should be specified as a list of typed method signatures

$$\{\!|\ m_1 : \tau_{11} * \ldots * \tau_{1n_1} \rightarrow \delta_1 \ \ldots\ m_k : \tau_{k1} * \ldots * \tau_{kn_k} \rightarrow \delta_k\ |\!\}$$

To ensure that all method calls can be resolved correctly, the constraint must include all of the parameter's methods that are called in the phunctor's body. More precisely, suppose that $P$ is the parameter, and a method $P.m()$ is called (perhaps more than once) in the phunctor's body:

$$\begin{array}{rcl} y_1 & = & P.m(x_{11}, \ldots, x_{1n}) \\ & \ldots & \\ y_r & = & P.m(x_{r1}, \ldots, x_{rn}) \end{array}$$

Let $\tau_{ij}$ be the type of the argument $x_{ij}$, and $\delta_i$ the type of the result $y_i$. Then the parameter constraint must include the following signature:

$$m : \hat{\tau}_1 * \ldots * \hat{\tau}_n \rightarrow \hat{\delta}$$

where for all call sites $i$ and parameters $j$, $\tau_{ij}$ <: $\hat{\tau}_j$, and $\hat{\delta}$ <: $\delta_i$ (<: denotes the subtyping relation). This is the usual subtyping rule for function types: the argument types in the signature must be at least as general as the actual arguments, and the return type must be at least as specific as the actual return type. This part of the constraint can be inferred automatically by analyzing all method invocations in the phunctor's body.

Of course, the programmer who implements the phunctor may require that the parameter have certain methods even though they are not called in the phunctor's body. The appropriate signatures can be added to the constraint list by hand.

An example of phunctor parameter constraint parameter can be found in section 3.2.2. The parameter of `UUEncode` phunctor is constrained by `AnyStreamType`. Any class that has a `read()` method can be used to derive a `UUEncoded` stream from it.

### 4.1.2 Class name

When specifying parameter constraint, the programmer has the option of giving a specific class name instead of a method signature list. This means that the phunctor can be applied to only one class, and parameterized inheritance becomes equivalent to conventional single inheritance.

### 4.2 Type checking phunctor application

The type checker rejects any attempt to create a new class by applying a phunctor to a class whose interface (i.e., list of typed method signatures) does not conform to the phunctor's constraint.

When phunctor $P$ is applied to class $C$, for each method signature in $P$'s parameter constraint

$$m_i : \tau_{i1} * \ldots * \tau_{in_i} \rightarrow \delta_i$$

$C$'s interface must contain a method with the same name whose type is a supertype of the one listed in the constraint. More formally, $C$'s interface must include

$$m_i : \tilde{\tau}_1 * \ldots * \tilde{\tau}_{n_i} \rightarrow \tilde{\delta}$$

where for all argument types $j \in 1..n_i$, $\tau_{ij} <: \tilde{\tau}_j$, and the result type $\tilde{\delta} <: \delta_i$. This is the standard function subtyping rule.

If $P$ defines a new method $m$, and $C$ defines a method with the same name, then it is assumed that $P.m()$ is a redefinition of $C.m()$ ($P.m()$ overrides $C.m()$ in C++ terminology). In this case, it is necessary to check that $C.m()$'s type is the supertype of $P.m()$'s type. This is done using the same subtyping algorithm that is used for all other signatures.

## 5 Advantages of parameterized inheritance

- Conventional inheritance is a special case of parameterized inheritance arising when phunctor parameter is fixed (section 4.1.2). Essentially, phunctors provide a unified framework for single, multiple, and parameterized inheritance. This is similar to the result obtained for mixin-based inheritance by Bracha and Cook [4].
- Parameterized inheritance may be used to linearize multiple inheritance in a disciplined way. In particular, parameterized inheritance allows one to add new behavior and/or interface to multiple classes. Instead of deriving a class from several base classes, the programmer applies phunctors in sequence to the main base class. The order is made explicit by the programmer as opposed to the linearization algorithm that may silently yield unexpected results.
- With parameterized inheritance, class extension is symmetric. A class's parent may be changed (by using a different phunctor parameter) as easily as a new child can be derived. This means that there is no dependence on the base class unless necessary. Any class supporting a given interface can be used as the base class. Therefore, base class implementation can be changed without affecting the derived class.
- Parameterized inheritance provides a limited form of metaclasses by allowing one to define inheritance operations over a set of classes meeting a certain constraint rather than a single class. This is mainly useful for creational design patterns such as Abstract Factory and Factory Method [9], where it is desirable that the code for object construction and instantiation is independent from the object's actual class. See section 8.3 for more details.
- One can mix in the same behavior twice since there is no clash between private names defined in the phunctor (e.g., double encryption is possible).

# 6 Similar constructs in existing languages

Phunctors are similar to some language constructs from the existing languages.

## 6.1 C++ templates

Phunctors are used for parameterizing inheritance, while C++ templates are used for general class parameterization. Phunctors are similar to templates, but having a more specialized purpose, phunctors are implemented differently. Phunctors produce only one copy of the code; templates produce a new copy of the code for each parameter. In section 7 we discuss the implementation issues involving phunctors. In addition to implementation, phunctors and templates have differences visible to the programmer. Unlike templates, phunctor parameters can carry a constraint that describes which classes can be used as a parameter. Also, phunctors can be applied more than once, while templates can be applied more than once only if they add no fields.

We use phunctors combined with multiple interfaces and single inheritance as an alternative to multiple inheritance. Since C++ does not support multiple interfaces, we cannot use templates instead of multiple inheritance. To see this, consider the `StreamableHTMLEditor` example from section 3.2.1. It inherits from `HTMLEditor` but not from `Stream`; the `Stream` functionality is added by the phunctor but not inherited. Since in C++ only inheritance results in a subtyping relationship, `StreamableHTMLEditor` objects cannot be used as `Streams`. For templates to be used instead of multiple inheritance, support for multiple interfaces must be added to C++.

## 6.2 ML functors

Phunctors are similar to ML functors. A phunctor takes a class and uses it to define a new class. A functor takes a module and uses it to define a new module. Unlike ML functors, however, phunctors are not forgetful. The derived class does inherit the methods that are hidden by the parameter's constraint. This can be contrasted with ML, where if the signature of the functor's parameter does not mention a certain function, this function is not exported into the new module created by the functor.

## 6.3 CLOS mixins

Mixins in CLOS are a programming idiom rather than a language construct. As mentioned before, a mixin is a class implementing a single property or a set of closely related properties. By deriving a class from the main base class (e.g., `Window`), and one or more mixins (e.g., `Scrollable` or `Streamable`), the programmer can assemble properties in the order required by the system design specification.

Unlike phunctors, CLOS mixins are unstructured and loosely type checked. With mixins, the programmer cannot indicate to which classes the mixin may or may not be added. Even though phunctors derive their inspiration from mixins, the concept of constrained parameterization is essential for phunctors since it enables the programmer to specify that the phunctor may be applied only to certain classes. We believe that this approach combined with static type checking leads to more robust designs and better compile-time diagnostics.

# 7 Implementation issues

The main issue when compiling a phunctor definition is that no assumptions can be made about the layout of the class to which the phunctor will be applied, even though some information about its interface may be available from the parameter constraint. This is similar to compiling a conventional class definition with an unknown base class.

This does not present any problems for Smalltalk-like languages where method lookup is performed at run-time (possibly with caching to improve performance). Each class has a pointer to its superclass which is used when searching for a method at run-time. If this is the case, a phunctor simply defines a class whose superclass pointer is empty. All of the phunctor's methods are compiled once, and for each phunctor application a copy of the superclass pointer is created and made to point to the phunctor's parameter. This assumes that instance variables are looked up at run-time as well. If the offsets of the inherited instance variables have to be known at compile-time, then the situation becomes similar to C++.

C++ optimizes method lookup by using virtual method tables. Because the exact base class is unknown, the compiler cannot determine the offsets of the inherited data members, nor the offsets of the functions in the virtual method table. The simple solution is to take the same approach as C++ does with templates, and compile a separate copy of phunctor code each time the phunctor is applied to a class. This will result in as many copies of phunctor code as there are phunctor applications in the program.

Another solution is to put all unknown offsets in a lookup table (the *virtual offset table*). When compiling a phunctor method that accesses inherited methods and/or data members, the compiler generates code that looks up the appropriate offset values in the virtual offset table. This is different from conventional C++ where all offsets are known at compile-time. When phunctor $P$ is applied to class $C$, a new copy of $P$'s virtual offset table is created and the offsets of $C$'s data members and methods are assigned according to $C$'s layout. This results in space savings since only one compiled copy of phunctor methods is needed. The tradeoff is an extra level of indirection when accessing inherited methods and data members, and space needed for one copy of virtual offset table per each phunctor application.

# 8 Relevant design patterns

This section describes how phunctors can be used for implementing some of the design patterns from [9].

## 8.1 Adapter

The Adapter pattern describes both static and dynamic mechanisms for adding interface to an existing class. As described in section 3.2.1, static adapters can be implemented with phunctors and forwarding. The phunctor takes any class that supports the adaptee interface as its parameter, and derives a new class from it that supports the target interface, forwarding all method calls to the base (adaptee) class.

The advantage of this approach is that a single phunctor can be used to provide new interface to multiple classes. For example, a single phunctor can be used to add `Stream` interface to all classes in the `Editor` hierarchy. In general, suppose that a class hierarchy contains $m$ classes supporting the basic set of methods $S_m$. By adding $n$ phunctors that implement $n$ interfaces in terms of $S_m$, it is possible to obtain $m \cdot n$ different interface-implementation combinations. With multiple inheritance, a separate class would have been needed for each interface-implementation combination.

## 8.2 Decorator

The Decorator pattern describes a *dynamic* mechanism for adding functionality to an object. For example, a `Window` object can be decorated with `ScrollDecorator`, `BorderDecorator`, etc. A decorator object contains a pointer to a `Window` object and supports full `Window` interface. All calls to `Window` methods are forwarded to the `Window` object, but the decorator may add some behavior of its own. For example, `BorderDecorator.draw()` draws a border around the window before calling the `draw()` method of the `Window` object.

Decorator's main advantage over multiple inheritance is flexibility since decorators can be added and removed at run-time. The disadvantages as listed in [9] are:

Tiny objects. An object with $n$ pieces of additional functionality is implemented with $n+1$ small objects. Since all method calls are forwarded to the underlying decorated object, there are as many layers of indirection for each call as there are decorators.

Object identity. If Decorator is used, a component without decorator is not equal to the decorated component, even though they are interchangeable from the user's viewpoint.

Phunctors are a *static* mechanism for adding functionality to a class. It preserves most of the Decorator's advantages: it is more flexible and requires fewer classes than conventional inheritance, classes are simpler and smaller, and a property can be added twice to the same class. Phunctors do not suffer from either of the disadvantages listed above.

Phunctors' main disadvantage is that when they are used, the association of the decorator with the decorated class is fixed at compile-time (phunctor creates a new class that implements both the original class and the additional behavior). Yet, the phunctor approach is more flexible than conventional inheritance since it is easier to add the same functionality to multiple classes when developing code. The main advantage of phunctors is that there is no need for method call forwarding at run-time. The methods of the decorated class are called directly since the phunctor inherits them, so they are as efficient as the native methods. If needed, decorators can be implemented with phunctors are described in section 3.2.2.

## 8.3 Creational patterns

As mentioned before, phunctors are useful for creational patterns since they allow object instantiation code to be independent of the particular class. For example, a phunctor can take any class that has a constructor with a certain name, and derive a Singleton (a class that has no more than one instance) from it.

The Abstract Factory pattern provides an interface for creating families of objects without specifying their concrete classes. For each family, a concrete factory implements the abstract interface to create concrete objects. Abstract Factory can be implemented as a phunctor that takes a concrete factory as a parameter.

This idea is also applicable to the Builder pattern, which separates the construction of a complex object from its representation so that the same construction process can create different representations. Builder can be a phunctor that takes a concrete factory as a parameter, and invokes operations on it to create parts of the object and assemble them in a certain order.

### 8.4 Other patterns

The Bridge pattern decouples the abstraction and the implementation so that the two can vary independently. Phunctors provide some of the advantages of Bridge, such as separate hierarchies for the interface and the implementation, and "cross-product" classes (i.e., all possible interface-implementation combinations). The difference is that phunctors are static, and therefore bind interface and implementation at compile-time.

## 9 Conclusion

We have demonstrated the problems caused by multiple inheritance, and evaluated parameterized single inheritance as an alternative. An appropriate use of phunctors (parameterized class definitions) can replace multiple inheritance in the presence of multiple interface subtyping. We have shown how interface and implementation reuse can be achieved with phunctors instead of multiple inheritance, often resulting in greater flexibility.

Essentially, phunctors provide a structured way to linearize multiple inheritance hierarchies. Signature constraints on phunctor parameters help in specifying design requirements and static error checking. Because the use of parameterization in phunctors is limited to base classes, phunctors are susceptible to efficient implementation in which only one compiled copy of phunctor code is needed for multiple instantiations.

We intend to continue exploring parameterized inheritance as a tool for class hierarchy organization in the context of OBSTACL. We are also interested in applying the phunctor concept to other object-oriented languages that currently lack multiple inheritance but provide support for parameterization.

## References


[1] Arnold, K., and Gosling, J. (1996). *The Java Programming Language.* Addison-Wesley.

[2] Bank, J., Liskov, B., and Myers, A. (1997). *Parameterized Types and Java.* POPL '97.

[3] Barrett, K., Cassels, B., Haahr, P., et al. (1996). *A Monotonic Superclass Linearization for Dylan.* OOPSLA '96.

[4] Bracha, G., and Cook., W. (1990). *Mixin-Based Inheritance.* OOPSLA '90.

[5] Cook, W. R. (1992). *Interfaces and Specification for the Smalltalk-80 Collection Classes.* OOPSLA '92.

[6] Coplien, J. O. (1992). *Advanced C++ Programming Styles and Idioms.* Addison-Wesley.

[7] Dahl, O-J., Myrhaug, B., and Nygaard, K. (1970). *SIMULA Common Base Language.* Norwegian Computing Center S-22.

[8] Ellis, M., and Stroustrup, B. (1990). *The Annotated C++ Reference Manual.* Addison-Wesley.

[9] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns.* Addison-Wesley.

[10] Ingalls, D. H. (1978). *The Smalltalk-76 Programming System Design and Implementation.* POPL '78.

[11] Meyer, B. (1988). *Object-Oriented Software Construction*. Prentice Hall.

[12] Moon, D. (1986). *Object-Oriented Programming with Flavors*. OOPSLA '86.

[13] Odersky, M., and Wadler, P. (1997). *Pizza into Java: Translating Theory into Practice.* POPL '97.

[14] Snyder, A. (1986). *Encapsulation and Inheritance in Object-Oriented Programming Languages*. OOPSLA '86.

[15] Steele, G. (1990). *Common Lisp: the Language* (2nd ed.). Digital Press.

[16] Stroustrup, B. (1992). *The C++ Programming Language* (2nd ed.). Addison-Wesley.

[17] Waldo, J. (1991). *Controversy: The Case for Multiple Inheritance in C++*. USENIX Spring '91.