

Base Class Injection

Douglas Gregor, Sibylle Schupp, and David Musser

Computer Science Department
Rensselaer Polytechnic Institute
{gregod,schupp,musser}@cs.rpi.edu

Abstract. Class hierarchies, though theoretically reusable, have generally not seen much practical reuse in applications, due in part to the inflexibility of the inheritance relationship. We present a technique, base class injection, that allows the creation of generative class hierarchies that may be adapted by adding new methods, data members, and ancestor classes without modification to the class library code; an implementation of this technique in the C++ language is given.

1 Introduction

Inheritance is a powerful yet inflexible tool for program construction in object-oriented programming languages [22]. Inheritance is intended to convey the hierarchical structure of data and code and to foster a large degree of code reuse. While proper design of inheritance hierarchies has been successful in conveying the structure of data, the much-touted reusability of class hierarchies has not occurred [7, 9, 16].

The limited reuse of class hierarchies is due to many factors, one of which is the inextensibility of inheritance. The direct ancestors of a given class are chosen at the design time of that class, and are not changed barring a source code rewrite. While this is often reasonable—a class’s parents define much of what that class is—it is inflexible, in that certain relationships may exist between two independently developed class hierarchies that cannot be expressed through inheritance without modifying either class hierarchy. Essentially, the limitation to reusability of class hierarchies is that they may only be extended by creating a new leaf in the inheritance graph.

To foster reuse of class hierarchies, we must make them extensible beyond the limited benefit of creating a new leaf in the inheritance graph and instead allow the class hierarchy to be customized at all levels. Extensibility in this case means that we must be able to express relationships between classes and hierarchies that were unknown or nonexistent at class hierarchy design time, but come into existence because of a particular usage of the class hierarchy. Such relationships occur when a class hierarchy is adapted to a particular program environment that requires certain global properties such as persistence, reflection, or garbage collection; they may also occur when two independently developed libraries are found to have common properties that could be exploited if only the libraries shared the same interface.

We will present a motivating example in Section 2 along with a sketch of our solution using a technique we call *base class injection*. Section 3 will highlight the requirements of a base class injection system, along with the basic language requirements needed to implement such a scheme, and Section 4 details the implementation of base class injection in Standard C++. Complete source for a base class injection library is listed in the appendix and we are planning a submission to Boost [11] for formal review.

2 Adapting to New Interfaces

We start our example considering an application developed with a hypothetical graphical user interface widget library named GOOEY. The GOOEY library handles the usual widgets, including buttons, menus, and text entry fields. For reference, we will assume it is implemented as an object-oriented hierarchy similar to that in Figure 1.

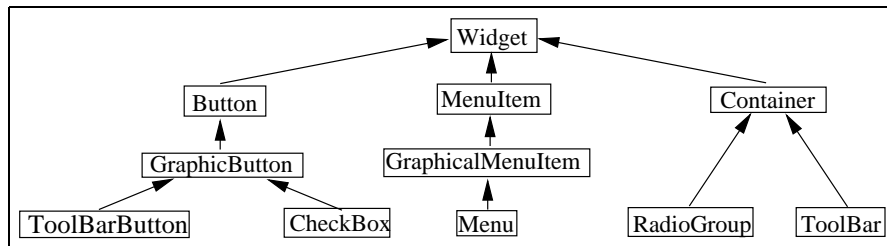


Fig. 1. Inheritance hierarchy for the hypothetical widget library, GOOEY

Late in the development of our application, it is determined that our customers require an interface that also suits visually impaired users. While much of the information required for an audible interface exists in the instances of widgets used in the user interface, the information is not accessible in the form necessary for an audible interface. For instance, the label of a **Button** conveys the functionality of the button and could be used in conjunction with a text-to-speech synthesizer and, similarly, a **RadioGroup** may have a caption describing its function and a **CheckBox** would convey both its label and whether it is selected or not. Had we designed the GOOEY widgets with an audible interface in mind, we would have provided a common interface to the audible description of each widget. Each widget would provide an implementation appropriate for the style of widget: a **Button** would convey the functionality of a button press, and a **CheckBox** would customize a **Button** to include whether it is selected or not.

Given that we are unable to directly modify the GOOEY library, how can we include the audible description interface in each widget of the GOOEY library, customizing the access methods for any widget? One common technique is to wrap the functionality of the GOOEY library in a set of wrapper classes that

each implement the audible description interface. This method is tedious and error-prone and is also brittle: changes in the GOOEY library will have to be reflected in the wrapper classes. We present a more natural and resilient solution based on base class injection.

With base class injection, we create an audible description interface class and then inject it into the GOOEY class hierarchy as a base class of `Widget`. Then, we use base class injection to override the methods of the audible description interface in subclasses of `Widget` that require a specific audible interface. Our application remains unchanged by the addition of this code, but we now possess the capability to access the audible listener interface of any widget. Figure 2 illustrates a portion of the resulting GOOEY hierarchy with the injection of the `AudibleDescription` interface at the root, with overrides injected for the `Button` and `CheckBox` subclasses. We have essentially customized the entire GOOEY class hierarchy without requiring a change to the GOOEY library source code by injecting an additional interface (and its implementation) into the preexisting hierarchy. Note also that the injections are performed without regard to the intervening class hierarchy, so this solution does not suffer from the same brittleness experienced with wrapper approaches. Furthermore, additional interfaces—such as one for handheld computers—could be developed independently of the audible interface, and at compile time the appropriate set of interfaces could be selected, essentially giving the developer the power to generate a class hierarchy specific to the needs of the given application but without requiring hooks into the actual class hierarchy source code.

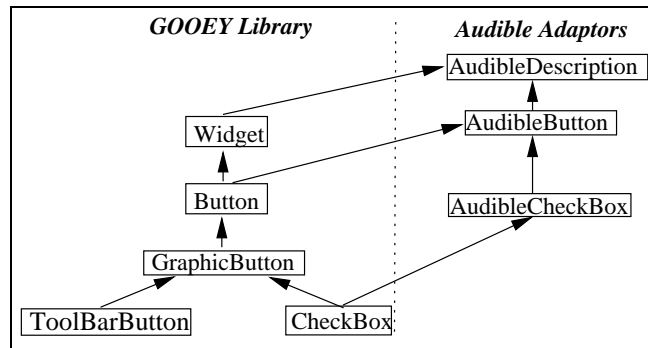


Fig. 2. The GOOEY class hierarchy with the `AudibleDescription` interface injected at the root class `Widget` and later reimplemented for the `Button` and `CheckBox` classes.

3 Base Class Injection

Base class injection may be characterized by the extension and augmentation of existing class libraries at any point in the class hierarchy without the need for

rewriting library source code. We now enumerate the exact requirements of a base class injection implementation to better understand its capabilities:

1. It should allow arbitrary classes to be prepared to receive injections. Once prepared any class can be injected into that class without modifying the source code for the class receiving the injection.
2. It should not change the type of the class receiving the injection, so that no code modifications are necessary after the injection.
3. Common object-oriented features, such as virtual method invocation, encapsulation, and polymorphism, should be available with respect to the injected base class and its members.
4. It should minimize additional run-time overhead.

We see from our example in the previous section that all of the properties are necessary to augment the GOOEY library with an interface for the visually impaired. We inject an interface class `AudibleDescription` not known at the time the GOOEY class library was designed (Property #1), without changing the types of any of the classes in the GOOEY library so that only recompilation is required to access the new functionality (#2); finally, we use virtual function overriding to customize the implementation of the `AudibleDescription` interface for different classes in the GOOEY class library (#3) and, although it was not explicitly stated, we assume polymorphic behavior from the augmented class hierarchy, that is, we can rely on the ability of any `Widget` to be viewed as an implementation of the `AudibleDescription` interface. The final requirement (#4) is one of efficiency—for an organizational technique to be adopted it must clearly have little negative impact on the run-time efficiency of the system.

It is beneficial to understand the key language features required for an implementation of base class injection. It is clear that we are considering object-oriented or hybrid languages with inheritance. We restrict our discussion to statically typed languages; similar techniques for the dynamically typed languages Ruby and Smalltalk are discussed in Section 5. We now isolate the language requirements for base class injection:

1. It must support multiple inheritance or, more generally, allow for a given class to extend two classes neither one of which is derived from the other.
2. It must support a method of specifying the base classes to inject for a given class, and statically collect this list of base classes.
3. It must support a method of delaying the full definition of a class until all of its base classes are known.

The first requirement is obvious; the second essentially requires the ability to specify a list of base classes that a given class must inherit. For a statically typed language, this list must be built statically. The third requirement is the most interesting; it requires that a class be developed with no knowledge of the base classes that may later be injected, but the language must not prohibit references to the class prior to its full definition. Forward referencing minimally fits this requirement, though it places an unfortunate burden on the library developer

and user to provide declarations and definitions in entirely different areas, with base class injections declared after forward declarations, but before definitions.

It is worth noting that such a system could be constructed from a very type-strict and non-generic language such as Java [4] only with the assistance of external programs. For instance, a source code preprocessor or source-to-source translator that scans an auxiliary database and adds additional `implements` clauses and methods could be used to satisfy the second and third language requirements (Java interfaces support multiple inheritance).

We next illustrate that the C++ language [3] already meets or exceeds these requirements and stress that no extensions to the language are necessary to support base class injection.

4 Implementation

Our C++ implementation relies on some advanced features of the C++ language and on techniques well-known in the C++ community. Most prominently, we will use C++ templates to meet the second and third language requirements for base class injection, using partial specialization, traits [5], and template metaprogramming [10, 21]. Templates will also allow us to delay the actual definition of a class until it is absolutely necessary, thereby allowing the user a large degree of freedom in the timing of class injections.

We will explore the implementation of base class injection first from the user's perspective, covering injection of base classes and overriding virtual methods in injected classes in Sections 4.1 and 4.2, then move to the interface used by the library designer to build a class hierarchy that can accept injections in Section 4.3. Finally, we explain the implementation of a base class injection library in Section 4.4.

To avoid any ambiguity regarding the actual class being injected and the class receiving the injection, we will call the class receiving the injection the *receiver*. The class that is being injected into the receiver will be called the *injected base class*. Therefore, in our example from Figure 2, `Widget`, `Button`, and `CheckBox` are all receivers, while `AudibleDescription`, `AudibleButton`, and `AudibleCheckBox` are all injected base classes.

4.1 Injecting a Base Class into a Receiver

Injecting an arbitrary base class into a receiver is performed by specializing one of two template classes, `base_class` or `virtual_base_class`. Figure 3 illustrates the injection of the base classes `AudibleDescription`, `AudibleButton`, and `AudibleCheckBox` into the receiver classes `Widget`, `Button`, and `CheckBox`, respectively. The `Tag` template parameter is essential to the C++ implementation of base class injection and is explained in Section 4.3.

From Figure 3, we see that `virtual_base_class` is parameterized by two template parameters. The first parameter is the name of the receiver, and the second is an index value. For the first injection into a given receiver, this value

```

template<typename Tag> struct virtual_base_class<Widget<Tag>, 0>
{ typedef AudibleDescription base; };

template<typename Tag> struct virtual_base_class<Button<Tag>, 0>
{ typedef AudibleButton base; };

template<typename Tag> struct virtual_base_class<CheckBox<Tag>, 0>
{ typedef AudibleCheckBox base; };

```

Fig. 3. Injecting the base classes `AudibleDescription`, `AudibleButton`, and `AudibleCheckBox` into the GOOEY library classes `Widget`, `Button`, and `CheckBox`

should be zero; for the second injection into that receiver, it should be one, and so on (each receiver has its own counter). The member type `base` declares the base class to be injected. The template classes `base_class` and `virtual_base_class` differ only in the method used to derive from the injected base class—`base_class` defines base classes that will be injected into the receiver via nonvirtual public inheritance, whereas `virtual_base_class` defines base classes injected via virtual public inheritance. For those unfamiliar with C++ terminology, a given class A that is inherited nonvirtually a certain number of times in a hierarchy will show up that many times in a hierarchy. However, if all inheritance of A in a hierarchy is virtual, it will appear only once. We will use this powerful feature later when describing the overriding of virtual functions introduced through base class injection in Section 4.2.

4.2 Overriding Virtual Function Injections

Our injected base classes may declare virtual functions and it is essential that we be able to override these virtual functions at a later point in the hierarchy. Figure 4 illustrates the definition of the `AudibleDescription` class and its descendants, including the virtual function `getAudioText` that is overridden in each descendant.

The only point of interest in the overriding of injected virtual functions is the `virtual` keyword that specifies virtual inheritance of the `AudibleDescription` and `AudibleButton` classes. Revisiting Figure 2, we see that we had assumed virtual inheritance of the injected classes in the construction of our inheritance lattice—without it, both `AudibleButton` and `AudibleCheckBox` would have their own distinct copies of the `AudibleDescription` interface, causing conflicts during name resolution on the `AudibleDescription` interface.

4.3 Preparing the Receiver

Preparing a receiver class to accept injections requires only a few small changes to the definition of the receiver class. The receiver will publicly derive from a class `bases` that represents the unknown set of base classes, specializing it based on the name of the receiver, as is illustrated in Figure 5. This is reminiscent of

```
class AudibleDescription
{ public: virtual AudioText getAudioText(); };

class AudibleButton : virtual public AudibleDescription
{ public: virtual AudioText getAudioText(); };

class AudibleCheckBox : virtual public AudibleButton
{ public: virtual AudioText getAudioText(); };
```

Fig. 4. Injecting and overriding virtual methods with base class injection

the well-known B & N trick [5] that customizes the ancestors of a class based on the descendant class as a method of supporting multiple implementations of a common interface through a common base class template while preserving static bindings.

```
template<typename Tag = default_tag>
class Widget : public bases< Widget<Tag> > { /* ... */};
```

Fig. 5. Preparing a receiver for base class injection

The explanation of the `Tag` parameter delves deep into the intricacies of the C++ language. The so-called One Definition Rule [3] states that any entity may only be defined once. A non-template class is defined when its class body is defined, therefore non-template receiver classes give us no chance to inject injectable base classes because they are immediately defined. Template classes, on the other hand, are not fully defined until they are instantiated with a given set of template parameters. By adding the `Tag` template parameter we ensure that our receiver class is a template class and is therefore not fully defined before user code requires a definition. In effect, this allows the receiver class to be written without regard to the base classes that may be injected later. The injected base classes may then be included up until the first concrete usage (i.e., one that requires knowledge of the size of the class or accesses its member variables).

The `Tag` parameter has an additional usage allowing multiple distinct hierarchies in a single translation unit. The tag denotes a specific set of injections, e.g., `Audible` or `Handheld`. Figure 6 illustrates that `Widget` can be injected with either an `AudibleDescription` class or a `HandheldDescription` class, depending on whether the tag is `Audible` or `Handheld`.

4.4 Collecting Injected Base Classes

Moving to the implementation of a base class injection library, we present a technique for collecting the set of injected base classes and deriving from each

```

template<typename Tag> struct virtual_base_class<Widget<Tag>, 0>
{ typedef placeholder_base_class base; };

template<> struct virtual_base_class<Widget<Audible>, 0>
{ typedef AudibleDescription base; };

template<> struct virtual_base_class<Widget<Handheld>, 0>
{ typedef HandheldDescription base; };

template<typename Tag> struct virtual_base_class<Widget<Tag>, 1>
{ typedef GarbageCollected base; };

```

Fig. 6. Using tags to allow multiple versions of a hierarchy to coexist in a translation unit. All `Widgets` will derive from `GarbageCollected`

of them. We use template metaprogramming to traverse the list of injected base classes and create an inheritance chain containing all injected base classes.

We refer again to Figure 3 and note that the second template parameter of `base_class`, the index value, mimics the familiar concept of an array index. Essentially, `base_class` is a compile-time resizeable array: to insert elements, we specialize `base_class` given the next index value; the `base` member type is the value of the element at the given position in the array. This linearization does not hamper base class injection in any way: injected base classes parameterized by their receivers may access extensions made by other injected base classes regardless of base class ordering.

Iteration through such a data structure is clear: starting a counter at zero, we access the `base` type to get the element value and increment the count to access the next element, but how do we know when to terminate iteration? C++ contains the notion of a *primary* template, that is, a template that will be selected if no (partial) specialization matches the given set of arguments. We therefore define a sentinel type `end` and a primary template for our compile-time array `base_class` that contains `end` as its value; then, our iteration construct terminates when it finds that the current element is `end`. Figure 7 illustrates the `end` class, the primary template for `base_class`, and the class `gather_bases` that creates the derivation chain including all injected base classes.

5 Related Work

Our work is most similar to work done by Ossher and Harrison [19] where object systems were extended by combining *base* hierarchies (e.g., the GOOEY library) with *extension* hierarchies (e.g., the audible and handheld hierarchies). Ossher and Harrison discuss extension by adding new superclasses, but opted to use composition operators within a new language because adding new superclasses “changes the structure of the inheritance hierarchy at an internal node, a type of change that is usually poorly supported by object-oriented systems” and which,

```

struct end {};

template<typename Receiver, int Index = 0>
struct base_class { typedef end base; };

template<
    typename Receiver,
    int Index = 0,
    typename Base = typename base_class<Receiver, Index>::base>
struct gather_bases : public Base,
                    public gather_bases<Receiver, Index+1> {};

template<typename Receiver, int Index>
struct gather_bases<Receiver, Index, end> {};

```

Fig. 7. Implementation of a template class to manage the derivation from any number of injected base classes, along with the default definition of the `base_class` class

they assumed, “requires editing the original class.” While this assumption was probably valid for all programming languages (including C++) at the time of writing, our technique demonstrates how modern C++ fully supports such changes in the inheritance hierarchy without source code editing.

Our view of the reusability problems with class hierarchies can be likened to the motivations behind aspect-oriented programming [15, 14] in that class hierarchies that are conceptually distinct often have common roots that express basic system-wide properties. In aspect-oriented programming terminology, these common base classes implement functionality that cross-cuts the hierarchies, and the resulting implementation that joins them tangles the two hierarchies together in an unnatural way. Our technique is supportive of aspect-oriented programming in that it provides another method for keeping conceptually distinct hierarchies distinct in design, and joining them only when necessary by injecting base classes developed separately to form a single hierarchy. In this way, base class injection can be seen as a restricted implementation of the weaving process—one for which, in C++ at least, one does not need any additional tools to implement.

A technique apparently similar to our own is that of mixin classes [8, 13], which are (often abstract) classes that are intended only as base classes to be “mixed in” to a class as it is used. In most languages mixins are limited in that they can only generate a new leaf in the inheritance hierarchy. Therefore the use of mixins requires system-wide changes and, for extensive hierarchy modifications, necessitates something similar to the wrapping approach described above because inheritance relationships between the newly-created leaves will need to be specified manually.

Ruby [1] is a dynamically typed, object-oriented scripting language in which mixins can be injected at any level of a class hierarchy through reopening of class definitions. Thus in Ruby, even though it only has single inheritance among classes, one can use mixins to extend a class hierarchy at all levels without modi-

fyng its source code. Unfortunately, in its unrestricted form in Ruby, the ability to reopen class definitions breaks encapsulation. This problem could perhaps be fixed by disallowing access to data members when classes are reopened, but another drawback to implementing base class injection in this way in Ruby is that only one extension of a given class hierarchy can exist within a single program, whereas our C++ injection library supports coexistence of multiple extensions.

In Smalltalk, which is also dynamically typed, a technique that could be described as a limited form of base class injection was introduced by Kent Beck [6]. This technique allows extending an *instance* of a class without changing the source code of the class. However, it only allows modifying a particular instance of a leaf node.

Work in the refactoring of inheritance hierarchies [18, 12] is rooted in the same realization that at some point in the lifetime of a class hierarchy inheritance becomes too inflexible. Refactoring itself is very different—it relies on source code transformations and changing inheritance relationships among classes already in a given class hierarchy. Since base class injection allows reorganization with minimal code change, it may be preferable to refactoring in some situations, such as independent development of classes by different programmers. Refactoring may, however, help identify classes that would best be abstracted outside of the class hierarchy and later injected.

6 Conclusion

Inheritance is a powerful and essential feature of object-oriented languages, but its ability to foster reuse is degraded by its inflexibility. One limitation to class hierarchy reuse across applications is that extensibility of class hierarchies is confined to extension by new leaves.

Base class injection overcomes this significant limitation to the reusability of class hierarchies by enabling the injection of new base classes and new functionality into a class hierarchy. Such a capability enables class libraries to better adapt to specific uses, whether it be to environmental requirements such as base classes common to all class types (e.g., for reflection, persistence, or synchronization) in a system or through the injection of new interfaces. Base class injection is a tool for increasing the adaptability of a class hierarchy, overcoming limitations in inheritance and resulting in more reusable class hierarchies.

The appendix contains source code for a complete C++ base class injection library. Though the implementation of base class injection requires in-depth knowledge of the C++ language, its application is easily grasped by non-experts. The examples given in Figs. 3 and 5 may be reused substituting only class names and index numbers, making base class injection practical for all programmers.

We have successfully employed the base class injection technique in the Algebra library [20] as part of the Simplicissimus [2] project, where concepts—sets of abstractions—are represented as C++ template classes as a way of expressing concept lattices such as those defined by the Tecton language [17]. Concept refinement is expressed as class inheritance, but some refinement relations are

not known at library design time and are introduced only later in user code—as a counterpart of Tecton lemmas, which establish refinement relations between previously defined concepts. Base class injection allows us to model such lemmas without modifying the library source code.

References

1. Ruby: a gem of a language. <http://www.ruby-lang.org>.
2. Simplicissimus. <http://www.cs.rpi.edu/research/gpg/Simplicissimus>.
3. ANSI-ISO-IEC. *C++ Standard, ISO/IEC 14882:1998*, ANSI standards for information technology edition, 1998.
4. K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, 1998.
5. J. J. Barton and L. R. Nackman. *Scientific and Engineering C++: An introduction with advanced techniques and examples*. Addison-Wesley, 1994.
6. K. Beck. Instance specific behavior: How and why. *Smalltalk Report*, 6(2), 1993.
7. J. M. Bieman and J. X. Zhao. Reuse through inheritance: A quantitative study of C++ software. In *ACM SIGSOFT Symposium on Software Reusability*, pages 47–52, 1995.
8. G. Bracha and W. Cook. Mixin-based inheritance. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Programming*, pages 303–311, Ottawa, Canada, 1990. ACM Press.
9. S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.
10. K. Czarnecki and U. W. Eisenecker. *Generative Programming—Towards a New Paradigm of Software Engineering*. Addison Wesley Longman, 2000.
11. B. Dawes and D. Abrahams. Boost. <http://www.boost.org>.
12. G. Fischer, D. Redmiles, L. Williams, G. I. Puhr, A. Aoki, and K. Nakakoji. Beyond Object-Oriented Technology: Where Current Approaches Fall Short. *Human-Computer Interaction*, 10:79–119, 1995.
13. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 171–183, New York, NY, 1998.
14. G. Kiczales, E. Hisdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *European Conference on Object-Oriented Programming (ECOOP'01)*, 2001.
15. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwi. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP'97)*, 1997.
16. A. Lake and C. Cook. A software complexity metric for C++. Technical Report 92-60-03, Computer Science Dept., Oregon State University, 1992.
17. D. R. Musser. The Tecton Concept Description Language. <http://www.cs.rpi.edu/~musser/gp/tecton/tecton1.ps.gz>, July 1998.
18. W. F. Opdyke and R. J. Johnson. Refactoring: An Aid in Designing Application Frameworks. In *Proceedings of the Symposium on Object-Oriented Programming emphasizing Practical Applications*, pages 145–160, 1990.

19. H. Ossher and W. Harrison. Combination of inheritance hierarchies. In A. Paepcke, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 27, pages 25–40, New York, NY, 1992. ACM Press.
20. S. Schupp, D. P. Gregor, and D. Musser. Algebraic concepts represented in C++. Technical Report TR-00-8, Rensselaer Polytechnic Institute, 2000.
21. T. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4), 1995.
22. P. Wegner. Dimensions of object-based language design. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 22, pages 168–182, New York, NY, 1987. ACM Press.

Injection Library Source

```

struct default_tag {};
struct end {};

// Users add nonvirtual base classes to this list
template<typename T, int Index = 0>
struct base_class { typedef end base; };

// Users add virtual base classes to this list
template<typename T, int Index = 0>
struct virtual_base_class { typedef end base; };

template<typename T,
        int      Index = 0,
        typename Base = typename base_class<T, Index>::base>
struct gather_bases : public Base, public gather_bases<T, Index+1> {};

template<typename T, int Index>
struct gather_bases<T, Index, end> {};

template<typename T,
        int      Index = 0,
        typename Base = typename virtual_base_class<T, Index>::base>
struct gather_virtual_bases :
    virtual public Base, public gather_virtual_bases<T, Index+1> {};

template<typename T, int Index>
struct gather_virtual_bases<T, Index, end> {};

// Receiver classes derive from this
template<typename T>
struct bases : public gather_bases<T>,
              public gather_virtual_bases<T> {};

```