

A Solution to the Constructor-Problem of Mixin-Based Programming in C++

(Presented at the GCSE'2000 Workshop on C++ Template Programming)

Ulrich W. Eisenecker*, Frank Blinn*, and Krzysztof Czarnecki†

*University of Applied Sciences Kaiserslautern, Zweibrücken
Amerikastraße 1
66482 Zweibrücken
Germany
Ulrich.Eisenecker@T-Online.de
Frank.Blinn@student-zw.fh-kl.de

†DaimlerChrysler Research and Technology
Software Engineering Lab
Wilhelm-Runge-Str. 11
89081 Ulm
Germany
czarnecki@acm.org

Abstract. Mixin-Based Programming in C++ is a powerful programming style based on the parameterized inheritance idiom and the composition of C++ templates. Type expressions describing specific inheritance hierarchies can be composed either automatically using generative programming idioms in C++ or manually. Unfortunately, the mixin-based C++ programming techniques published to date do not adequately support optional and alternative mixin classes with constructors expecting varying numbers of arguments, which are common in practice. This is because the varying base class constructors do not provide a uniform interface on which the constructors of the derived classes could rely. This paper discusses several partial solutions to this problem that were proposed to date and presents a new, complete solution. The new solution uses generative programming techniques to automatically generate the appropriate constructors, and this way it avoids the overhead and clumsiness of instantiating composed mixin classes in the client code using the partial solutions. In fact, the new solution allows users to instantiate automatically composed mixin classes with the simplicity of instantiating concrete classes from traditional class hierarchies. Finally, the new solution does not suffer from the scalability problems of the partial solutions.

1 Introduction

A mixin is a fragment of a class in the sense that it is intended to be composed with other classes or mixins. The term *mixin* (or mixin class) was originally introduced in Flavors [Moo86], the predecessor of CLOS [Kee89]. The difference between a regular, stand-alone class such as *Person* and a mixin is that a mixin models some small functionality slice, for example, *printing* or *displaying*, and is not intended to be used “stand-alone,” but rather to be composed with some other class needing this functionality (e.g., *Person*). One possibility to model mixins in OO languages is to use classes and multiple inheritance. In this model, a mixin is represented as a class, which is then referred to as a mixin class, and we derive a composed class from a number of mixin classes using multiple inheritance. Another possibility is to use parameterized inheritance. In this case, we can represent a mixin as a class template derived from its parameter, e.g.:

```
template<class Base>
class Printing : public Base
{...}
```

Indeed, some authors (e.g., [BC90]) define mixins as “abstract subclasses” (i.e., subclasses without a concrete superclass). Mixins based on parameterized inheritance in C++ have been used to implement highly configurable collaboration-based and layered designs (e.g., see [VN96, SB98]).

In this paper, we present a solution to the constructor problem in parameterized-inheritance-based mixin programming in C++. This problem, which was also described by Smaragdakis and Batory in [SB00], is illustrated in Listing 1. Let us take a closer look at the code. The mixin classes `PhoneContact` and `EmailContact` can be easily composed with `Customer`, i.e., `PhoneContact<Customer>` and `EmailContact<Customer>`. In both cases, the constructor interface of the base class (i.e., `Customer`) is known, and the arguments for initializing the base class are passed to the base constructor in the initializer lists of the derived mixin classes (i.e., `PhoneContact` and `EmailContact`). Unfortunately, compositions including both mixins, i.e., `PhoneContact<EmailContact<Customer> >` or `EmailContact<PhoneContact<Customer> >`, although semantically plausible, do not work. This is so because the necessary constructors accepting four arguments and calling the appropriate base class constructor with three arguments are missing.

```
#include <iostream>
using namespace std;

class Customer
{
public:
    Customer(const char* fn,const char* ln)
        :firstname_(fn),lastname_(ln)
    {}

    void print() const
    {
        cout << ' ' << firstname_
            << ' ' << lastname_;
    }

private:
    const char  *firstname_,
                *lastname_;
};

template <class Base>
class PhoneContact: public Base
{
public:
    PhoneContact(const char* fn,const char* ln,const char* pn)
        :Base(fn,ln),phone_(pn)
    {}

    void print() const
    {
        Base::print();
        cout << ' ' << phone_;
    }

private:
    const char  *phone_;
};

template <class Base>
class EmailContact: public Base
{
public:
    EmailContact(const char* fn,const char* ln,const char* e)
        :Base(fn,ln),email_(e)
    {}

    void print() const
    {
        Base::print();
        cout << ' ' << email_;
    }

private:

```

```

    const char *email_;
};

int main()
{
    Customer c1("Teddy","Bear");
    c1.print(); cout << endl;
    PhoneContact<Customer> c2("Rick","Raccoon","050-998877");
    c2.print(); cout << endl;
    EmailContact<Customer> c3("Dick","Deer","dick@deer.com");
    c3.print(); cout << endl;
    // The following composition isn't legal because there
    // is no constructor that takes all four arguments.
    // EmailContact<PhoneContact<Customer> >
    // c4("Eddy","Eagle","049-554433","eddy@eagle.org");
    // c4.print(); cout << endl;
    return 0;
}

```

Listing 1: The constructor problem

There are several partial solutions to the constructor problem. The worst idea is to restrict or change the order in which mixin classes can be composed. Because the previously described example cannot be fixed using this strategy anyway, we will not further explore it. Sections 2.1 through 2.4 describe four partial solutions that are somewhat better, but still suffer from problems such as incurring unnecessary overhead, leading to clumsy client code, and poor scalability when adding new mixin classes. Nevertheless, their study provides the basic insights for understanding the more advanced, complete solution, which we propose in Section 3.

2 Partial Solutions

2.1 Defining an additional mixin class

One aspect of the previously described problem is that a mixin class with an appropriate constructor simply does not exist. A straightforward solution – at least at first glance – would be to simply implement an extra mixin class with the needed constructor using multiple inheritance (Listing 2). Unfortunately, this approach requires major changes to the existing code. First, we have to prepare the inheritance hierarchy for using multiple inheritance. In order to avoid potential duplication of subobjects of the class `Customer` in `PhoneAndEmailContact`, `Customer` has to be changed into a virtual base of `PhoneContact` and `EmailContact`. Additionally, `PhoneAndEmailContact` – as the most derived class – must take care of initializing the virtual base class.

```

#include <iostream>
using namespace std;

// Define a new mixin class with a constructor
// that accepts all arguments.

class Customer
{
public:
    Customer(const char* fn,const char* ln)
        :firstname_(fn),lastname_(ln)
    {}

    void print() const
    {
        cout << firstname_
              << ", "
              << lastname_;
    }

private:
    const char *firstname_,
               *lastname_;
}

```

```

};

// The new mixin class will be defined using
// multiple inheritance. Therefore Base must
// be turned into a virtual base class.
template <class Base>
class PhoneContact: virtual public Base
{
public:
    PhoneContact(const char* fn,const char* ln,const char* pn)
        :Base(fn,ln),phone_(pn)
    {}

    void print() const
    {
        Base::print();
        basicprint();
    }

protected:
    // We need an "inner" print method
    // that prints the PhoneContact-specific
    // information only.
    void basicprint() const
    {
        cout << ' ' << phone_;
    }

private:
    const char *phone_;
};

// Base has to be declared as virtual base class here, too.
template <class Base>
class EmailContact: virtual public Base
{
public:
    EmailContact(const char* fn,const char* ln,const char* e)
        :Base(fn,ln),email_(e)
    {}

    void print() const
    {
        Base::print();
        basicprint();
    }

protected:
    // We need an "inner" print method
    // that prints the EmailContact-specific
    // information only.
    void basicprint() const
    {
        cout << ' ' << email_;
    }

private:
    const char *email_;
};

template <class Base>
class PhoneAndEmailContact :
    public PhoneContact<Base>,
    public EmailContact<Base>
{
public:
    // Because Base is a virtual base class,
    // PhoneAndEmailContact is now responsible for
    // its initialization.
    PhoneAndEmailContact(const char* fn,
                          const char* ln,char* pn,const char* e)
        :PhoneContact<Base>(fn,ln,pn),
        EmailContact<Base>(fn,ln,e),

```

```

        Base(fn,ln)
    {}

    void print() const
    {
        Base::print();
        PhoneContact<Base>::basicprint();
        EmailContact<Base>::basicprint();
    }
};

int main()
{
    Customer c1("Teddy","Bear");
    c1.print(); cout << endl;
    PhoneContact<Customer> c2("Rick","Racoon","050-998877");
    c2.print(); cout << endl;
    EmailContact<Customer> c3("Dick","Deer","dick@deer.com");
    c3.print(); cout << endl;
    PhoneAndEmailContact<Customer>
    c4("Eddy","Eagle","049-554433","eddy@eagle.org");
    c4.print(); cout << endl;
    return 0;
}

```

Listing 2: Defining an additional mixin class

An annoying change is that the `print()` methods of `PhoneContact` and `EmailContact` must be split into `basicprint()` and `print()`. Otherwise, it would be impossible to produce an appropriate output with `PhoneAndEmailContact::print()` (the method splitting technique in the context of virtual base classes is described in [Str97, p. 398]). Imagine what would happen if we introduced another mixin class, for example, `PostalAddress`. This would require adding special mixin classes that combine `PhoneContact` with `PostalAddress`, `EmailContact` with `PostalAddress`, as well as `PhoneContact`, `EmailContact`, and `PostalAddress`. Thus, the number of such combination classes grows exponentially (without considering the composition order).

2.2 Providing a special argument class

The basic idea of this solution is to provide a standardized interface for the constructors of all mixin classes by introducing a special class that wraps the union of all arguments of all mixin class constructors (Listing 3). The technique of bundling arguments in a special argument class was described in [Str94, pp. 156-157].

```

#include <iostream>
using namespace std;

// Define a class that wraps the union of
// all constructor arguments of Customer
// and all derived mixin classes.

// CustomerParameter combines all constructor
// arguments of CustomerParameter and its
// derived mixin classes.
// The default values for the last two arguments
// provide some convenience to the client
// programmer.
struct CustomerParameter
{
    const char
        * fn,
        * ln,
        * pn,
        * e;

    CustomerParameter( const char* fn_,const char*ln_,
                      const char* pn_ = "",const char* e_ = "" )
        :fn(fn_),ln(ln_),pn(pn_),e(e_)
    {}
}

```

```

};

class Customer
{
public:
    Customer(const CustomerParameter& cp)
        :firstname_(cp.fn),lastname_(cp.ln)
    {}

    void print() const
    {
        cout << firstname_
              << ' '
              << lastname_;
    }

private:
    const char *firstname_,
               *lastname_;
};

template <class Base>
class PhoneContact: public Base
{
public:
    PhoneContact(const CustomerParameter& cp)
        :Base(cp),phone_(cp.pn)
    {}

    void print() const
    {
        Base::print();
        cout << ' ' << phone_;
    }

private:
    const char *phone_;
};

template <class Base>
class EmailContact: public Base
{
public:
    EmailContact(const CustomerParameter& cp)
        :Base(cp),email_(cp.e)
    {}

    void print() const
    {
        Base::print();
        cout << ' ' << email_;
    }

private:
    const char *email_;
};

int main()
{
    Customer c1(CustomerParameter("Teddy","Bear"));
    c1.print(); cout << endl;
    PhoneContact<Customer>
    c2(CustomerParameter("Rick","Racoon","050-998877"));
    c2.print(); cout << endl;
    EmailContact<Customer>
    c3(CustomerParameter("Dick","Deer","", "dick@deer.com"));
    c3.print(); cout << endl;
    EmailContact<PhoneContact<Customer> >
    c4(CustomerParameter("Eddy","Eagle","049-554433","eddy@eagle.org"));
    c4.print(); cout << endl;
    PhoneContact<EmailContact<Customer> >
    // The following composition prints the last two
    // arguments in a reverse order because

```

```

    // the print() method is now composed differently.
    c5(CustomerParameter("Eddy", "Eagle", "049-554433", "eddy@eagle.org"));
    c5.print(); cout << endl;
    return 0;
}

```

Listing 3: Providing a special argument class

This solution also has several drawbacks. Every time when a new mixin class with additional constructor parameters is added, the argument class has to be updated accordingly. Fortunately, the source code of the already defined mixin classes does not break, but only needs to be recompiled. Furthermore, an additional parameter object has to be created when defining the desired object, which is awkward for the client programmer. We should also note that this solution is not very efficient because arguments will be created even if they are not required. By providing default values for optional arguments, we only (partially) hide this fact from the client programmer. This illusion breaks if one of the arguments preceding the last argument is optional. In this case, the optional argument must be specified, as the declaration of the `EmailContact<Customer>` object shows.

This solution allows symmetric composition, e.g., `PhoneContact<EmailContact<Customer> >` and `EmailContact<PhoneContact<Customer> >`. But it should be recognized that the resulting `print()` method is composed differently in both cases.

2.3 Providing initialization methods

This approach, which is also described in [SB00], departs from the common C++ practice that an object has to be properly initialized by executing one of its constructors. No constructor is defined – which implies the generation of a default constructor by the compiler – or a standard constructor is implemented that initializes the class members with some reasonable default values. The actual initialization is left to special initialization methods that assign their argument values to the class members (Listing 4).

```

#include <iostream>
using namespace std;

// Define special initialization methods in each class
// and no longer rely on the proper initialization
// though constructors.

class Customer
{
public:
    // Initialization method for Customer.
    // A default constructor will be
    // generated automatically.
    void init(const char* fn, const char* ln)
    {
        firstname_ = fn;
        lastname_ = ln;
    }
    void print() const
    {
        cout << firstname_
              << ", "
              << lastname_;
    }

private:
    const char *firstname_,
               *lastname_;
};

template <class Base>
class PhoneContact: public Base
{

```

```

public:
    // Initialization method for PhoneContact only.
    // A default constructor will be
    // generated automatically.
    void init(const char* pn)
    {
        phone_ = pn;
    }

    void print() const
    {
        Base::print();
        cout << ' ' << phone_;
    }

private:
    const char *phone_;
};

template <class Base>
class EmailContact: public Base
{
public:
    // Initialization method for EmailContact only.
    // A default constructor will be
    // generated automatically.
    void init(const char* e)
    {
        email_ = e;
    }

    void print() const
    {
        Base::print();
        cout << ' ' << email_;
    }

private:
    const char *email_;
};

int main()
{
    // Compiler generated default constructor
    // gets called.
    Customer c1;
    // Now explicitly invoke the initialization method.
    c1.init("Teddy", "Bear");
    c1.print(); cout << endl;
    // Basically the same as above.
    PhoneContact<Customer> c2;
    // But initialization method for Customer
    // must also be explicitly invoked!
    c2.Customer::init("Rick", "Racoon");
    c2.init("050-998877");
    c2.print(); cout << endl;
    // Basically the same as above.
    EmailContact<Customer> c3;
    c3.Customer::init("Dick", "Deer");
    c3.init("dick@deer.com");
    c3.print(); cout << endl;
    // Now the three initialization methods of
    // three different mixin classes must
    // be explicitly invoked! The composed
    // class does not provide its own
    // initialization method.
    EmailContact<PhoneContact<Customer> > c4;
    c4.Customer::init("Eddy", "Eagle");
    c4.PhoneContact<Customer>::init("eddy@eagle.org");
    c4.EmailContact<PhoneContact<Customer> >::init("049-554433");
    c4.print(); cout << endl;
    // Basically the same as above.
    PhoneContact<EmailContact<Customer> > c5;

```

```

    c5.Customer::init("Eddy","Eagle");
    c5.EmailContact<Customer>::init("eddy@eagle.org");
    c5.PhoneContact<EmailContact<Customer> >::init("049-554433");
    c5.print(); cout << endl;
    return 0;
}

```

Listing 4: Providing initialization methods

This approach is error-prone because the client programmer is responsible for calling the necessary initialization methods in the correct order. The latter is important because one cannot generally assume that the initialization of the members of derived classes never depends on the base class members. Furthermore, inherited initialization methods must be invoked using explicit qualification syntax, which is awkward.

2.4 Defining additional constructors

This approach makes use of the template instantiation mechanism of C++. It is important to know that only those parts of a template class get instantiated that are actually used. Consequently, even the case is possible where partial instantiations of a given template class are legal, whereas a full instantiation is not. This allows us to define the different constructors in mixin classes for the different possible base classes (Listing 5).

```

#include <iostream>
using namespace std;

// Define additional constructors that will
// be instantiated only if required.

class Customer
{
public:
    Customer(const char* fn,const char* ln):firstname_(fn),lastname_(ln)
    {}

    void print() const
    {
        cout << firstname_
              << ' '
              << lastname_;
    }

private:
    const char    *firstname_,
                  *lastname_;
};

template <class Base>
class PhoneContact: public Base
{
public:
    // The following constructors will be instantiated
    // only if required.
    PhoneContact( const char* fn,const char* ln,
                  const char* pn):Base(fn,ln),phone_(pn)
    {}

    PhoneContact( const char* fn,const char* ln,
                  const char* pn,const char* e)
        :Base(fn,ln,e),phone_(pn)
    {}
    void print() const
    {
        Base::print();
        cout << ' ' << phone_;
    }

private:

```

```

    const char    *phone_;
};

template <class Base>
class EmailContact: public Base
{
public:
    // The following constructors will be instantiated
    // only if required.
    EmailContact( const char* fn, const char* ln,
                  const char* e):Base(fn,ln),email_(e)
    {}

    EmailContact( const char* fn,const char* ln,
                  const char* pn,const char* e)
        :Base(fn,ln,pn),email_(e)
    {}

    void print() const
    {
        Base::print();
        cout << ' ' << email_;
    }

private:
    const char    *email_;
};

int main()
{
    Customer c1("Teddy","Bear");
    c1.print(); cout << endl;
    PhoneContact<Customer> c2("Rick","Racoon","050-998877");
    c2.print(); cout << endl;
    EmailContact<Customer> c3("Dick","Deer","dick@deer.com");
    c3.print(); cout << endl;
    EmailContact<PhoneContact<Customer> >
    c4("Eddy","Eagle","049-554433","eddy@eagle.org");
    c4.print(); cout << endl;
    PhoneContact<EmailContact<Customer> >
    // The following composition prints the last two
    // arguments in reverse order because
    // the print() method is composed differently
    // than previously.
    c5("Eddy","Eagle","049-554433","eddy@eagle.org");
    c5.print(); cout << endl;
    return 0;
}

```

Listing 5: Defining additional constructors

At first glance, this approach might look very attractive. But after introducing a new mixin class with special arguments for its initialization, additional constructors would have to be implemented in all dependent classes. Depending on the number of possible compositions of mixin classes, this may result in a maintenance nightmare. Please note that, as already described in Section 2.2, composition of the same mixin classes, but in a different order, may lead to different behaviors, e.g., `PhoneContact<EmailContact<Customer> >` and `EmailContact<PhoneContact<Customer> >` lead to different implementations of the `print()` method. Thus, the number of different compositions may grow more than exponentially.

3 Designing an advanced solution

The previously described approaches 3.2 (Providing a special argument class) and 3.4 (Defining additional constructors) provide the basic framework for designing the more advanced solution.

First, it is a good idea to provide a special class that wraps the constructor arguments and thus provides a uniform constructor interface. What should be achieved then is to provide different

argument wrappers for different compositions of mixin classes. We will see how to solve this problem by applying some template-metaprogramming techniques such as described in [CE00].

Second, the client programmer should be able to adhere to the usual way of declaring objects without the obscuring and awkward argument wrapper construction syntax. Thanks to member templates, highly generic constructors can be implemented that take ordinarily specified arguments and automatically convert them to instances of the argument wrapper classes.

3.1 Heterogeneous value lists defining argument lists

The object model of C++ requires that an object is constructed by calling the constructor of its topmost base(s), then the immediately descendant class(es), and so on, until the constructor of the most derived class is executed. Therefore, the set of constructor arguments of a base class is generally a subset of the constructor arguments of any derived class. Imagine passing a singly linked list containing all arguments instead of separate arguments. Each constructor could then take the elements from the front of the list that it needs as arguments and pass the remainder of the list to the base class constructor. This process goes on until the last element has been taken away and the list is empty. Unfortunately, this is not so easy in C++ because each list element may have a different type and, consequently, each list node also represents a unique type. Hence a highly flexible list is required that links unique node types, each with a possibly different type for storing a value.

The basis for a solution provide compile-time type lists and recursively synthesized types such as described in [CE00] and [CE99]. A type list is a singly-linked list constructed by recursively instantiating a structure template such as the following template `List`:

```
struct NIL
{};

template<class Head_, Tail_ = NIL>
struct List
{
    typedef Head_ Head;
    typedef Tail_ Tail;
};
```

A list of basic signed integral types can be represented as follows:

```
List<signed char,List<short,List<int,List<long> > > >
```

Type lists and an implementation of Lisp primitives on top of them were presented in [CE98, Cza98]. The application of type lists was also discussed in [VA00].

In order to represent a list of arguments of any type and number, we need a kind of a type list that additionally stores values. Such a list of different types and values, which we may call *heterogeneous value list*, was described in [Jär99]. Listing 6 shows a heterogeneous value list for representing argument lists..

```
#include <iostream>
using namespace std;

// We need NIL because - as opposed to
// void - it must be possible to create instances of it.
struct NIL
{};

template <class T,class Next_ = NIL>
struct Param
{
    Param(const T& t_,const Next_& n_ = NIL()):t(t_),n(n_)
    {}
};
```

```

    const T& t;
    Next_ n;
    typedef Next_ N;
};

struct SomePersonParameters
{
    const char *firstname_,
               *lastname_;
    int age_;

    SomePersonParameters(const char* fn,const char* ln,const int age)
        :firstname_(fn),lastname_(ln),age_(age)
    {}
};

int main()
{
    SomePersonParameters p1("Peter","Parrot",3);
    cout << p1.firstname_ << ' '
         << p1.lastname_ << ' '
         << p1.age_ << endl;

    // Can be rewritten as
    Param<const char*,Param<const char*,Param<int> > >
    p2("Peter",Param<const char*,Param<int> >("Parrot",3)); //please note
    //that we can pass 3 as the last element instead of Param<int>(3)
    //because it will be automatically converted using the constructor
    //of Param
    cout << p2.t << ' '
         << p2.n.t << ' '
         << p2.n.n.t << endl;

    // Or more easily readable
    typedef Param<int> T1;
    typedef Param<const char*,T1> T2;
    typedef Param<const char*,T2> T3;
    T3 p3("Peter",T2("Parrot",3));
    cout << p3.t << ' '
         << p3.n.t << ' '
         << p3.n.n.t << endl;
    return 0;
}

```

Listing 6: Template-metafunctor for creating heterogeneous value lists

The code in Listing 6 also demonstrates how to use the `Param` template. Appropriate typedefs slightly simplify the declaration of a parameter type and its instances. Accessing the nodes and the remainder of a list is straightforward. Because `Param` has a type member `N` referring to the type of the remainder of the list, the latter can be easily accessed as `SomeRemainder::N`. Whereas the manual definition of recursive heterogeneous value lists is remarkably awkward, it is easy to automatically compute them as you will see in Section 3.3.

3.2 Configuration repositories and parameter adapter

The next step is more complex because two different techniques have to be appropriately combined. The first technique is to use traits classes [Mye95] as configuration repositories [CE00, CE99]. Configuration repositories allow us to separate the configuration aspect from the implementation of a component, i.e., a mixin class. The only parameter of a base mixin class is then the configuration repository (see `Customer` in Listing 7). The base mixin class exports the configuration repository by defining an alias name as its member type. As a result, if the instantiated mixin class is used itself as a base class parameter, the descendant mixin class can read out this configuration repository and export it again (see `PhoneContact` and `EmailContact` in Listing 7). This way, the configuration repository can be propagated along the inheritance hierarchy. Each mixin class can retrieve any desired information from the configuration repository.

A configuration repository can contain any type information, e.g., types needed by a mixin other than its base class (so called “horizontal parameters” [Gog96]), static constants, and enumeration constants. Furthermore, it can also contain any type that is being composed, even the final type of the complete composition. The configuration repository idiom is inherently recursive because it parameterizes base mixin classes with itself. Listing 7 contains four sample configuration repositories c1, c2, c3, and c4. They are designed to represent the four possible compositions that are also part of the previous examples. In order to demonstrate the ability to read out horizontal parameters, the types of first name, last name, phone number, and email address are also parameterized and as such defined in the configuration repositories and retrieved by the mixin classes. In our example, the components assemble the needed heterogeneous parameter lists themselves rather than retrieving them from the configuration repository. This is because the assembly always follows the same pattern, so that it can be safely performed locally in the components. Consequently, the resulting code is simpler than the one with heterogeneous parameter lists assembled in the configuration repository.

The other technique to use is a highly generic parameter adapter (see `ParameterAdapter` in Listing 7) that accepts any number of constructor arguments of any type and converts them into a singly linked list of types and arguments. The wrapper is a mixin class itself and is expected to be used as the most-derived mixin in a composition. The use of member templates to allow method signatures with arguments of any type and number was previously published in [Mey99, Bat00]. However, to our knowledge, a generic parameter adapter combining this technique with heterogeneous argument lists has not been published elsewhere before. The construction of the parameter adapter is amazingly simple. The adapter includes one generic constructor as a member template for a given number of arguments. In this example, we provided template member functions covering constructors with up to four arguments (Listing 7). In practice, it would be useful to define a few additional generic constructors, let’s say for up to 15 or 20 arguments. The scheme how to implement these template member functions is obvious and can be easily followed. Please note that this is a truly universal parameter adapter that will work for any mixin classes that accept heterogeneous value lists as arguments of their constructors! By the way, if the most-derived mixin class is initialized by a standard constructor, the parameter adapter has to be omitted, of course.

Finally the constructors of the mixin classes have to be adjusted for accepting heterogeneous value lists as arguments.

```
#include <iostream>
#include <string>
using namespace std;

struct NIL
{};

template <class T,class Next_ = NIL>
struct Param
{
    Param(const T& t_,const Next_& n_ = NIL()):t(t_),n(n_)
    {}

    const T& t;
    Next_ n;
    typedef Next_ N;
};

template <class Config_>
class Customer
{
public:
    // Exporting config
    typedef Config_ Config;
    // Create parameter type
    typedef
        Param< typename Config::LastnameType,
```

```

        Param< typename Config::FirstnameType > > ParamType;

    Customer(const ParamType& p)
        :lastname_(p.t),firstname_(p.n.t)
    {}

    void print() const
    {
        cout << firstname_
            << ' '
            << lastname_;
    }

private:
    typename Config::FirstnameType firstname_;
    typename Config::LastnameType lastname_;
};

template <class Base>
class PhoneContact: public Base
{
public:
    // retrieve config and export it
    typedef typename Base::Config Config;
    // retrieve the constructor parameter type from the base class
    // and extend it with own parameters
    typedef Param< typename Config::PhoneNoType,
        typename Base::ParamType > ParamType;

    PhoneContact(const ParamType& p)
        :Base(p.n),phone_(p.t)
    {}

    void print() const
    {
        Base::print();
        cout << ' ' << phone_;
    }

private:
    typename Config::PhoneNoType phone_;
};

template <class Base>
class EmailContact: public Base
{
public:
    // retrieve config and export it
    typedef typename Base::Config Config;
    // retrieve the constructor parameter type from the base class
    // and extend it with own parameters
    typedef Param< typename Config::EmailAddressType,
        typename Base::ParamType> ParamType;

    EmailContact(const ParamType& p)
        :Base(p.n),email_(p.t)
    {}

    void print() const
    {
        Base::print();
        cout << ' ' << email_;
    }

private:
    typename Config::EmailAddressType email_;
};

template <class Base>
struct ParameterAdapter: Base
{
    // Retrieve config form Base and export it.
    typedef typename Base::Config Config;

```

```

// Retrieve the most complete param type
typedef typename Config::RET::ParamType P;
typedef typename P::N P1;
typedef typename P1::N P2;

// Constructor adapter with 1 argument
template < class A1
>
ParameterAdapter( const A1& a1)
:Base(a1)
{}

// Constructor adapter with 2 arguments
template < class A1,
          class A2
>
ParameterAdapter( const A1& a1,
                  const A2& a2)
:Base(P(a2,a1))
{}

// Constructor adapter with 3 arguments
template < class A1,
          class A2,
          class A3
>
ParameterAdapter( const A1& a1,
                  const A2& a2,
                  const A3& a3)
:Base(P(a3,P1(a2,a1)))
{}

// Constructor adapter with 4 arguments
template < class A1,
          class A2,
          class A3,
          class A4
>
ParameterAdapter( const A1& a1,
                  const A2& a2,
                  const A3& a3,
                  const A4& a4)
:Base(P(a4,P1(a3,P2(a2,a1))))
{}
};

struct C1
{
// Provide standard name for config
typedef C1 ThisConfig;

// Provide elementary types
typedef const char* FirstnameType;
typedef const char* LastnameType;

// Parameterize base class
typedef Customer<ThisConfig> CustomerType;

// Add ParameterAdapter
typedef ParameterAdapter<CustomerType> RET;
};

struct C2
{
// Provide standard name for config
typedef C2 ThisConfig;

// Provide elementary types
typedef const char* FirstnameType;
typedef const char* LastnameType;
typedef const char* PhoneNoType;

// Assemble mixin classes

```

```

typedef Customer<ThisConfig> CustomerType;
typedef PhoneContact<CustomerType> PhoneContactType;

// Add ParameterAdapter
typedef ParameterAdapter<PhoneContactType> RET;
};

struct C3
{
    // Provide standard name for config
    typedef C3 ThisConfig;

    // Provide elementary types
    typedef const char* FirstnameType;
    typedef const char* LastnameType;
    typedef const char* EmailAddressType;

    // Assemble mixin classes
    typedef Customer<ThisConfig> CustomerType;
    typedef EmailContact<CustomerType> EmailContactType;

    // Add ParameterAdapter
    typedef ParameterAdapter<EmailContactType> RET;
};

struct C4
{
    // Provide standard name for config
    typedef C4 ThisConfig;

    // Provide elementary types
    typedef const char* FirstnameType;
    typedef const char* LastnameType;
    typedef const char* PhoneNoType;
    typedef const char* EmailAddressType;

    // Assemble mixin classes
    typedef Customer<ThisConfig> CustomerType;
    typedef PhoneContact<CustomerType> PhoneContactType;
    typedef EmailContact<PhoneContactType> EmailContactType;

    // Add ParameterAdapter
    typedef ParameterAdapter<EmailContactType> RET;
};

int main()
{
    C1::RET c1("Teddy", "Bear");
    c1.print(); cout << endl;
    C2::RET c2("Rick", "Raccoon", "050-998877");
    c2.print(); cout << endl;
    C3::RET c3("Dick", "Deer", "dick@deer.com");
    c3.print(); cout << endl;
    C4::RET c4("Eddy", "Eagle", "049-554433", "eddy@eagle.org");
    c4.print(); cout << endl;

    return 0;
}

```

Listing 7: Configuration repositories and parameter adapter

3.3 Configuration generator

Obviously, the manual creation of configuration repositories is tedious and error-prone [CE00, CE99]. First, there may be a large number of mixins in a library and the application programmer would have to be aware of them. Second, not all possible configurations of mixin classes are semantically correct and the application programmer would have to know what the correct ways to configure the mixins are. Third, given a larger number of mixin classes, the configuration repositories themselves can reach considerable sizes and the number of possible configurations

usually grows exponentially (for example, we developed a matrix library, whose mixin classes can be configured into almost 2000 different matrix types [Neu98, Cza98, CE00]). Finally, achieving abstract features such as performance specifications requires preferring certain constellations of mixin classes over other constellations and the application programmer would have to know them. Therefore, we would like to allow application programmers to specify the desired properties of a configuration and have the appropriate configuration generated automatically from the specification. Template metaprogramming [Vel95] (which is a set of programming techniques in C++) and generative programming (which is an analysis, design, and implementation approach) provide the necessary foundation for developing so called “configuration generators” [CE00, CE99], which solve the problems outlined above. Listing 8 contains such a configuration generator for the various customer types. It is implemented as a template metafunction (i.e., a class template conceptually representing a function to be executed at compile time; see [CE00] for a discussion of template metafunctions) that accepts a specification of the desired customer type in a so called “domain specific language” (DSL) and returns the concrete customer type in its type member `RET`. The DSL in this example is kept relatively simple. Using enumeration constants, client programmers can specify whether a customer type should include information for phone contact or email contact or not. Furthermore, they can specify the types of the various members of the mixin classes. A default value is provided for each parameter, so that even a default customer type can be “ordered” using an empty specification. Of course, the DSL needs to be clearly documented, so that clients know what parameters and parameter values are available. Similar as in [CE99], the use of template parameters to control code generation is also advocated in [VA00].

The implementation of the generator follows the principles described in [CE00, CE99]. First the DSL is parsed. There is no buildability check because we assume that no wrong specification is requested by the client programmer (or the client program using the generator). The next step is to assemble the components, i.e., the mixin classes, according to the specified features. Meta-control structures, such as `IF` and `SWITCH` (see [CE00]; they are part of the include file `meta.h`, which is not shown here), are used for assembling the components. The final result is made available through the `typedef-name RET`, which is a convention commonly used in template metaprogramming.

Finally, the configuration repository is computed. This example demonstrates a special technique for assembling configuration repositories. Normally, one could put all the information into one single configuration repository. In most cases, this works perfectly because the involved components, i.e., mixin classes, will retrieve only the information they need for themselves from the configuration repository. However, in order to preserve the different structures of the configuration repositories from Listing 7, we decided to compose these different configuration repository types using inheritance and to select the one that is best suited for the generated component using a `SWITCH`. To our knowledge, this technique also has not been published elsewhere before. Please note that we use a kind of static overriding technique when redefining selected names in the derived configuration repositories, e.g., `FinalParamType`.

```
#include <iostream>
#include <string>
using namespace std;
#include "meta.h"
using namespace meta;
#ifdef _MSC_VER
    #pragma warning (disable:4786)
    #pragma warning (disable:4305)
    #define GeneratorRET RET
#else
    #define GeneratorRET Generator::RET
#endif

struct NIL
{
};
```

```

template <class T,class Next_ = NIL>
struct Param
{
    Param(const T& t_,const Next_& n_ = NIL()):t(t_),n(n_)
    {}

    const T& t;
    Next_ n;
    typedef Next_ N;
};

// We will pass Generator to Customer
// rather than Config; the latter will be
// nested in Generator, and Customer
// has to retrieve it; this modification
// of Customer is necessary to avoid certain
// circularity problems.
template <class Generator_>
class Customer
{
public:
    // Exporting config
    typedef typename Generator_::Config Config;
    // ...
    // Rest of Customer is the same as in listing 7.

// The remaining mixin classes and the parameter adapter
// are the same as in listing 7.

// Bitmask for describing customer options - part of the domain
// specific language (DSL) for describing customers
enum CustomerSpec
{
    BasicCustomer = 0, // values represent bits in a bitmask
    withPhone     = 1,
    withEmail     = 2
};

// Customer generator (the generator parameters represent rest of DSL)
template <
    int spec = BasicCustomer, // spec is a bitmask
    class Firstname = const char*,
    class Lastname  = const char*,
    class PhoneNo   = const char*,
    class EmailAdd  = const char*
>
struct CUSTOMER_GENERATOR
{
    // Provide a shorthand for CUSTOMER_GENERATOR ...
    typedef CUSTOMER_GENERATOR < spec,
        Firstname,
        Lastname,
        PhoneNo,
        EmailAdd
    > Generator;

    // Parse DSL
    // Assume there is always a basic customer ...
    enum
    {
        hasPhone = spec & withPhone,
        hasEmail = spec & withEmail
    };

    // Assemble components
    typedef Customer<Generator> Part1;

    typedef typename
        IF < hasPhone,
            PhoneContact<Part1>,
            Part1
        >::RET Part2;

    typedef typename

```

```

    IF < hasEmail,
        EmailContact<Part2>,
        Part2
    >::RET Part3;

// Result of the generator template metafunction:
typedef ParameterAdapter<Part3> RET;

// Compute config
struct BasicCustomerConfig
{ // Provide some metainformation
    enum
    { specification = spec };
    typedef Firstname      FirstnameType;
    typedef Lastname       LastnameType;
    typedef GeneratorRET   RET;
};

struct CustomerWithPhoneConfig: BasicCustomerConfig
{
    typedef PhoneNo        PhoneNoType;
};

struct CustomerWithEmailConfig: BasicCustomerConfig
{
    typedef EmailAdd       EmailAddressType;
};

struct CustomerWithPhoneAndEmailConfig
    : CustomerWithPhoneConfig, CustomerWithEmailConfig
{};

typedef typename
    SWITCH < spec,
        CASE<BasicCustomer, BasicCustomerConfig,
        CASE<withPhone, CustomerWithPhoneConfig,
        CASE<withEmail, CustomerWithEmailConfig,
        CASE<withPhone+withEmail, CustomerWithPhoneAndEmailConfig
    > > > >::RET Config;
};

int main()
{
    CUSTOMER_GENERATOR<>::RET c1("Teddy", "Bear");
    c1.print(); cout << endl;
    CUSTOMER_GENERATOR<withPhone>::RET c2("Rick", "Racoon", "050-998877");
    c2.print(); cout << endl;
    CUSTOMER_GENERATOR<withEmail>::RET c3("Dick", "Deer", "dick@deer.com");
    c3.print(); cout << endl;
    CUSTOMER_GENERATOR<withPhone + withEmail>::RET
        c4("Eddy", "Eagle", "049-554433", "eddy@eagle.org");
    c4.print(); cout << endl;

    return 0;
}

```

Listing 8 Configuration generator for customer types

4 Conclusions

This paper presents a complete solution to the constructor problem in mixin-based programming with C++ as described in [SB00]. The solution effectively helps to completely avoid dependencies between components and special assembly orderings imposed by the number of arguments required by mixin class constructors. The solution also allows client programmers to create instances of the composed types in a natural way. Furthermore, it scales very well because adding new mixin classes requires minimal adaptations only (i.e., only the generator and the DSL need to be extended). Finally, the solution is also very efficient. The only overhead in runtime and memory consumption results from the necessary creation of instances of heterogeneous value lists.

However, this cost seems to be affordable, especially when compared to the costs introduced by the other approaches described at the beginning of this paper. Thanks to configuration generators based on template metaprogramming [CE00], the complexity of the solution is completely hidden from the client programmers. Although this complexity has to be mastered by the generator programmer, it needs to be mastered only once for a given set of mixins. This effort then pays back every time generated components are requested by the client programmers.

The sources of all program examples are available as a zipped archive at <http://home.t-online.de/home/Ulrich.Eisenecker/cpmbp.zip>. They were tested with gcc 2.95.2 and Microsoft Visual C++ 6.0.

References

- [Bat00] V. Batov. Safe and Economical Reference-Counting in C++: Smart pointers keep getting smarter. In *C/C++ Users Journal*, June 2000
- [BC90] G. Bracha and W. Cook. Mixin-Based Inheritance. In *Proceedings of the 8th Conference on Object-Oriented Programming, Systems, Languages, and Applications / European Conference on Object-Oriented Programming (OOPSLA/ECOOP)*, ACM SIGPLAN Notices, vol. 25, no. 10, 1990, pp. 303-311
- [CE98] K. Czarnecki and U. W. Eisenecker. Template-Metaprogramming, <http://home.t-online.de/home/Ulrich.Eisenecker/meta.htm>
- [CE99] K. Czarnecki and U. W. Eisenecker. Synthesizing Objects. In *Proceedings of ECOOP'99 - Object-Oriented Programming*, R. Guerraoui, (Ed.), LNCS 1628, Springer-Verlag, Berlin and Heidelberg, Germany, 1999, p. 18-42. En extended version will appear in *Concurrency: Practice and Experience*, see www.prakinf.tu-ilmenau.de/~czarn/cpe2000
- [CE00] K. Czarnecki and U. W. Eisenecker. *Generative Programming. Methods, Tools, and Applications*. Addison-Wesley 2000, see www.generative-programming.org
- [Cza98] K. Czarnecki. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*. Ph. D. Thesis, Department of Computer Science and Automation, Technical University of Ilmenau, Germany, October 1998
- [Gog96] J. A. Goguen. Parameterized Programming and Software Architecture. In *Proceedings of the Fourth International Conference on Software Reuse*, April 23-26, Orlando, Florida. M. Sitaraman (Ed.), IEEE Computer Society Press, Los Alamitos, CA, 1996, pp. 2-10
- [Jär99] J. Järvi. Tuples and multiple return values in C++. Turku Centre for Computer Science, Technical Report no. 249, Finland, 1999, <http://www.tucs.abo.fi/publications/techreports/TR249.html>
- [Kee89] S. Keene. *Object-oriented programming in Common Lisp: a Programmer's Guide to CLOS*. Addison-Wesley, Reading, MA, 1989
- [Mey99] S. Meyers. Implementing operator->* for Smart Pointers. In *Dr. Dobb's Journal*, October 1999, <http://www.ddj.com/articles/1999/9910/9910b/9910b.htm>
- [Moo86] D. A. Moon. Object-Oriented Programming with Flavors. In *Proceedings of the 1st ACM Conference on Object-Oriented Programming Languages and Applications (OOPSLA '86)*, ACM SIGPLAN Notices, vol. 21, no. 11, 1986, pp. 1-8
- [Mye95] N. C. Myers. Traits: a new and useful template technique. In *C++ Report*, June

1995, see www.cantrip.org/traits.html

- [Neu98] T. Neubert. Anwendung von generativen Programmier-techniken am Beispiel der Matrixalgebra. Master Thesis (in German), Technical University of Chemnitz, Germany, 1998
- [SB98] Y. Smaragdakis and D. Batory. Implementing Layered Designs with Mixin Layers. In *Proceeding of the 12th European Conference on Object-Oriented Programming (ECOOP'98)*, E. Jul, (Ed.), 1998, pp. 550-570
- [SB00] Y. Smaragdakis and D. Batory. Mixin-Based Programming in C++. In *Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering (GCSE'2000)*, Erfurt, Germany, 2000 (to appear)
- [Str94] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, Reading, MA, 1994
- [Str97] B. Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley, Reading, MA, 1997
- [VA00] J. Vlissides and A. Alexandrescu. To Code or not to code. Part I in *C++ Report*, March 2000, and Part II, in *C++ Report*, June 2000
- [Vel95a] T. Veldhuizen. Using C++ template metaprograms. In *C++ Report*, vol. 7, no. 4, May 1995, pp. 36-43, see <http://oonumerics.org/blitz/papers/>
- [VN96] M. VanHilst and D. Notkin. Using Role Components to Implement Collaboration-Based Designs. In *Proceedings of the 1996 ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96)*, 1996, pp. 359-369