Type Mosaicing with Consultables and Delegates

Nicolas Bouillot

When assembling a class with complex behaviour out of several other feature classes, the most common approaches are multiple inheritance [1] and object composition [2]. This article introduces an other alternative called *type mosaicing* which enables the assembly of multiple object interfaces, all being visible independently to the class user without the need for writing wrappers. This is achieved using an update of previously described making and forwarding of consultables and delegates [Bou15], which use C++ template meta-programming for automating the wrapping of delegate's public methods, possibly enabling const only methods (*consultable*) or all public methods (*delegate*).

Consider for instance a base class called `Element` being composed of members offering complex features, such as a documentation tree with serialization and deserialization, an event system, a state system and others. All these features and their interfaces already exist independently (or are wanted to be kept independent). Feature being class members are not enough: how to use them remains to be specified. It is wanted for instance that the user has a read-only (const methods) access to the documentation tree interface, while the base class and subclasses has read/write access. How can this kind of behaviour be implemented? Several approaches can be used: writing wrappers gives full control, but while the event system interface is powerful, it also very detailed and requires many wrappers. A less verbose approach could be writing simple feature accessors for feature members, but this introduces ownership related issues. One could instead use multiple inheritance, but both the documentation three member and the state system member have a `get` method. Would it be easier if the `Element` class could "import" read-only or read/write member interfaces into its public and/or protected declarations, with a compile time generated wrapper that avoid possible conflicts? This would allow expressive design of feature assembled classes without ownership related issue, no wrapper to maintain, no conflicts among feature interfaces and no constraint imposed on feature classes.

So the motivation here is the assemblage of feature classes interfaces[3]. In this context, there are some issues with common approaches. With C++ *multiple inheritance*, base classes interfaces are visible to the subclass user as if they were inside in the subclass, requiring explicit specification of a method scope name only when method name is ambiguous. Accordingly, including a new feature with multiple inheritance may add ambiguity and therefore may break already present code. Another issue occurs when two features of the same type are wanted in the same assemblage: inherit directly twice from the same base class does not import two isolated interfaces, but rather conflicts and fail to compile.

[1] From [GHJV95], "class inheritance lets you define the implementation of one class in terms of another's. Reuse by subclassing is often referred to as white-box reuse. The term "white-box" refers to visibility: With inheritance, the internals of parent classes are often visible to subclasses."

[2] From [GHJV95], "object composition is an alternative to class inheritance. Here new functionality is obtained by assembling or composing objects to get more complex functionality. Object composition requires that the objects being composed have well-defined interfaces. This style of reuse is called black-box reuse, because no internal details of objects are visible. Objects appear only as "black boxes."

[3] while the decorator pattern intent is to "attach additional responsibilities to an object dynamically" and seems close to the motivation of assembling feature interfaces, the decorator pattern [GHJV95] does not apply here. Indeed, it requires a decorator object's interface to conform to the interface of the component it decorates. Accordingly, a decorator's interface is not added to the decorated component, but used in order to add a specific behaviour to the already existing interface.

With *object composition*, the assemblage is achieved declaring features as members, and then delegate them manually writing boilerplate wrappers, resulting (according to my experience) in a task prone to errors and muscle pains, with a very likely incomplete wrapping of delegated methods. However, unlike inheritance, object composition is avoiding ambiguity among feature's methods and allows for manual hooking of wanted calls for various purpose, such and thread safety, log, etc.

Type mosaicing with *consultable* and *delegate* aims at avoiding the previously described limitations, while keeping flexibility of object composition. Ambiguity avoiding is achieved using automated wrapper generation that makes delegate methods accessible through an *accessor method* (see Listing 1) that isolate delegated features from each other. Additionally, the possibility of specifying particular behaviour around delegate invocation is enabled through two new additions to consultable. They are described here: **selective hooking** allows for replacing the invocation of a delegated method wrapper by a delegator method sharing the same signature. This enables either bypassing the invocation, or adding a particular behaviour when the hook is subcalling the delegated method (log and call, count and call, etc). The second addition is the **global wrapping** feature that allows to user-defined code to be executed before and after delegate method invocation.

Compared to a simple accessor, the *consultable* and *delegate* method provides a much safer approach with a more powerful control over the feature use. Suppose the use of a simple accessor for a feature member: once the user has the object, invocation on it cannot be handled by the assembled class. The second reason is ownership: if the simple accessor returns as reference or a pointer to the feature object, the obtained reference or pointer can be saved by the user. Moreover, with a reference or a pointer, a read-only object can be made writable with `const_cast`. With wrapper generation in *consultable* and *delegate*, the class user does not have access to the object (ownership remains with the assembled class) and each call to its methods can be controlled from the assembled class if wanted.

```
// 'Name' is the class delegated twice by
// the 'NameOwner' class, as declared lines
// 13 and 14. The two delegated member are
// used lines 22 and 23, invoking their
// 'print' method.

1  class Name {
2   public:
3    Name(const string &name): name_(name){}
4    string get() const { return name_; }
5    void print() const { cout << name_; }
6    // ...
7   private:
8    string name_{};
9  };
10
11 class NameOwner {
12  public:
13   Make_consultable(NameOwner, Name, &first_, first);
14   Make_consultable(NameOwner, Name, &second_, second);
15  private:
16   Name first_{"Augusta"};
17   Name second_{"Ada"};
18 };
19
20 int main() {
21   NameOwner nown;
22   nown.first<MPtr(&Name::print)>();  // Augusta
23   nown.second<MPtr(&Name::print)>(); // Ada
24 }
```

Listing 1: consultable

The following will provide a more in-depth view of type mosaicing use and implementation,

providing first a description/reminder of the updated initial design of consultables and delegates. Then *selective hooking* and *global wrapper* are introduced with example code. After, a more complete example of type mosaicing is given, where a class `Element` will have several templated properties `Prop` installed. Finally, brief explanations about implementation will be given.

1 Update about consultables & delegates

Consultables and delegates basically provide automated class member delegation. This is implemented using C++ templates, generating wrapper in user code. The generated wrapper is actually a templated method (called the *accessor method*) taking the original delegated method pointer as parameter. Accordingly, declaring a member as consultable or delegate (see Listing 1) results in the declaration of the *accessor method* templates. `Make_delegate` enables the invocation of **all public methods**, and `Make_consultable` enables the invocation of **public const methods only**.

The previous implementation of consultables and delegates [Bou15] used a delegated method pointer as the accessor method's first argument. However, the implementation of *selective hooking* needs to detect which delegated method is invoked at compile time. The updated implementation now takes delegate method pointer as template parameter, with invocation arguments only aimed at being forwarded to the delegated method.

The new syntax for invocation is shown with Listing 1, lines 22 & 23. Notice the use of the `MPtr(X)` (for method pointer) macro that shortcuts `decltype(X), X`. Indeed, with pointer to method as a template parameter, the method type must be specified before the method pointer. This however does not handle ambiguity among possibly overloaded methods in the delegator. Handling this case, additional shortcut macros are provided: `OPtr` (for overloaded method pointer) and `COPtr` (for const overloads). They however require the specification of return and arguments types, along with the method pointer. For instance, if the `print` method (line 5) would have been overloaded, a user could have specified it using `COPtr(&Name::print, void)`. For this reason, it is preferable to avoid overloaded methods when writing a class that will be delegated with a consultable or a delegate.

```
// 'first' and 'second' are accessor methods
//  declared in the 'NameOwner' class and
// therefore available as methods of 'nown_'
// member. Forward_consultable (lines 3 and 4)
// allows them to be forwarded as Box methods,
// respectively 'fwd_first' and 'fwd_second'.

 1 class Box {
 2  public:
 3   Forward_consultable(Box, NameOwner, &nown_, first, fwd_first);
 4   Forward_consultable(Box, NameOwner, &nown_, second, fwd_second);
 5  private:
 6   NameOwner nown_{};
 7 };
 8
 9 int main() {
10   Box b;
11   b.fwd_first<MPtr(&Name::print)>();  // Augusta
12   b.fwd_second<MPtr(&Name::print)>(); // Ada
13 }
```

Listing 2: forwarding a consultable

Forwarding a consultable or a delegate has the same meaning as in the previous implementation: an accessor method of a class member can be forwarded to a local class accessor method. Listing 2 shows an example of `Forward_consultable`: the `NameOwner` member (line 6) accessor methods

`first` and `second` are forwarded as Box accessor methods, respectively `fwd_first` (line 3) and `fwd_second` (line 4).

As with previous implementation, a delegate (read/write access) can forwarded as a consultable (read-only), blocking then write access to the feature.

2 Accessor Method Hooks

Making and forwarding consultable reduces code size and allows for automating delegation without giving reference to the delegated member. However, when *making* or *forwarding*, a class might want to implement some specific behaviour for all (or for a given) delegated method. This is achieved with *selective hooking* and *global wrapping* described here.

2.1 Selective Hooking

```
// 'Box2' is having a 'NameOwner' member,
// from which it is forwarding accessor
// methods. However, for a particular
// invocation, i.e. '&Name::get' with
// 'fwd_second' (declared lines 5-8), the
// call is hooked and replaced by the
// invocation of 'hooking_get' (declared
// lines 11-15).

1 class Box2 {
2  public:
3   Forward_consultable(Box2, NameOwner, &nown_, first, fwd_first);
4   Forward_consultable(Box2, NameOwner, &nown_, second, fwd_second);
5   Selective_hook(fwd_second,
6                  decltype(&Name::get),
7                  &Name::get,
8                  &Box2::hooking_get);
9  private:
10   NameOwner nown_{};
11   string hooking_get() const {
12     return "hooked! (was "
13         + nown_.second<MPtr(&Name::get)>()
14         + ")";
15   }
16 };
17
18 int main() {
19   Box2 b;
20   cout << b.fwd_first<MPtr(&Name::get)>()  // Augusta
21        << b.fwd_second<MPtr(&Name::get)>() // hooked! (was Ada)
22        << endl;
23 }
```

Listing 3: selective hooking

Sometimes one wants to forward all methods of a delegate, but also wants to change or augment the behaviour of one particular method. Motivations can be testing performance of a new algorithm, emulating a behaviour during debug, counting the calls, measuring the call duration of the delegate method, etc. As presented in Listing 3, this is achieved declaring a *selective hook* (lines 5-8). The selective hook applies to a particular delegated method, here `Name::get`, and is attached to a particular accessor method, here `fwd_second`.

The last argument of `Selective_hook` is a method that will be invoked instead of the original delegated method, here the private `Box2::hooking_get` method. It must therefore have an identical signature (not merely compatible) as the original delegated method. In this hook, the original

method is invoked (line 13) and the returned string "Ada" is placed into an other string, resulting in the printing of a new string "hooked! (was Ada)" when invoked on line 21.

Notice the type of the delegated method is given as second argument (line 6). This allows consultable internals to support overloaded delegate methods, whose type is the only way to distinguish them from other overloaded methods. This argument might have been made not required, but at the cost of not supporting overloads.

2.2 Global Wrapping

```
// the 'Box3' class is forwarding (line 3)
// accessor method of the 'nown_' member
// (line 5). A global wrap is declared for
// this particular forwarding (line 11),
// that will cause the wrapper to call
// 'make_TorPrinter' and store the result
// (a 'TorPrinter') before invoking the
// delegated method. The result will be
// destructed when the wrapper will be
// unstacked.

1 class Box3 {
2  public:
3   Forward_consultable(Box3, NameOwner, &nown_, first, fwd_first);
4  private:
5   NameOwner nown_{};
6   struct TorPrinter{
7     TorPrinter(){ cout << " ctor "; }
8     ~TorPrinter(){ cout << " dtor "; }
9   };
10  TorPrinter make_TorPrinter() const {return TorPrinter();}
11  Global_wrap(fwd_first, TorPrinter, make_TorPrinter);
12 };
13
14 int main() {
15   Box3 b;
16   b.fwd_first<MPtr(&Name::print)>();       // ctor Augusta dtor
17   cout << b.fwd_first<MPtr(&Name::get)>()  // ctor dtor Augusta
18        << endl;
19 }
```

Listing 4: global wrapping

A *Global wrap* is specified for a particular accessor method, but is applied regarding the delegated method invoked. It basically allows for invoking code before and after any delegate invocation, with the exception of delegate methods having a selective hook attached.

Listing 4 shows a use of a *global wrap*. As seen line 11, the `fwd_first` accessor method has a global wrap that will invoke the `make_TorPrinter` method, which returns a `TorPrinter` object. As seen in lines 6-9, a `TorPrinter` object prints "ctor" when constructing, and "dtor" when destructing. As a result, the invocation of `Name::print` (line 16) first constructs a TorPrinter (printing "ctor"), then invokes print (printing "Augusta"), and then destructs TorPrinter (printing "dtor").

Notice that *global wrapping* is here used with the custom type (`TorPrinter`) but can be used with other types, as for instance `std::unique_lock`. The use of a delegate can then be made thread safe with the declaration of a global wrapper that returns a lock applied to a mutex member of the assembled class.

3 Type Mosaicing with consultables and delegates

Listing 5 is an example of *type mosaicing*. The public interface of the class `Element` is built as the assembly of interfaces from its private members, declaring class members as delegates and/or consultables (lines 15-19). An equivalent assembly would be impossible with inheritance since two members share the same type (`info_` and `last_msg_`). With composition, such an assemblage would require the writing of many wrappers, increasing significantly class definition size.

Before entering the listing details, suppose that a new feature is eventually wanted, such as log `set` invocation on `last_msg` (as invoked line 35). Then you would declare a selective hook for the set method applied to `last_msg` accessor method that would invoke set and log. This feature addition would consist only in adding a selective hook without modifying existing methods.

With a closer look at the listing, it can be seen that the use of the `Element` class is achieved through creation of a `Countess` subclass instance line 34. The `last_msg_` can be updated (line 35) through invocation of the `last_msg` accessor method. `num_` can be used at line 36 since `num_` has been declared as a consultable line 16.

Things are a little different for the member `info_`. It has two accessor methods declared: one as consultable for public use (line 15) and one as delegate for subclasses (line 19). As it is it, the class user has a read only access to `info_`. In other words, `Prop<string>::get` is allowed for `info_` and return "programmer" when invoked line 38. Then, invocation of the `mutate` method (declared in the `Countess` subclass) is updating `info_` to "mathematician". This `mutate` method is allowed to write `info_` since the `Countess` class have access to the `prot_info` access method. A new call to `info_` getter (line 40) is accordingly returning "mathematician".

The `info_` related part of the code demonstrates how declaration of *consultables* and *delegates* makes explicit which part of the interface can be invoked by a subclass or by a user, without the noise introduced by wrapper declaration when wrapping multiple features. Achieving the same with manually written wrappers would have resulted into wrapping non-const methods for subclasses and wrapping `info_` const methods once or twice (if protected and public wrappers need to have different implementations). Moreover, suppose a const method is added latter into the `Prop` class, then additional wrapper(s) need to be written in the `Element` class. At the inverse, with *type mosaicing*, the `Element` class stays unchanged, but the addition is visible to the user and the subclass(es).

4 Implementation

My previous paper [Bou15] gave several details about consultable and delegate implementation. The focus here is about description of *selective hooking* and *global wrapping* implementation and a discussion about the use of macros.

4.1 About the use of macros

All previously presented features are implemented using macro declaring types and templates in the class definition. It is argued here that making, forwarding and hooking of consultable and delegates follows a declarative programming paradigm. Indeed, when declaring consultables, delegates and hooks, the programmer is describing *what* the program should accomplish (which wrappers are generated), rather than describing *how* to go about accomplishing (writing wrappers). Unfortunately, C++11 offers very few (or nothing excepted macros?) that enable such a paradigm.

While I have read on the Internet that the use of macro is almost always bad, there might be no other option for approaching declarative programming. Here, making consultable if very close to the declaration of a member qualifier (such as const for instance). For instance, (and naively), if one could add custom qualifier to class members, a consultable could have been declared as follow:

```
// the 'Element' class interface is an assemblage
// of several private members interfaces, using
// 'Make_consultable' and 'Make_delegate'.

template<typename T> class Prop {
 public:
  Prop() = default;
  Prop(const T &val) : val_(val){}
  T get() const { return val_; }
  void set(const T &val) { val_ = val; }
  // ...
 private:
  T val_{};
};

class Element {
 public:
  Element(string info, int num) : info_(info), num_(num){}
  Make_consultable(Element, Prop<string>, &info_, info);
  Make_consultable(Element, Prop<int>, &num_, num);
  Make_delegate(Element, Prop<string>, &last_msg_, last_msg);
 protected:
  Make_delegate(Element, Prop<string>, &info_, prot_info);
 private:
  Prop<string> info_{};
  Prop<int> num_{0};
  Prop<string> last_msg_{};
};

struct Countess : public Element {
  Countess() : Element("programmer", 1){}
  void mutate(){
    prot_info<MPtr(&Prop<string>::set)>("mathematician");
  }
};

int main() {
  Countess a;
  a.last_msg<MPtr(&Prop<string>::set)>("Analytical Engine");
  a.num<MPtr(&Prop<int>::get)>();              // "1"
  // a.info<MPtr(&Prop<string>::set)>("..."); // does not compile
  a.info<MPtr(&Prop<string>::get)>();          // "programmer"
  a.mutate();
  a.info<MPtr(&Prop<string>::get)>();          // "mathematician"
}
```

Listing 5: type mosaicing

```
Prop<string> info_{} : public consultable info, protected delegate prot_info;
```

Which would have the same meaning for `info_` than related declarations in Listing 5, generating two accessor methods, `info` and `prot_info`.

This is not possible with C++ (and maybe not desirable), but as demonstrated with consultable implementation, this can be approached using macros.

Notice also that the use of macro for building declarative statement in C++ has already be promoted, for instance by Andrei Alexandrescu for implementation of `scope_exit`, `scope_fail` and `scope_success` [4].

4.2 Selective Hook

Selective hooking is implemented using template specialization. When declaring a consultable or delegate, a primary template for the `get_alternative` method is declared. The parameter of

[4]See Andrei Alexandrescu's "Declarative flow control" talk at the NDC 2014 conference: `https://vimeo.com/97329153`

```
// 'fun' is the delegate method pointer, 'args' the
// arguments passed for invocation and 'BTs' their
// types. '_member_raw_ptr' is the pointer to the
// delegated member.

1  auto alt = get_alternative<decltype(fun), fun>();
2  if(nullptr != alt)
3    return (this->*alt)(std::forward<BTs>(args)...);
4  auto encap = internal_encaps();
5  return ((_member_rawptr)->*fun)(std::forward<BTs>(args)...);
```

Listing 6: accessor body

this template is the delegated method pointer, allowing specialization per delegated method. This primary template returns `nullptr`. However, when a `selective_hook` is declared, an explicit specialization for `get_alternative` is declared, returning the pointer to the *hook* method.

In the generated wrapper (see Listing 6), the pointer to the *hook* pointer is stored into the `alt` variable (line 1). Then, if a method for hooking has been declared, `alt` is holding a non null pointer. In this case, the hook is invoked instead of delegated method (line 3).

4.3 Global wrapper

Global wrapping is implemented with the `internal_encaps` method, used by the accessor method (line 4 of Listing 6). When declaring a `Global_wrap`, the `internal_encaps` method is declared, wrapping the user method and returning its result. When the consultable or delegate is declared, a dummy `internal_encaps` returning `nullptr` is declared if none has been declared. This is achieved with the use of `enable_if` and a template testing existence of a specific class method.

As a result, when the accessor method is invoked, `internal_ encaps` returns either `nullptr` or the object produced by the global wrapper, which is stored in the stack with the `encap` variable. `encap` will then be freed when the accessor method will be unstacked, after invocation of the delegated method (line 5).

5 Summary

The notion of *type mosaicing* has been introduced as an alternative to *multiple inheritance* and *object composition* when a class interface is designed as an assembly of several other feature interfaces. *Type mosaicing* is made possible thanks to *consultables* and *delegates* which allow for automated wrapper generation with C++ templates. New additions to *consultables* and *delegates* are introduced: *selective hooking* for attaching a specific behaviour to a given feature's method wrapper, and *global wrapping* for attaching code to execute before and after any wrapper invocation of a given feature. This results in an automated interface assembly with full control over feature wrappers.

With *type mosaicing*, being feature classes (consultable or delegate) does not require for compliance with particular constrains like inheriting from a specific base class or implementing one or several specific method(s). It is therefore easier to reuse an existing class as a feature class, and reuse a feature class in a different context. In addition, features are used through wrappers that control read/write accesses with constness, encouraging const-correct feature classes. Moreover, there is no way for a user to access the feature object by copy, reference or address, which helps avoid ownership related issues with features.

As with previous implementation, samples and source code are available on github[5] and have

[5]Source code and examples are available at `https://github.com/nicobou/cpp_make_consultable`

been compiled and tested successfully using with gcc 4.8.4-2ubuntu1, clang 3.4-1ubuntu3 & Apple LLVM version 6.0.

Design and development of *type mosaicing* with *consultable* and *delegate* has been initially made into production code in which major refactoring has been engaged several times: updating wrappers became a major concern. In this production code, the addition of new features is also required periodically and is sometimes necessitating modifications/additions in the core. In this context, feature assembly with minimal boilerplate code became critical.

After searching for similar approaches, comparable techniques have been published. Alexandrescu's *policies* [Ale01], based on template-controlled inheritance, "fosters assembling a class with complex behaviour out of many little classes". MorphJ [HS11] is a Java-derived languages for "specifying general classes whose members are produced by iterating over members of other classes" (Morphing), with possible inlining of delegate into the delegator and optional selecting all members or a subset. DelphJ [GBS13] is extending Morphing with possible transformation of delegated methods (including exposed signature). Delegation proxy in Smalltalk [WNTD14] is comparable to consultable and delegate forwarding, allowing to implement "variations that propagate to all objects accessed in the dynamic extent of a message send. [...] With such variations, it is for instance possible to execute code in a read-only manner".

6 Acknowledgement

Many thanks to Frances Buontempo and the Overload review team for providing constructive comments. This work has been done at the Société des Arts Technologiques and funded by the Ministère de l'Économie, de l'Innovation et des Exportations (Québec, Canada).

References

[Ale01] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Bou15] Nicolas Bouillot. Make and forward consultables and delegates. *Overload Journal*, (127):20–24, 2015.

[GBS13] Prodromos Gerakios, Aggelos Biboudis, and Yannis Smaragdakis. Forsaking inheritance: Supercharged delegation in DelphJ. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 233–252, New York, NY, USA, 2013. ACM.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[HS11] Shan Shan Huang and Yannis Smaragdakis. Morphing: Structurally shaping a class by reflecting on others. *ACM Trans. Program. Lang. Syst.*, 33(2):6:1–6:44, February 2011.

[WNTD14] Erwann Wernli, Oscar Nierstrasz, Camille Teruel, and Stéphane Ducasse. Delegation proxies: The power of propagation. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 1–12, New York, NY, USA, 2014. ACM.

7 Biography

Nicolas Bouillot is a research associate at the Society for Arts and Technology ([SAT], Montreal, Canada). He likes C++ programming, team-based working, writing research papers, networks and distributed systems, data streaming, distributed music performances, teaching, audio signal processing and many other unrelated things.