Separating Interface and Implementation in C++ - accu.org

Overload Journal #66 - Apr 2005 + Programming Topics   Author: Alan Griffiths

This article discusses three related problems in the design of C++ classes and surveys five of the solutions to them found in the literature. These problems and solutions are considered together because they relate to separating the design choices that are manifested in the interface from those that are made in implementing the class. The problems are:

- Reducing implementation detail exposed to the user
- Reducing physical coupling
- Allowing customised implementations

These have led developers to seek ways to separate interface from implementation and practice has seen all of the following idioms used and documented. We will be evaluating them to see how they compare as solutions to the above problems:

- Interface Class
- Cheshire Cat
- Delegation
- Envelope/Letter
- Non-Virtual Public Interface

In order to illustrate the problems and solutions we are going to use a telephone address book (with very limited functionality) as an example. For comparison purposes we have implemented this as a naïve implementation (see first sidebar) which does not attempt to address any of the stated problems. We have also refactored this example to use each of the idioms - the header files are reproduced in the corresponding sidebars. (The full implementation and sample client code for all versions of the example are available with the online version of this article [WEB05].)

Examining the Problems

Problem 1: Reducing Implementation Detail Exposed to the User

Client code makes use of an object via its public interface, without any recourse to implementation details. Since the authors of client code have to use an object through its public interface that interface is all they need to understand. This public interface typically comprises member function declarations.

Naïve Implementation

```
// naive.h - implementation hiding example.
#ifndef INCLUDED_NAIVE_H
#define INCLUDED_NAIVE_H
#include <string>
#include <utility>
#include <map>

namespace naive {
/** Telephone list. Example of implementing a
*     telephone list using a naive implementation.
*/
class telephone_list {
public:
  /** Create a telephone list.
  * @param    name    The name of the list.
  */
  telephone_list(const std::string& name);

  /** Get the list's name.
  * @return   the list's name.
  */
  std::string get_name() const;

  /** Get a person's phone number.
  * @param    person   Person's name (exact match)
  * @return   pair of success flag and (if success)
  *             number.
  */
  std::pair<bool, std::string>
  get_number(const std::string& person) const;

  /** Add an entry. If an entry already exists for
  *     this person it is overwritten.
  *   @param  name    The person's name
  *   @param  number  The person's number
  */
  telephone_list&
  add_entry(const std::string& name,
            const std::string& number);
private:
  typedef std::map<std::string, std::string> dictionary_t;
  std::string  name;
  dictionary_t dictionary;
  telephone_list(const telephone_list& rhs);
  telephone_list& operator=(const telephone_list& rhs);
};
} // namespace naive
#endif
```

C++ allows developers to separate the implementation code for member functions from the class definition, but there is no comparable support for separating the member data that implements an object's state (or, for that matter, for separating the declarations of private member functions). Consequently the implementation detail exposed in a class's definition is still there as background noise, providing users with an added distraction. The definition of a class is typically encumbered with implementation "noise" that is of no interest to the user and is inaccessible to the client code written by that user: the naïve implementation shows this with `MyDict`, `myName` and `dict`.

Problem 2: Reducing Physical Coupling

The purpose of defining a class in a header file is for the definition of that class to be included in any translation units that define the client code for that class. If classes are designed in a naïve manner this leads to compilation dependencies upon details of the implementation that are not only inaccessible to the client code but also (in most cases) do not affect it in any way.

These compilation dependencies are undesirable for two reasons:

- Additional header file inclusions may be required to compile the class definition. This increases the size of all dependent translation units. The "Naïve Implementation" example needs `<map>` even though `std::map` is not used in the public interface - if this were a user header with its own inclusions these too might be "bloat".
- When changes are made to implementation elements in the header - even without affecting the interface - the client code must be recompiled. (When using shared libraries this can also introduce binary incompatibilities between versions.) Should the example implementation change the choice of using `MyDict`, `myName` or `dict` this affects all client code.

In a medium to large system the effect of these compilation dependencies can multiply to an extent that causes excessive and problematic build times.

Problem 3: Allowing Customised Implementations

Library code frequently defines points of customisation for user code to exploit. One of the ways to do this is to specify an interface as a class and allow the user code to supply objects that conform to this interface.

Such a library is typically compiled before the user code is written. In this case the library contains the "client code" and for this to have compilation dependencies on the implementation would be problematic.

Clearly, the naïve implementation makes no provision for alternative implementations.

The Idioms

We present the best known idioms for implementation hiding along with some comments in italics.

Each of these idioms can have advantages and these need to be understood when choosing between them.

Cheshire Cat

A private "representation" class is written that embodies the same functionality and interface as the naïve class - however, unlike the naïve version, this is defined and implemented entirely within the implementation file. The public interface of the class published in the header is unchanged, but the private implementation details are reduced to a single member variable that points to an instance of the "representation" class, each of its member functions forwards to the corresponding function of the "representation" class.

The term "Cheshire Cat" (see [Murray1993]) is an old one, coined by John Carollan over a decade ago. Sadly it seems to have disappeared from use in contemporary C++ literature. It appears described as a special case of the Bridge pattern in "Design Patterns" [GOF95], but the name "Cheshire Cat" is not mentioned. Herb Sutter (in [Sut00]) discusses it under the name "Pimpl idiom", but considers it only from the perspective if its use in reducing physical dependencies. It has also been called "Compilation Firewall".

Cheshire Cat requires "boilerplate" code in the form of forwarding functions (see "Cheshire Cat Implementation" sidebar below) that are tedious to write and (if the compiler fails to optimise them away) can introduce a slight performance hit. It also requires care with the copy semantics (although it is possible to factor this out into a smart pointer - see Griffiths99). As the relationship between the public and implementation classes is not explicit it can cause maintenance issues.

Cheshire Cat

```
// cheshire_cat.h  Cheshire Cat -
//           implementation hiding example

#ifndef INCLUDED_CHESHIRE_CAT_H
#define INCLUDED_CHESHIRE_CAT_H
#include <string>
#include <utility>

namespace cheshire_cat {
class telephone_list {
public:
  telephone_list(const std::string& name);
  ~telephone_list();

  std::string get_name() const;

  std::pair<bool, std::string>
  get_number(const std::string& person) const;

  telephone_list&
  add_entry(const std::string& name,
            const std::string& number);
private:
  class telephone_list_implementation;
  telephone_list_implementation* rep;
  telephone_list(const telephone_list& rhs);
  telephone_list& operator=(
                 const telephone_list& rhs);
};
} // namespace cheshire_cat
#endif
```

Delegation

One or more areas of the class functionality are factored out from the naïve implementation into separate helper classes. The class published in the header holds a pointer to each of these classes and delegates responsibility for the corresponding functionality by forwarding the corresponding operations. This is similar to Cheshire Cat, except that some implementation may remain exposed (like myName in the example) and there may be more than one helper class. (The helper classes may be defined and implemented in the implementation file - as in the sample code - or placed in a header file and made available for use by other code.)

Delegation is attractive where there is a distinct area of functionality that can be factored out or shared with another class. In maintenance and performance terms it is similar to Cheshire Cat.

Delegation

```
// delegation.h - Delegation implementation hiding
//   example.

#ifndef INCLUDED_DELEGATION_H
#define INCLUDED_DELEGATION_H
#include <string>
#include <utility>

namespace delegation {
class telephone_list {
public:
  telephone_list(const std::string& name);
  ~telephone_list();

  std::string get_name() const;

  std::pair<bool, std::string>
  get_number(const std::string& person) const;

  telephone_list&
  add_entry(const std::string& name,
            const std::string& number);
private:
  std::string name;
  class dictionary;
  dictionary* lookup;
  telephone_list(const telephone_list& rhs);
  telephone_list& operator=(
               const telephone_list& rhs);
};
} // namespace delegation
#endif
```

Envelope/Letter

As with Cheshire Cat a private "representation" class is written which implements the same functionality and interface as the naïve class but is defined and implemented entirely within the implementation file. The variations from Cheshire Cat are:

- The "representation" class is derived from the public one.
- The member functions of the public class are declared `virtual` (and overridden in the implementation class).
- The class published in the header holds a pointer to what appears to be another instance of the class but, in fact, is an instance of the derived class.

This is described in some detail in Coplien's "Advanced C++ Style and Idioms" [Cope92].

Frankly Envelope/Letter confuses us - we don't see what advantage it gives over Cheshire Cat. (Maybe it is just a misguided attempt to represent the correspondence of interface and implementation functions explicitly?) But please read Coplien and make up your own mind! In performance terms each client call initiates two function calls dispatched via the v-table - so it is the slowest of the idioms. (However it is rare that the overhead of a virtual function call is significant.)

Envelope/Letter

```
// envelope_letter.h - Envelope/Letter
//   implementation hiding example.

#ifndef INCLUDED_ENVELOPE_LETTER_H
#define INCLUDED_ENVELOPE_LETTER_H
#include <string>
#include <utility>

namespace envelope_letter {
class telephone_list {
public:
  telephone_list(const std::string& name);
  virtual ~telephone_list();

  virtual std::string get_name() const;

  virtual std::pair<bool, std::string>
  get_number(const std::string& person) const;
  virtual telephone_list&
  add_entry(const std::string& name,
            const std::string& number);
protected:
  telephone_list();
private:
  telephone_list* rep;
  telephone_list(const telephone_list& rhs);
  telephone_list& operator=(
                  const telephone_list& rhs);
};
} // namespace envelope_letter
#endif
```

Interface Class

All member data is removed from the naïve class and all member functions are made pure virtual. In the implementation file a derived class is defined that implements these member functions. The derived class is not used directly by client code, which sees only a pointer to the public class.

This is described in some detail in Mark Radford's "C++ Interface Classes - An Introduction" [Radford04].

Conceptually the Interface Class idiom is the simplest of those we consider. However, it may be necessary to provide an additional component and interface in order to create instances. Interface Classes, being abstract, can not be instantiated by the client. If a derived "implementation" class implements the pure virtual member functions of the Interface Class, then the client can create instances of that class. (But making the implementation class publicly visible re-introduces noise.) Alternatively, if the implementation class is provided with the Interface Class and (presumably) buried in an implementation file, then provision of an additional instantiation mechanism - e.g. a factory function - is necessary. This is shown as a static create function in the corresponding sidebar.

As objects are dynamically allocated and accessed via pointers this solution requires the client code to manage the object lifetime. This is not a handicap where the domain understanding implies objects are to be managed by a smart pointer (or handle) but it may be significant in some cases.

Note: Interfaces may play an additional role in design to that addressed in this article - they may be used to delineate each of several roles supported by a concrete type. This allows for client code that depend only on (the interface to) the relevant role.

Interface Class

```
// interface_class.h - Interface Class
//   implementation hiding example.

#ifndef INCLUDED_INTERFACE_CLASS_H
#define INCLUDED_INTERFACE_CLASS_H
#include <string>
#include <utility>

namespace interface_class {
class telephone_list {
public:
  static telephone_list*
                 create(const std::string& name);
  virtual ~telephone_list()   {}

  virtual std::string get_name() const = 0;

  virtual std::pair<bool, std::string>
  get_number(const std::string& person) const = 0;

  virtual telephone_list&
  add_entry(const std::string& name,
            const std::string& number) = 0;
protected:
  telephone_list()   {}
  telephone_list(const telephone_list& rhs) {}
private:
  telephone_list& operator=(
               const telephone_list& rhs);
};
} // namespace interface_class
#endif
```

Non-Virtual Public Interface

All member data is removed from the naïve class, the public interface becomes non-virtual forwarding functions that delegate to corresponding private pure virtual functions. As with Interface Class the implementation file defines a derived class that implements these member functions.

The derived class is not used directly by client code, which sees only a pointer to the public class.

This is described in some detail in Sutter's "Exceptional C++ Style" [Sut04].

We had thought Non-Virtual Public Interface an idea that had been tried and discarded as introducing unjustified complexity. While the standard library uses this idiom in the iostreams design we've yet to see an implementation of the library that exploits the additional flexibility (in implementing the public functions) it offers over Interface Class. Further, there are some costs to providing this flexibility:

- *A class definition embodies the contract between code that uses and code that implements that class. By splitting the contract into (public) non-virtual usage and (private) virtual implementation parts it introduces a need to understand both and also a need to document and follow the relationship between them.*
- *There is a development and maintenance cost: because the implementation functions are private to the base class they cannot be called directly by a unit test.*
- *There is a potential performance cost: if the extra function call is not optimised away it can use additional stack space and time.*

Non-Virtual Public Interface

```
// non_virtual_public_interface.h - Non-Virtual
//   Public Interface implementation hiding example

#ifndef INCLUDED_NONVIRTUAL_PUBLIC_INTERFACE_H
#define INCLUDED_NONVIRTUAL_PUBLIC_INTERFACE_H
#include <string>
#include <utility>

namespace non_virtual_public_interface {
class telephone_list {
public:
  static telephone_list* create(
                        const std::string& name);
  virtual ~telephone_list() {}

  std::string get_name() const
    { return do_get_name(); }
  std::pair<bool, std::string>
  get_number(const std::string& person) const
    { return do_get_number(person); }

  virtual telephone_list&
  add_entry(const std::string& name,
            const std::string& number)
    { return do_add_entry(name, number); }
protected:
  telephone_list()    {}
  telephone_list(const telephone_list& rhs) {}
private:
  telephone_list& operator=(
                 const telephone_list& rhs);
  virtual std::string do_get_name() const = 0;
  virtual std::pair<bool, std::string>
  do_get_number(const std::string& person) const = 0;
  virtual telephone_list&
  do_add_entry(const std::string& name,
               const std::string& number) = 0;
};
} // namespace non_virtual_public_interface
#endif
```

Evaluating the Solutions

Problem 1: Reducing Implementation Detail Exposed to the User

All the idioms considered address this problem reasonably successfully. The only implementation detail any of these idioms expose is the mechanism by which they support the separation:

- Interface Class declares virtual functions
- Cheshire Cat exposes a pointer to the "real" implementation
- Non-Virtual Public Interface declares forwarding functions and virtual functions
- Envelope/Letter declares virtual functions and a pointer to the "real" implementation

Delegation is in a way the odd one out, because it does not by nature conceal all the implementation detail. This point is illustrated in our example implementation where the `std::string` member `myName` is visible in the definition of `TelephoneList`. Delegation reduces the implementation noise exposed to clients, but - unless all functionality is delegated to one (or more) other classes - it leaves the class still vulnerable to the problems suffered by the naïve implementation.

Cheshire Cat Implementation

```
// MCheshireCat.cpp - implementation hiding example.

#include "cheshire_cat.h"
#include <map>

namespace cheshire_cat {
// Declare the implementation class
class telephone_list::telephone_list_implementation {
public:
  telephone_list_implementation(
                           const std::string& name);
  ~telephone_list_implementation();
  std::string get_name() const;
  std::pair<bool, std::string>
  get_number(const std::string& person) const;
  void add_entry(const std::string& name,
                 const std::string& number);
private:
  typedef std::map<std::string, std::string>
                                       dictionary_t;
  std::string  name;
  dictionary_t dictionary;
};

// Implement the stubs for the wrapper class
telephone_list::telephone_list(
                            const std::string& name)
   : rep(new telephone_list_implementation(name)) {}

telephone_list::~telephone_list() { delete rep; }
```

```
std::string telephone_list::get_name() const {
  return rep->get_name();
}

std::pair<bool, std::string> telephone_list::
get_number(const std::string& person) const {
  return rep->get_number(person);
}


telephone_list& telephone_list::add_entry(
                        const std::string& name,
                        const std::string& number) {
  rep->add_entry(name, number);
  return *this;
}

// Implement the implementation class
telephone_list::telephone_list_implementation::
          telephone_list_implementation(
                            const std::string& name)
   : name(name) {}

telephone_list::telephone_list_implementation::
          ~telephone_list_implementation() {}

std::string telephone_list::
   telephone_list_implementation::get_name() const {
  return name;
}

std::pair<bool, std::string>
telephone_list::telephone_list_implementation::
       get_number(const std::string& person) const {

  dictionary_t::const_iterator i
       = dictionary.find(person);

  return(i != dictionary.end()) ?
```

```
        std::make_pair(true, (*i).second) :
        std::make_pair(true, std::string());
  }

  void telephone_list::telephone_list_implementation::
  add_entry(const std::string& name,
            const std::string& number) {
    dictionary[name] = number;
  }
  } // namespace cheshire_cat
```

Problem 2: Reducing Physical Coupling

When the principal concern is reducing compile time dependencies the size (including indirect inclusions) of the header is more significant than that of the implementation file. However, in most cases, there is very little difference between the header files required by the different idioms - in our example they all have the same includes and the file lengths are as follows:

```
$ wc *.h | sort
   62    163   1580  cheshire_cat.h
   62    184   1677  interface_class.h
   65    162   1535  naive.h
   66    163   1554  delegation.h
   66    164   1605  envelope_letter.h
   94    285   2688  non_virtual_public_interface.h
```

The lack of variation is not surprising: all of the examples have eliminated the `<map>` header file and the only substantial difference is that Non-Virtual Public Interface declares twice as many functions (having both public and private versions of each).

Problem 3: Allowing Customised Implementations

It should be noted that only Interface Class and Non-Virtual Public Interface allow user implementation - the other idioms do not publish an implementation interface.

When our principal concern is that of simplifying the task of implementing the class then the size of the implementation file is most significant:

```
$ wc interface_class.cpp /
                non_virtual_public_interface.cpp
   85   147  2013  interface_class.cpp
   89   151  2186  non_virtual_public_interface.cpp
```

There is no substantial difference in implementation cost between these approaches as they contain almost identical code.

Interface Class Implementation

```
// MAbstractBaseClass.cpp - implementation hiding
// example.
#include "interface_class.h"
#include <map>

// Declare the implementation class
namespace {
class telephone_list_implementation
   : public interface_class::telephone_list {
public:
  telephone_list_implementation(const std::string& name);
  virtual ~telephone_list_implementation();
private:
  virtual std::string get_name() const;
  virtual std::pair<bool, std::string>
  get_number(const std::string& person) const;
  virtual interface_class::telephone_list&
  add_entry(const std::string& name,
            const std::string& number);
  typedef std::map<std::string, std::string>
                                     dictionary_t;
  std::string name;
  dictionary_t dictionary;
};
} // anonymous namespace

// Implement the stubs for the base class
namespace interface_class {
telephone_list* telephone_list::create(
                          const std::string& name) {
  return new telephone_list_implementation(name);
}
} // namespace interface_class
```

```
// Implement the implementation class
namespace {
telephone_list_implementation::
telephone_list_implementation(const std::string& name)
   : name(name) {}

telephone_list_implementation::
~telephone_list_implementation() {}

std::string telephone_list_implementation::get_name() const
  { return name; }

std::pair<bool, std::string>
telephone_list_implementation::
get_number(const std::string& person) const {
  std::pair<bool, std::string> rc(false,
                                  std::string());
  dictionary_t::const_iterator i
      = dictionary.find(person);
  return(i != dictionary.end()) ?
      std::make_pair(true, (*i).second) :
      std::make_pair(true, std::string());
}

interface_class::telephone_list&
telephone_list_implementation::
add_entry(const std::string& name, const std::string& number) {
  dictionary[name] = number;
  return *this;
}
} // anonymous namespace
```

Conclusion

In scenarios where customisation of implementation needs to be supported the choice is between Interface Class and Non-Virtual Public Interface. In this case we would prefer the simplicity of Interface Class (unless we have a need for the public functions to do more work than forwarding - which leads us into the territory of TEMPLATE METHOD [GOF95]).

Sometimes we wish to develop "value based" classes - these can, for example, be used directly with the standard library containers. Only three of the idioms (Cheshire Cat, Envelope/Letter and Delegation) permit this style of class. (Using value-based classes implies that the identity of class instances is transparent - and that may not be appropriate). Of these options, Cheshire Cat is most often the appropriate choice - although Delegation may be appropriate if it allows common functionality to be factored out.

There are many occasions where user customisation of implementation is not required, and the identity of instances of the class is important. In these circumstances it is reasonable to expect client code to manage object lifetime explicitly (e.g. by using a smart pointer). Both Interface Class and Cheshire Cat are reasonable choices here. Interface Class is simpler, but where a strong separation of interface and implementation is required Cheshire Cat may be preferred.

Acknowledgments

Thanks to Tim Penhey and Phil Bass for commenting on drafts of this article.

References

[WEB05] http://www.octopull.demon.co.uk/c++/ implementation_hiding.html (http://www.octopull.demon.co.uk/c++/%20implementation_hiding.html)

[Cope92] J. Coplien. *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, 1992

[Murray1993] Robert B Murray, *C++ Strategies and Tactics*, Addison-Wesley, 1993.

[Sut00] Herb Sutter. *Exceptional C++*, Addison-Wesley, 2000

[Griffiths99] http://www.octopull.demon.co.uk/c++/ TheGrin.html (http://www.octopull.demon.co.uk/c++/%20TheGrin.html)

[Radford04] Mark Radford, "C++ Interface Classes - An Introduction", *Overload* 62, and also available from http://www.twonine.co.uk/articles/ CPPInterfaceClassesIntro.pdf (http://www.twonine.co.uk/articles/%20CPPInterfaceClassesIntro.pdf)

[Sut04] Herb Sutter. *Exceptional C++ Style*, Addison-Wesley, 2004

[GOF95] Gamma, Helm, Johnson & Vlissides. *Design Patterns*, Addison-Wesley, 1995