SUBTLE METHODS IN C++

Zalán SZŰGYI, Norbert PATAKI, József MIHALICZA
Department of Programming Languages and Compilers, Eötvös Loránd University, Pázmány Péter sétány 1/C, H-1117 Budapest, Hungary, e-mail: lupin@ludens.elte.hu, patakino@elte.hu, jmihalicza@gmail.com

ABSTRACT

Nowadays complex software systems are designed and implemented with the help of the object-oriented paradigm principally. However, object-oriented languages support the object-oriented paradigm in different ways with different constructs. C++ has a sophisticated inheritence notation based on access modifiers. C++ distinguishes virtual, pure virtual and non-virtual methods. Java uses final classes and methods to disable inheritance. However, Java does not support multiple inheritance. Eiffel allows renaming inherited methods.

In this paper we present some method utilites for C++ to create safer and more flexible object-oriented systems. We present how the method renaming can be implemented. We developed constructs to create final and unhidable methods. These constructs are implemented with the help of C++ template facilities. We present scenarios where one can write safer code with our constructs.

Keywords: *C++, methods, object-oriented programming, template*

1. INTRODUCTION

Object-oriented programming (OOP) is still the most common programming paradigm. It represents an attempt to make programs more closely model the way people think about and deal with the world. In the older styles of programming, a programmer who is faced with some problem must identify a computing task that needs to be performed in order to solve the problem. In this way, programming consists of finding a sequence of instructions that will accomplish that task. In the object-oriented realm instead of tasks we find *objects* - entities that have behaviors, that hold information, and that can interact with one another. Programming consists of designing a set of objects that model the problem. Software objects in the program can represent real or abstract entities in the problem domain [12]. This is supposed to make the design of the program more natural and hence easier to get right and easier to understand.

Many programming languages support object-orientation. *Simula 67* was the very first language that supports this paradigm. Languages like C++, C#, and Java are the most famous ones nowadays. Eiffel has been developed in 1986 by Bertrand Meyer [11], which is also an object-oriented language. Script languages are typically not based on the object-oriented paradigm. However, for instance the current version of PHP supports OOP. In fact, different languages support this paradigm with different constructs [8]. These constructs are highly analyzed and compared with each other because of the popularity of the object-oriented programming paradigm [3, 6, 7, 9].

For example, C++ does not have a superclass of every classes, but in Java class *Object* is the root of the class hierarchy. C++ distinguishes between public, private and protected inheritance. One can write *final* class in Java and C# which are cannot be superclasses [1, 4].

C++ is a multiparadigm programming language that supports the object-oriented paradigm [18]. Multiple inheritance is allowed, but there is no language construct for final classes, final methods, or renaming methods [2]. In C++, a method in a base class is hidden, when a method is declared in a derived class with the same name but with different parameter types and/or constness [16]. Although, this can be avoided with *using* declarations, this scenario is strange [12].

C++ offers the template construct for writing generic functions and classes. However, a new direction has been developed with this construct called *template metaprogramming* (TMP). Metaprograms – among other advantages – are able to check conditions in compilation time. If the condition fails, the compilation process can be stopped. However, in this paper we do not deal with metaprograms, but we take advantage of the power of templates.

We make an effort to make C++ much more sophisticated. We developed useful extensions for C++ to deal with object-orientation in more sophisticated way [14,15,20,21].

This paper is organized as follows. Unhidable methods in C++ are detailed in section 2. Method renaming scenarios are described in section 3. Development of final methods is detailed in section 4. We conclude our results and describe our future work in section 5.

2. UNHIDABLE METHODS

It is common mistake in C++ that the signature of virtual methods disagree in the base and derived class. In this case the virtual methods are not overriden, but *hidden*. Let us consider the hereinafter code snippet:

```
struct X
{
  virtual void f()
  {
    std::cout << "X::f()" << std::endl;
  }

  virtual ~X() {}
};

struct Y:X
{
  virtual void f() const
  {
```

```
    std::cout << "Y::f()" << std::endl;
  }
};

int main()
{
  X* x = new X();
  x->f();
  delete x;
  x = new Y();
  x->f();
  delete x;
}
```

The output of this program is the following:

```
X::f()
X::f()
```

This output seems to be strange. The source of the problem is that the signature of virtual method is not exactly the same in base and in derived class. There is a `const` modifier in class `Y`.

This situation should be avoided. However, the compilers compile this code without error message and only some of them give a warning. To overcome this situation we take advantage of C++ templates facility and preprocessor is used for making our solution convenient to use.

First, we wrap a pointer to a member function into a template class:

```
#define __PTR_MEM(paramlist) template \
<class T> \
struct __Ptr_Mem \
{ \
  void (T::*p)paramlist; \
};
```

After that, we create the tester template class that instantiates the previous template. This tester class checks if the two wrapped pointers can be assigned to each other. If the signature of method of base and derived class is the exactly the same, then this assignment works properly. But, if the signatures are not the same, then this assignment results in compilation error message, that it cannot be converted.

```
#define __TEST(funcname) template \
<class Base, class Der> \
struct __Test \
{ \
  __Ptr_Mem<Base> a; \
  __Ptr_Mem<Der> b; \
  __Test() \
  { \
    a.p = &Base::funcname; \
    b.p = &Der::funcname; \
  } \
};
```

After these macros, we develop the macro that start to check this feature. Macro calls the previous macros, and creates a new method in the anonymous namespace, called `__test_if_hidden_methods`, which calls the `__Test` template's default constructor and checks if the signatures are same:

```
#define TEST_IF_HIDDEN_METHODS( Base, Der, \
  function, paramlist) \
namespace { \
  __PTR_MEM(paramlist) \
  __TEST(f) \
  void __test_if_hidden_methods() \
  { \
    __Test<Base, Der>(); \
  } \
}
```

Let us consider how can one use this solution to disable hide the virtual method in this section very first example:

```
TEST_IF_HIDDEN_METHODS( X, Y, f, () )
```

This macro must be called in the global space. If the code compiles, then the signature is the same in base and derived class, which means proper usage of virtual methods. This causes a minimal overhead, because it creates a global `__Test` object and executes two assignment between two member-to-pointers at runtime, which is cheap operation. Otherwise, the code does not compile, results in the following error message:

```
error: cannot convert 'void (X::*)()'
       to 'void (Y::*)()const'
       in assignment
```

In this section we provide a solution to avoid the problem of hidden virtual methods, which appears when the signature of a method is not the same in the base and derived class.

3. METHOD RENAMING

In the Eiffel programming language the inherited `features` can be `renamed`. When two methods inherited from different base classes have the same name this language element helps to avoid ambiguity in the derived class. While this disambiguation is compulsory in Eiffel, C++ allows us to redefine both methods once as a single method of the derived class. It can happen, however, that the two base classes represents different concepts, and the name clash is simply coincidental. Then we may want to redefine those semantically different methods in the derived class(es) separately just like if they had nothing in common. Through a simple example we show how to rename inherited methods in C++ and thus be able to override equally named methods separately. Let `A` and `B` be our base classes each having a method `foo`:

```
struct A
{
    virtual void foo();
};

struct B
```

```
{
    virtual void foo();
};
```

and a derived class C:

```
struct C : public A, public B
{
    virtual void foo(); // overrides both
};
```

Instead of merging the two methods into one we would like to have separate methods, one for each inherited foo:

```
struct C : public A, public B
{
    virtual void A_foo();
    virtual void B_foo();
};
```

We can achieve that by introducing two extra helper classes, one for each base class, whose purpose is to rename A::foo to A_foo and B::foo to B_foo respectively. For symmetry reasons we present only RenA:

```
struct RenA : public A
{
    virtual void A_foo() { A::foo(); }
    virtual void foo() { A_foo(); }
};
```

By default A_foo behaves like A::foo. This way if A_foo is not overridden, its calls through the derived classes will call foo's original implementation in A. On the other hand, a call to foo through a pointer or reference to A should result in a call to A_foo, this is the actual renaming step. Calls to foo through the base class interface leads the execution to the implementation of A_foo either in this or in the appropriate derived class.

Note that it is advisable to set foo final in this class to avoid misuse in derived classes by further overriding foo instead of its renamed equivalent. In the internals of the C++ object model, the foo function remains present in all derived classes in the virtual dispatch table, after all it is part of the hierarchy's interface. The solution for finalizing a method can be found in 4.

Having this rename helper class implemented for B as well there is nothing more left than changing the base classes of C from A to RenA and from B to RenB respectively:

```
struct C : public RenA, public RenB
```

Now we can use the renamed methods in C as if they were the original.

This solution nicely fits to other features in connection with method overriding. The overridden version of A_foo can for example call its original implementation in A by simply calling A::foo just like if we did not have the helper class:

```
void A_foo()
{
    // added code
    A::foo(); // can simply call
              // base if needed
    // added code
}
```

A caveat with the helper class that one should implement do-nothing forwarders for each constructor in it to make them available in the derived class. In C++ the constructor's initializer list is allowed to refer only to *direct* base classes. With a few lines of preprocessing metaprogramming (see boost.preproc [10]) an even more comfortable syntax can be achieved:

```
struct A { ... };

DEF_RENAMER(RenA, A,
    ((foo) (A_foo))
    // ((bar) (A_bar)) // we can add more
                       // renames easily
)

// similarly for B

struct C : public RenA, public RenB { ... };
```

4. FINAL METHODS

In the Java programming language it is possible to declare a member function as final [5]. It means that the member function cannot be overridden in subclasses. There are two benefits of that: first is concerning to the program design and the code quality, the second belongs to the performance (compiler can inline these functions). In the C++ programming language there are good mechanisms to make functions inline, but there is no language support to prevent overriding virtual member functions in subclasses.

Stroustrup et. al. [22] presented a solution to stop deriving of a class. In this chapter we show a solution to make a C++ virtual member function unoverridable. Let us suppose we have a base class A with a virtual member function void f(), and we want to make it "final". In the rest of the section we suppose the objects are created dynamically, because in C++ the polymorphism works by pointers. It works by references also but our solution is limited to pointers. One of our future work is to extend it to references as well.

First, we need to work out that every object of class A or subclasses of A must be created by a specific factory function instead of writing new. This factory function checks whether the function f() is overridden. If not, it creates a new instance of the class, otherwise emits a compile-time error message and the compilation fails. To achieve this we have to define a private operator new in class A, and declare the factory function as friend.

We need a helper class, which describes the member function to make final. See below:

```
template <class C, void (C::*p)()>
class Helper { };
```

ISSN 1335-8243 (print) © 2011 FEI TUKE
www.aei.tuke.sk

ISSN 1338-3957 (online)
www.versita.com/aei

The first template argument is an arbitrary type and the second one is a pointer to the proper member function.

The `template<class T> struct Final` checks whether the `class T` has different member function `f` from `A::f()` in the following way:

```
template <class T>
struct Final
{
  Final()
  {
    const bool b =
      boost::is_same<
        Helper<A, &A::f>,
        Helper<A, &T::f> >::value;

    BOOST_MPL_ASSERT_MSG(
       b,
       ERROR_INVALID_OVERRIDE_OF_FUNCTION,
       (void)
    );
  }
};
```

Metafunction `is_same` [10] is provided by the boost library, and it checks in compile time whether its two template arguments are the same. The macro `BOOST_MPL_ASSERT_MSG` [10] creates a compile time error message when its first argument is false. The second argument is the error message, and the third one holds some type information which is not necessary here. If `T` is subclass of A and the member function `f` is not overidden in T then `T::f` is the same member function as `A::f`, thus the two Helper classes have the same type.
The factory function is the following:

```
template<typename T>
T* factory()
{
    Final<T>();
    T* t = new T();
    return t;
}
```

If it can create the temporary `Final<T>` object, it means that the member function `f` is not overridden. Otherwise the `BOOST_MPL_ASSERT_MSG` in the constructor of `Final` causes a compilation error.

Most of these source codes are generated by preprocessor macros in the following way:

```
struct A
{
    virtual void f() {}
    PREPARE_FINAL_METHODS
};
SET_FINAL(A, f, void, ())
```

The macro `PREPARE_FINAL_METHODS` generates the private `operator new` and the friend declaration of function `factory`. The macro `SET_FINAL(A, f, void, ())` makes the member function f as final member function. The first argument of the macro is the class, the second one is the function, the third one is the return type, and the last one is the argument type list.

We provide `SET_FINALn` preprocessor macros to define more than one member functions to final. The `n` indicates the number of member functions to set to final. This macro has `n` times four arguments. (Four for each member function like in the previous example.)

The following example shows a complex usage of this solution:

```
struct A
{
    virtual void f() { /* ... */ }
    virtual int g(int, double) { /* ... */ }
    virtual char h() { /* ... */ }
    virtual void k() { /* ... */ }

    PREPARE_FINAL_METHODS
};

SET_FINAL3(A, f, void, (),
           A, g, int, (int, double),
           A, h, char, () )

struct B : A
{
    int g(int, double) { /* ... */ }
};

struct C : A
{
    void k() { /* ... */ }
};

int main()
{
    B* b = factory<B>(); // ERROR
    C* c = factory<C>(); // OK
}
```

The `SET_FINAL3` macro creates the specific Helper and Final classes and factory function for the member functions `f`, `g` and `h` what we want to set final. The `struct B` overrides the member function `g` of `struct A`, and the `struct C` does the same with member function `k`. When we want to create an instance of `B` we get the following error message:

```
assertion_failed(mpl_::failed***********
(Final<R>::Final() [with R = C]::
  ERROR_INVALID_OVERRIDE_OF_FUNCTION
::***********)
```

This error message occurs because `struct B` overrides at least one final member function of its base class `A`. However we can create an instance of `struct C` because member function `k` is not final.

5. CONCLUSION AND FUTURE WORK

In this paper we present implementations of different object-oriented features in C++ that are not available as lan-

guage constructs. These features make development easier, safer and more flexible. The idea of final classes comes from Java and its implementation takes advantage of C++ templates. The idea of unhidable methods comes from a frequent mistake when inheritance and overloading is applied. The solution also uses the C++ template construct.

Defining final member functions is beneficial for both design and efficiency reasons. While the C++ programming language does not support it natively, we presented a solution how to apply this object-oriented feature in C++. In the current phase of developement there is a limitation of setting member functions as final: if there is a member function in a base class (called `f`) to set as final there is not allowed to create a member function f in the derived classes even it has different argument types. Our future work is to improve our solution to eliminate this limitation.

ACKNOWLEDGEMENT

The Project is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TÁMOP 4.2.1./B-09/1/KMR-2010-0003).

REFERENCES

[1] ARNOLD, K. – GOSLING, J. – HOLMES, D.: The Java Programming Language, Third Edition, Addison-Wesley, Reading, 2005.

[2] BACON, D. F. – SWEENEY, P. F.: *Fast static analysis of C++ virtual function calls*, Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '96), ACM SIGPLAN Notices, Vol. 31, No. 10, 1996, pp. 324–341.

[3] BETTINI, L. – CAPECCHI, S. – VENNERI, B.: *Double dispatch in C++*, Software: Practice and Experience, Vol. 36, No. 6, 2006, pp. 581–613.

[4] BETTINI, L. – CAPECCHI, S. – VENNERI, B.: *Featherweight Java with dynamic and static overloadingstar*, Science of Computer Programming, Vol. 74, No. 5–6, 2009, pp. 261–278.

[5] BETTINI, L. – CAPECCHI, S. – VENNERI, B.: *A Safe Implementation of Dynamic Overloading in Java-Like Languages*, Proceedings of Fundamentals of Software Engineering, Revised Selected Papers, Lecture Notes in Computer Science, Vol. 5961, No. 1, 2010, pp. 455-462.

[6] BIBERSTEIN, M. – GIL J. (Y.) – PORAT, S.: *Sealing, Encapsulation, and Mutablility*, Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2001), Lecture Notes in Computer Science, Vol. 2072, No. 1, 2001, pp. 28–52.

[7] DEAN, J. – CHAMBERS, C. – GROVE, D.: *Selective specialization for object-oriented languages*, ACM SIGPLAN 95 Conference on Programming Language Design and Implementation (PLDI95), ACM SIGPLAN Notices, Vol. 30, No. 6, 1995, pp. 93–102.

[8] DUCOURNAU, R. – MORANDAT, F. – PRIVAT, J.: *Empirical assessment of object-oriented implementations with multiple inheritance and static typing* Proc. of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA 2009), Vol. 44, No. 10, 2009, pp. 41–60.

[9] ISHIZAKI, K. – KAWAHITO, M. – YASUE, T. – TAKEUCHI, M. – OGASAWARA, T. – SUGANUMA, T. – ONODERA, T. – KOMATSU, H. – NAKATANI, T.: *Design, Implementation, and Evaluation of Optimalizations in a Just-In-Time Compiler*, Concurrency: Practice and Experience, Vol. 12, No. 6, 2000, pp. 457–475.

[10] KARLSSON, B.: Beyond the C++ Standard Library: An Introduction to Boost, Addison-Wesley, Reading, MA., 2005.

[11] MEYER, B.: Eiffel: The Language. Prentice-Hall, Upper Saddle River, 1991.

[12] MEYER, B.: Object-oriented Software Construction. Second Edition, Prentice-Hall, Upper Saddle River, 1997.

[13] MEYER, B.: *Overloading vs Object Technology*, Journal of Object-Oriented Programing (JOOP), Vol. 14, No. 4, 2001, pp. 3–7.

[14] MIHALICZA, J. – PATAKI, N. – PORKOLÁB, Z. – SIPOS, Á: Towards More Sophisticated Access Control, in Proceedings of 11th Symposium on Programming Languages and Software Tools and 7th Nordic Workshop on Model Driven Software Engineering, pp. 117–131.

[15] PATAKI, N.: *Advanced Functor Framework for C++ Standard Template Library*, Studia Univ. Babeş-Bolyai, Informatica, Vol. LVI, No. 1, 2011, pp. 99–113.

[16] PIRKELBAUER, P. – SOLODKYY, Y. – STROUSTRUP, B.: *Open multi-methods for C++*, in Proc. of of the 6th international conference on Generative programming and component engineering (GPCE 2007), 2007, pp. 123–134.

[17] PORKOLÁB, Z. – MIHALICZA, J. – SIPOS, Á: *Debugging C++ Template Metaprograms*, in Proc. of Generative Programming and Component Engineering 2006, The ACM Digital Library, pp. 255–264.

[18] STROUSTRUP, B.: *Why C++ is not just an object-oriented programming language*, ACM SIGPLAN OOPS Messenger, Vol. 6, No. 4, 1995, pp. 1–13.

[19] STROUSTRUP, B.: The C++ Programming Language, Special Edition, Addison-Wesley, Reading, MA., 2000.

[20] SZŰGYI, Z. – PATAKI, N.: *Sophisticated Methods in C++*, in Proc. of International Scientific Conference on Computer Science and Engineering (CSE 2010), pp. 93–100.

[21] SZŰGYI, Z. – PATAKI, N. – MIHALICZA, J. – PORKOLÁB, Z.: *C++ Method Utilities*, in Proc. of the Tenth International Conference on Informatics (Informatics 2009), pp. 112–117.

[22] http://www.research.att.com/bs/bs_faq2.html#no-derivation

Received June 4, 2011, accepted September 5, 2011

BIOGRAPHIES

Zalán Szűgyi was born on 11. 2. 1980. In 2007 he graduated (MSc) at the department of Eötvös Loránd University (ELTE), Budapest. He plans to defend his PhD in 2013, of which title is "Static analysis of C++ programs". Since 2010 he is working as a lecturer at Department of Informatics. His scientific research is focusing on static analysis of C++ source codes and extending the language expression power.

Norbert Pataki was born on 26. 2. 1982. In 2006 he graduated (MSc) at the department of Eötvös Loránd University (ELTE), Budapest. He takes part many industrial projects during his PhD. In 2009 he has become assistant professor at Eötvös Loránd University. His research area includes programming languages (especially the C++ programming language), multicore programming, software metrics, and generative programming.

József Mihalicza was born on 3. 6. 1979. In parallel to his university studies he started working as software developer. Currently he is lead architect at NNG Ltd. In 2006 he graduated (MSc) at the department of Informatics, Eötvös Loránd University (ELTE), Budapest. He plans to defend his PhD in 2012, of which title is "Practical problems of large-scale generic program systems". The major results of his scientific research are a C++ template metaprogram debugger, profiler and visualizer, a typed heap profiler and visualizer for C++, selective access control in C++ and build time optimisation of C/C++ software systems.

ISSN 1335-8243 (print) © 2011 FEI TUKE
www.aei.tuke.sk

ISSN 1338-3957 (online)
www.versita.com/aei