External Polymorphism

An Object Structural Pattern for Transparently Extending C++ Concrete Data Types

Chris Cleeland, Douglas C. Schmidt, and Tim Harrison

chris@envision.com, schmidt@cs.wustl.edu, harrison@cs.wustl.edu

Envision Solutions, St. Louis, MO 63141, (314) 878-4777

and

Department of Computer Science, Washington University

St. Louis, MO 63130, (314) 935-7538

This paper appeared in the Proceedings of the 3^{rd} Annual Pattern Languages of Programming Conference, Allerton Park, Illinois, September 4–6, 1996.

Abstract

This paper describes the External Polymorphism pattern, which allows classes that are not related by inheritance and/or have no virtual methods to be treated polymorphically. The pattern capitalizes both on C++ language features and on other basic patterns (e.g., *Adapter and Decorator) to give the appearance of polymorphic behavior on otherwise unrelated classes. This pattern has been used in a number of C++ frameworks such as ACE and the OSE class library.*

1 Intent

The External Polymorphism pattern allows classes unrelated by inheritance and/or having no virtual methods to be treated polymorphically. Thus, these unrelated classes can be treated in a common manner by software that uses them.

2 Context

Writing applications using reusable class libraries in C++ can be frustrating. Unless the original developer anticipated and covered all possible use-cases in the design, it's likely that at some point you'll run into a class that *almost* what you want. A common solution is to apply one of the various adapting patterns (such as Adapter or Decorator [1]).

Combining classes like this within an application can be even more frustrating. Consider the common desire to maintain a collection of object instances. This seemingly simple activity is complicated by the fact that the objects' classes herald from different sources. Some classes come from within the application itself, whereas others come from class library "A", and still others come from class library "B". There is no commonality other than the desire to collect these instances.

Consider the following example in which it is desired to collect handles to objects for debugging purposes:

```
1. // In-memory Singleton object database.
2. class Object_DB { /* ... */ };
3. MyClass foo;
4.
5. int main (void) {
6.    Lib_A_Class bar;   // Instance from library "A"
7.    Lib_B_Class* baz =
8.      new Lib_B_Class; // Instance from library "B"
9.    Object_DB::instance()->dump_all(cerr);
```

If the state of this program were dumped when reaching line 9, we might get the following output:

```
MyClass::this = 0x47c393ab,
                value_ = {-1}
Lib_A_Class::this = 0x2c49a45b,
                    name_ = ``''
Lib_B_Class::this = 0x3c48a432,
                    count_ = {0},
                    interval_ = {5}
```

which is a dump of the current state of each object.

To preserve encapsulation, the `Object_DB::dump_all` method could access the state information of the `MyClass`, `Lib_A_Class`, and `Lib_B_Class` objects by calling a `dump` method defined by each of these classes.

Implementing `Object_DB` in a statically-typed language like C++ requires the resolution of the following forces that constrain the solution:

1. *Space efficiency* – the solution must not constrain the storage layout of existing objects. In particular, classes that have no virtual methods ("concrete data types" [2]) must not be forced to add a virtual table pointer (`vptr`).
2. *Polymorphism* – all library objects must be accessed in a uniform manner.

3 Applicability

Use the External Polymorphism pattern when:

1. Your class libraries contain concrete data types that cannot inherit from a common base class that contains virtual methods; and
2. The behavior of your class libraries can be simplified significantly if you can treat all objects in a polymorphic manner.

Do not use the External Polymorphism pattern when

1. Your class libraries already contain abstract data types that inherit from common base classes and contain virtual methods; and
2. Your programming language or programming environment allows methods to be added to classes dynamically.

4 Solution

The *External Polymorphism pattern* provides an efficient and transparent way to extend concrete data types. It allows classes that unrelated by inheritance and/or having no virtual methods to be treated polymorphically. Likewise, it resolves the twin forces of object layout efficiency (*e.g.,* no `vptrs` in concrete data types) and polymorphism (*e.g.,* all library objects can be treated in a uniform way) that renders other solutions inadequate. Moreover, it can claim the conceptual elegance of a tree-based polymorphic solution (see Section 4.1.1), while maintaining the storage efficiency and performance benefits of a "brute-force" solution (see Section 4.1.2). Figure 1 shows the object model[1] for such a solution.

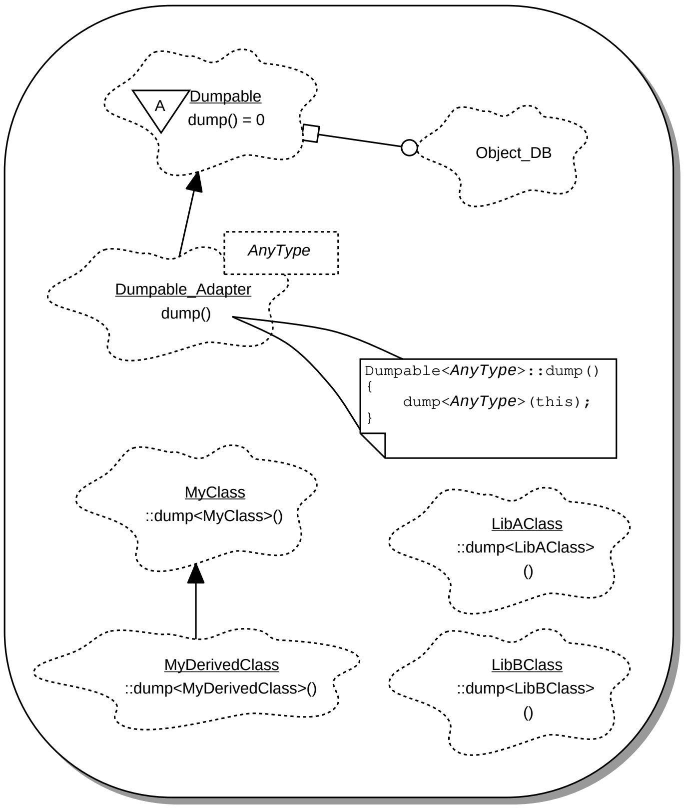

Figure 1: Object-Model of External Polymorphism Solution

The key to applying the External Polymorphism pattern is to define an abstract base class called `Dumpable` that contains a pure virtual `dump` method:

[1]Using classic *Booch* notation.

```
// Define the external polymorphic functionality.

class Dumpable
{
public:
  virtual void dump (void) = 0;
  virtual ˜Dumpable (void);
};
```

This class provides an abstract base class interface that can be used uniformly for all objects that are "dumpable." A subclass of this base class then provides a "concrete dumpable" type defined by the following template wrapper function:

```
template <class T> void
dump (const T *t)
{
  t->dump ();
}
```

This template function forwards the `dump` method call to the object. This allows the `dump` method to be used if it is defined on template class `T`. Otherwise, we can define a new `dump<>` function for a class and overload or supply missing functionality to dump the state of `T`.

The following Adapter makes any class with a `dump` method accessible through `Dumpable`'s interface:

```
template <class T>
class Dumpable_Adapter : public Dumpable
{
public:
  Dumpable_Adapter (T *t): this_ (t) {}

  virtual void dump (void) {
    // Delegate to the global dump<T> function
    dump<T> (this_);
  }

private:
  T *this_;
};
```

This solution uses C++ templates for the following reasons:

- *To ensure type-safety* – the compiler can detect type-mismatches at template instantiation time;
- *To eliminate the need for class* `T` *to inherit from a common base class* – this is useful for integrating 3^{rd} party classes, where it is not possible to modify the code;
- *To improve performance – e.g.,* by allowing the `dump<T>` template function (and the `T::dump` method) to be inlined to eliminate forwarding overhead.

Using this scheme, the `Object_DB::dump_all` method looks like:

```
void
Object_DB::dump_all (void)
{
  struct DumpDumpable {
    void operator()(const Dumpable& dump_obj) {
      dump_obj->dump();  // virtual method call
    }
  };

  for_each(this->object_table_.begin(),
           this->object_table_.end(),
           DumpDumpable());
}
```

4.1 Other Solutions

4.1.1 Tree-based Class Library Solution

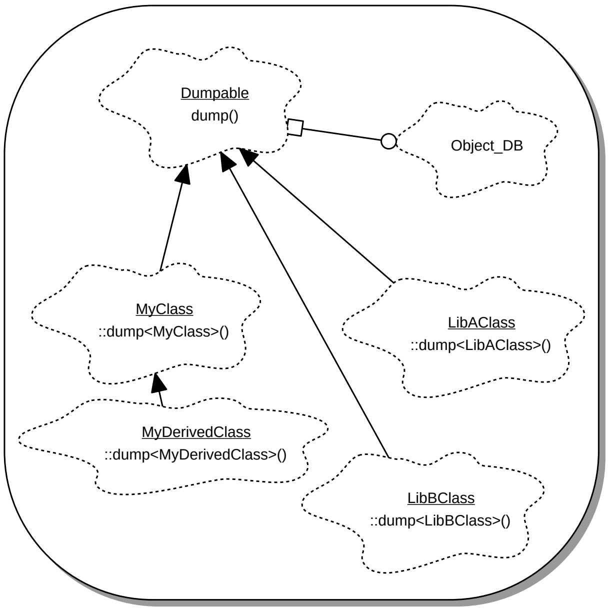

Figure 2: Object Model for Tree-based Solution

"Tree-based" class libraries have a common class (*e.g.,* class `Object`) that forms the root of all inheritance hierarchies. For these types of class libraries, the typical polymorphic solution is to add a pure virtual method called `dump` into the root class. Each subclass in the library could then override the `dump` method to display subclass-specific state (see Figure 2). With this approach, implementing `Object_DB::dump_all` is straightforward:[2]

```
void
Object_DB::dump_all (void)
{
  struct DumpObject {
    void operator()(const Object& obj) {
      obj->dump();
    }
  };
  for_each(this->object_table_.begin(),
           this->object_table_.end(),
           DumpObject());
}
```

There are several drawbacks to this approach, however:

1. It requires access to the source code: along with the ability to modify and use it, and the ability to maintain the modified code. Languages like C++ that do not allow methods to be added transparently to base classes are hard to extend in this manner. Note that other languages (such as Smalltalk and Objective-C) do not require programmers to have the source code in order to augment an interface or modify existing behaviors.

[2]Standard Template Library [3] classes are used wherever possible in examples, based on information in [4].

2. It requires all classes to inherit from a common root class: Conventional wisdom deprecates this class library design strategy in languages like C++. Inheriting from a common root object complicates integration with 3^{rd} party libraries. This approach also makes it hard to use subsets of library functionality without pulling in many unnecessary headers and library code modules. For these reasons, the Standard Template Library [3] from the ISO/ANSI C++ draft specifically avoids inheriting from a single root class.

3. It may require changes to storage layout: For C++ libraries, all objects must contain `vptrs` in their storage layout. This extra `vptr` may not be feasible for class libraries that contain "concrete data types" (such as classes for complex numbers, stacks and queues, IPC interfaces, etc.). The complicating factor for concrete data types is that they do not contain any virtual methods. Since virtual methods and inheritance are the C++ language mechanisms that support polymorphism, a concrete data type is by definition precluded from using those mechanisms to specialize the `dump` method.

Concrete data types are commonly used in C++ libraries like STL in order to enhance:

- *Performance – e.g.,* all method dispatching is static rather than dynamic (static dispatching also enables method inlining);
- *Storage efficiency – e.g.,* some objects cannot afford the space required for a virtual pointer for each instance;
- *Storage compatibility – e.g.,* ensure object layouts are compatible with C;
- *Flexibility – e.g.,* to facilitate the placement of concrete data objects in shared memory.

Therefore, for libraries that have concrete data types, it may not feasible to implement `Object_DB` by using a common root class.

Of the three drawbacks described above, the first two are relatively independent of the programming language. The third drawback is specific to C++.

4.1.2 Static Type Encoding Solution (Brute-Force)

One approach to avoid the drawbacks with the common-ancestor design is to modify the interface of `Object_DB`. The revised approach modeled in Figure 3 and shown below explicitly allows objects of each different type in the class library to register and remove themselves, as follows:

```
class Object_DB
{
public:
  void register_MyClass (MyClass*);
  void register_Lib_A_Class (Lib_A_Class*);
  void register_Lib_B_Class (Lib_B_Class*);
  // ...

private:
  list<MyClass> MyClass_table_;
  list<Lib_A_Class> Lib_A_Class_table_;
  list<Lib_B_Class> Lib_B_Class_table_;
  // ...
};
```

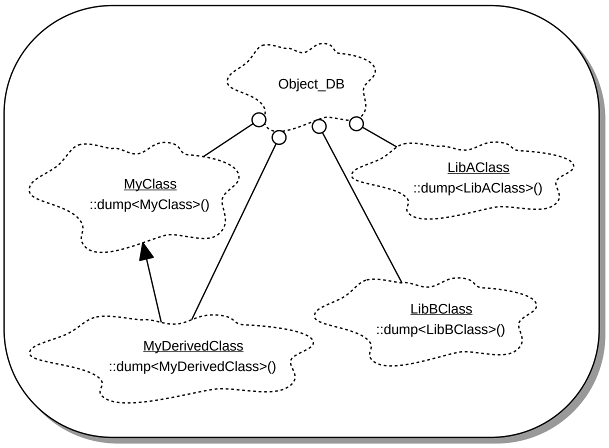

Figure 3: Object Model for "Brute-Force" Solution

In this scheme the `Object_DB::dump_all` method could be written as follows:

```
void
Object_DB::dump_all ()
{
  template <class T>
  struct Dump {
    void operator()(const T& t) {
      t->dump(); // virtual method call
    }
  };

  for_each(MyClass_table_.begin(),
           MyClass_table_.end(),
           Dump<MyClass>());

  for_each(Lib_A_Class_table_.begin(),
           Lib_A_Class_table_.end(),
           Dump<Lib_A_Class>());

  for_each(Lib_B_Class_table_.begin(),
           Lib_B_Class_.end(),
           Dump<Lib_B_Class>());

  // ...
}
```

Although it eliminates the need for a common ancestor, this brute-force approach of enumerating all types in the system is clearly tedious and fragile. Thus, by eliminating the common ancestor used by the tree-based solution, a new set of problems arise:

- Any time a class is added or removed from the library the interface and implementation of `Object_DB` must change. Considerable effort is required to maintain this scheme for large class libraries that evolve over time.
- This approach is potentially error-prone if a developer forgets to add the necessary class-specific `dump` code to the `Object_DB::dump_all` method.
- This does not simplify integrating separately developed libraries because `Object_DB` must be re-written for each combination of libraries.

4.2 Solution Comparison

The External Polymorphism borrows features of each of the other solutions presented. Looking again at Figure 1, notice that it is really a combination of the best aspects of the strategies discussed in Sections 4.1.1 and 4.1.2. Using the External Polymorphism pattern, `Dumpable` and `Dumpable_Adapter` combine to form the Tree model's `Dumpable` (see Figure 2). The template function `::dump`<*AnyType*> (shown in Figure 1 as a globally-scoped member function on each class) replaces the overloading of the virtual `dump` in the Tree model, thus eliminating the `vtbl`.

Similarities even extend to the code. In the External Polymorphism solution, the `Object_DB::dump_all` method looks almost identical to the polymorphic one shown in Section 4.1.1. The key difference is that instead of iterating over a collection of `Object*`'s, this new scheme iterates over a collection of `Dumpable*`'s. We can now treat all objects uniformly through a common ancestor (`Dumpable`) without forcing objects to inherit from a single root class. Essentially, the `vptr` that would have been stored in the target object is moved into the `Dumpable` object. The primary benefit is that the flexibility provided by a `vptr` can be added transparently without changing the storage layout of the original objects.

5 Structure and Participants

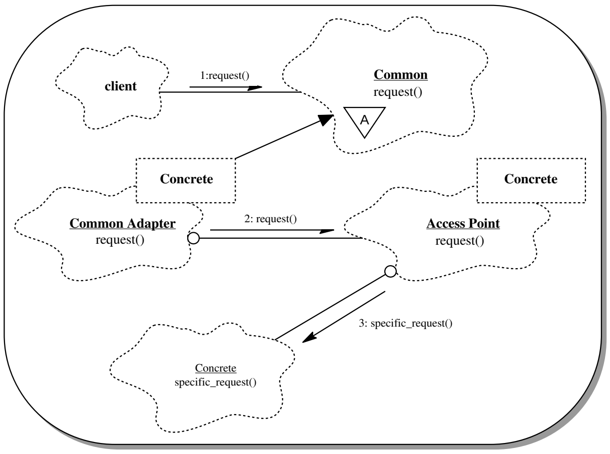

Figure 4: Structure of Participants in the External Polymorphism Pattern

The following describes the roles of the participants illustrated in Figure 4.

- **Common**
  - This abstract base class defines an abstract interface that defines the common pure virtual `request` method(s) that will be treated polymorphically by clients.

- **Common_Adapter<Concrete>**
  - This template subclass of `Dumpable` implements the pure virtual `request` method(s) defined in the `Common` base class. A typical implementation will simply forward the virtual call to the `specific_request` method in the parameterized class. If the signatures of methods in the `Concrete` class don't match those of the `Common` it may be necessary to use the Adapter pattern[1] to make them conform.
- **Access Point**
  - The template function forwards requests to the object. In some cases, *e.g.*, where the signature of `specific_request` is consistent, this feature may not be needed. However, if `specific_request` has different signatures within several `Concrete` classes, the Access Point can be used to insulate such differences from the `Common_Adapter`.
- **Concrete**
  - `Concrete` classes define `specific_request` methods that perform the desired tasks. Although `Concrete` classes are not related by inheritance, the External Polymorphism pattern allows you to treat them all or some of their methods polymorphically.
- **Collection**
  - The `Collection` maintains a table of all the `Common` objects that are currently active in the program. This table can be iterated over to "polymorphically" apply operations to all `Common` objects (*e.g.,* to dump them).

6 Collaborations

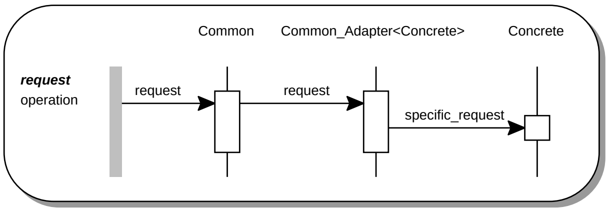

Figure 5: Interaction Diagram for Collaborators in External Polymorphism Pattern

The External Polymorphism pattern is typically used by having a function call a virtual `request` method(s) through a polymorphic `Common*`. Each of those methods, in turn, forwards to the corresponding `specific_request` method of the `Concrete` class via the `Common_Adapter`. Figure 5 shows an interaction iiagram for this collaboration.

7 Consequences

The External Polymorphism pattern has the following benefits:

- *Transparent* – Classes that were not originally designed to work together can be extended relatively transparently so they can be treated polymorphically. In particularly, the object layouts need not be changed by adding virtual pointers.
- *Flexible* – It's possible to polymorphically extend non-extensible data types such as `int` or `double` when the pattern is implemented in a language supporting parameterized types (*e.g.,* C++ templates).
- *Peripheral* – Because the pattern establishes itself on the fringes of existing code or "formal" classes, it's easy to use conditional compilation in order to remove all trace of this pattern. This feature is particularly useful for systems that use this pattern only for debugging purposes.

This pattern has the following drawbacks:

- *Instability* – The methods in the `Common` and `Common_Adapter` must track changes to methods in the `Concrete` classes.
- *Obtrusive* – It may be necessary to modify the source code of existing library classes in order to insert/remove pointers to `Common` classes.
- *Inefficient* – Extra overhead due to multiple forwarding from virtual methods in the `Common_Adapter` object to the corresponding methods in the `Concrete` object. However, using inline methods for the `Concrete` class will reduce this overhead to a single virtual method dispatch.
- *Inconsistent* – Externally Polymorphic methods are not accessible through pointers to the "polymorphized" classes, *e.g.,* in the object model in Figure 1 it is impossible to access `dump` through a pointer to `MyClass`. In addition, it is not possible to access other methods from the "polymorphized" classes through a pointer to `Dumpable_Adapter`.

8 Implementation

8.1 Implementation Steps

This section describes how to implement the External Polymorphism pattern by factoring behaviors into an abstract base class, implementing those behaviors in a descendant concrete class, and then doing the following:

1. Identify common polymorphic functionality and define it in an abstract base class: The key to polymorphic behavior is a common ancestor. In typical situations, when polymorphic behavior is desired, inheritance is used. However, it is not always possible, or desirable, to use the implementation language's inheritance to achieve polymorphism.[3] In these cases, the External Polymorphism pattern can be applied.

In either situation, one must first determine the desired shared behaviors and factor them into an abstract base class.[4] This class simply specifies an *interface* or a signature for the behaviors, not an implementation, as follows:

```
class Polymorphic_Object
{
public:
  virtual void operation1() = 0;
  virtual void operation2() = 0;
  ...
}
```

2. Define an access point for each behavior method: The abstract base defined in Step #8.1 defines the signatures of the behaviors; actual implementation of the behavior will differ (as one might expect) from concrete class to concrete class. Also, names of the interfaces to actual implementations may differ. In all cases, access to the implementation of each shared behavior is provided through a template wrapper function, such as

```
template <class T> void
operation1 (const T *t)
{
  t->operation_impl (...someargs...);
}
```

which provides a generic, default access point to an implementation named `operation_impl`. Likewise, the approach would be done for `operation2`, and any other shared behaviors defined in the `Polymorphic_Object` class.

Names of the interfaces may differ as well. In situations where `operation_impl` is not the correct interface name for some class `T`, a special-case access point can be provided. Consider a class `T1` implementing the required functionality through an interface named *some_impl*. The special-case access point would be defined as

```
void
operation1<T1> (const T1 *t)
{
  t->some_impl (...someargs...);
}
```

3. Define a parameterized adapter, inheriting from the abstract base: Step #8.1 defines an abstract base class to aggregate desired polymorphic behaviors. As in language-based inheritance, concrete descendant classes provide behavior implementation. In the External Polymorphism pattern, a concrete, parameterized adapter serves this purpose.

The parameterized adapter specifies an implementations for each interface defined in `Polymorphic_Object`. Each implementation calls the corresponding access point defined in Step #8.1, delegating to the access point the task of calling the true implementation.

The adapter for `Polymorphic_Object` might be written as

```
template <class T>
class Polymorphic_Adapter
  : public Polymorphic_Object
{
public:
  Polymorphic_Adapter (T *t) : this_(t) { }

  virtual void operation1 (void) {
    // delegate!
    operation1<T> (this_);
  }

  virtual void operation2 (void) {
    // delegate!
    operation2<T> (this_);
  }

  ...

private:
  // Insure that this_ is always set.
  Polymorphic_Adapter();
  T *this_;
}
```

4. Change application to reference through the abstract base: All facilities are now in place for the application to treat disparate classes as if they share a common ancestor. This can be done by creating instances of `Polymorphic_Adapter` parameterized over different types `T`, and managing those instances solely through a pointer to the abstract base, `Polymorphic_Object`.

It should be noted that this really is no different from managing concrete descendants in "normal" inheritance/polymorphism, except for the parameterization and additional layer of indirection provided by the access point template function.

8.2 Implementation Issues

The following issues arise when implementing this pattern.

Transparency: The scheme shown in Section 4 is not entirely transparent to the concrete class `T`. In particular, the `MyClass`'s constructor and destructor must be revised slightly to register and de-register instances with `Object_DB`, as follows:

```
MyClass::MyClass (void)
{
  Object_DB::instance ()->register_object
    (new Dumpable_Adapter<MyClass> (this));
  // ...
}

MyClass::~MyClass (void)
{
  // ...
  Object_DB::instance ()->remove_object
    ((void*) this);
}
```

[3] For instance, in C++, polymorphic behavior generally requires addition of a `vptr` a class' internal data structure.

[4] In some cases it may be desirable to define more than one abstract class, grouping related behaviors by class.

Therefore, this solution isn't suitable for transparently registering objects in binary-only libraries.

Note, however, that the changes shown above don't require the MyClass to inherit from a common class, nor do they change the storage layout of MyClass instances. Moreover, it's possible to use macros to conditionally include this feature at compile-time, as follows:

```
#if defined (DEBUGGING)
#define REGISTER_OBJECT(CLASS) \
  Object_DB::instance ()->register_object \
    (new Dumpable_Adapter<CLASS> (this))
#define REMOVE_OBJECT \
  Object_DB::instance ()->remove_object \
    ((void*) this)
#else
#define REGISTER_OBJECT(CLASS)
#define REMOVE_OBJECT
#endif /* DEBUGGING */

MyClass::MyClass (void)
{
  REGISTER_OBJECT (MyClass);
  // ...
}

MyClass::~MyClass (void)
{
  //...
  REMOVE_OBJECT (MyClass);
}
```

9 Sample Code

The following code was adapted[5] from the ACE framework, which is an object-oriented toolkit for developing communication software [5]. This code illustrates how to use the External Polymorphism pattern to implement a mechanism that registers all live ACE objects with a central in-memory object database that can dump the state of all live ACE objects (*e.g.,* from within a debugger).

There are several interesting aspects to this design:

- It uses the External Polymorphism pattern to avoid having to derive all ACE classes from a common base class that has virtual methods (this is crucial to avoid unnecessary overhead). In addition, there is no additional space added to ACE objects (this is crucial to maintain binary layout compatibility).
- This mechanism can be conditionally compiled to completely disable this feature entirely. Moreover, by using macros there are relatively few changes to ACE code.
- This mechanism copes with single-inheritance hierarchies of dumpable classes. In such cases we typically want only one dump, corresponding to the most derived instance.[6] Note, however, that this scheme doesn't generalize to work with multiple-inheritance or virtual base classes.

[5]The original code does not utilize STL in its operations.

[6]Thanks to Christian Millour for illustrating how to do this.

9.1 The Dumpable Class

The Dumpable class defines a uniform interface for all object dumping:

```
class Dumpable
{
friend class Object_DB;
friend class Dumpable_Ptr;
public:
  Dumpable (const void *);

  // This pure virtual method must be
  // filled in by a subclass.
  virtual void dump (void) const = 0;

protected:
  virtual ~Dumpable (void);

private:
  const void *this_;
};
```

The implementations are very simple:

```
Dumpable::~Dumpable (void) {}

Dumpable::Dumpable (const void *this_ptr)
  : this_ (this_ptr)
{
}
```

9.2 The Dumpable_Ptr Class

The Dumpable_Ptr is a smart pointer stored in the in-memory object database Object_DB. The pointee (if any) is deleted when reassigned.

```
class Dumpable_Ptr
{
public:
  Dumpable_Ptr (const Dumpable *dumper = 0);

  // Smart pointer delegation method.
  const Dumpable *operator->() const;

  // Assignment operator.
  void operator= (const Dumpable *dumper) const;

private:
  // Points to the actual Dumpable.
  const Dumpable *dumper_;
};
```

The Dumpable_Ptr is defined to cope with hierarchies of dumpable classes. In such cases we typically want only one dump, corresponding to the most derived instance. To achieve this, the handle registered for the subobject corresponding to the base class is destroyed (hence on destruction of the subobject its handle won't exist any more and we'll have to check for that).

The Dumpable_Ptr methods are implemented below:

```
Dumpable_Ptr::Dumpable_Ptr (const Dumpable *dumper)
  : dumper_ (dumper)
{
}

const Dumpable *
Dumpable_Ptr::operator->() const
{
```

```
  return this->dumper_;
}

void
Dumpable_Ptr::operator= (const Dumpable *dumper) const
{
  if (this->dumper_ != dumper) {
    delete (Dumpable_Ptr*) this->dumper_;
    ((Dumpable_Ptr*) this)->dumper_ = dumper;
  }
}
```

9.3 The Object Database (Object_DB) Class

The `Object_DB` class is the object database (Object_DB) that keeps track of all live ACE objects.

```
class Object_DB
{
public:
  // Iterates through the entire set of
  // registered objects and dumps their state.
  void dump_objects (void);

  // Add the tuple <dumper, this_> to
  // the list of registered ACE objects.
  void register_object
    (const Dumpable *dumper);

  // Use 'this_' to locate and remove
  // the associated 'dumper' from the
  // list of registered ACE objects.
  void remove_object (const void *this_);

  static Object_DB *Object_DB::instance (void);

private:
  static Object_DB *instance_;
  // Singleton instance of this class.

  // Ensure we have a Singleton...
  Object_DB (void);

  struct Tuple
  {
    // Pointer to the register C++ object.
    const void *this_;

    // Smart pointer to the Dumpable
    // object associated with this_.
    const Dumpable_Ptr dumper_;
  };

  typedef vector<Tuple> TupleVector;
  TupleVector object_table_;
};
```

The `Object_DB` methods can be implemented as follows:

```
Object_DB *
Object_DB::instance (void)
{
  if (Object_DB::instance_ == 0)
    Object_DB::instance_ = new Object_DB;
  return Object_DB;
}

void
Object_DB::dump_objects (void)
{
  struct DumpTuple {
    bool operator()(const Tuple& t) {
      t.dumper_->dump();
    }
  };
  for_each(this->object_table_.begin(),
           this->object_table_.end(),
           DumpTuple());
}

// Predicate function object to determine
// if the 'this_' member in two Tuples is
// equal.  This will be useful throughout.
struct thisMemberEqual :
    public binary_function<Tuple, Tuple, bool> {
  bool operator()(const Tuple& t1,
                  const Tuple& t2) const {
    return t1.this_ == t2.this_;
  }
};

// This method registers a new <dumper>.  It
// detects duplicates and simply overwrites them.
void
Object_DB::register_object (const Dumpable *dumper)
{
  TupleVector::iterator slot;

  slot = find_if(this->object_table_.begin(),
                 this->object_table_.end(),
                 bind2nd(thisMemberEqual(), dumper));

  if (slot == this->object_table_.end())
    this->object_table_.push_back(*dumper);
  else
    // Silently replace the duplicate;
    *slot = *dumper;
}

void
Object_DB::remove_object (const void *this_ptr)
{
  Dumpable d(this_ptr);

  (void) remove_if(this->object_table_.begin(),
                   this->object_table_.end(),
                   bind2nd(thisMemberEqual(), d));
}

Object_DB *Object_DB::instance_ = 0;
```

9.4 The Dumpable_Adapter Class

This class inherits the interface of the abstract `Dumpable` class and is instantiated with the implementation of the concrete component class `Concrete`. This design is similar to the Adapter and Decorator patterns [1]. Note that class `Concrete` need not inherit from a common class since Dumpable provides the uniform virtual interface.

```
template <class Concrete>
class Dumpable_Adapter : public Dumpable
{
public:
  Dumpable_Adapter (const Concrete *t);

  // Concrete dump method (simply delegates to
  // the <dump> method of <class Concrete>.
  virtual void dump (void) const;

  // Delegate to methods in the Concrete class.
  Concrete *operator->();

private:
  // Pointer to <this> of <class Concrete>
  const Concrete *this_;
};
```

The `Dumpable_Adapter` methods are implemented as follows:

```
template <class Concrete>
Dumpable_Adapter<Concrete>::Dumpable_Adapter
  (const Concrete *t)
  : this_ (t), Dumpable ((const void*) t)
{
}

template <class Concrete> Concrete *
Dumpable_Adapter<Concrete>::operator->()
{
  return (Concrete*) this->this_;
}

template <class Concrete> void
Dumpable_Adapter<Concrete>::dump (void) const
{
  this->this_->dump<Concrete>(this_);
}
```

The Access Point for the dump method can simply be defined as follows:

```
template <class Concrete> void
dump<Concrete>(const Concrete* t)
{
  t->dump();
}
```

The following are some useful macros for conditionally compiling External Polymorphism into an application or framework:

```
#if defined (DEBUGGING)
#define REGISTER_OBJECT(CLASS) \
        Object_DB::instance ()->register_object
          (new Dumpable_Adapter<CLASS> (this));
#define REMOVE_OBJECT \
        Object_DB::instance ()->remove_object
          ((void*) this);
#else
#define REGISTER_OBJECT(CLASS)
#define REMOVE_OBJECT
#endif /* DEBUGGING */
```

9.5 The Use Case

The following code illustrates how the Dumpable mechanisms are integrated into ACE components like the SOCK_Acceptor and SOCK_Stream.

```
class SOCK
{
public:
  SOCK (void) { REGISTER_OBJECT (SOCK); }
  ~SOCK (void) { REMOVE_OBJECT; }

  void dump (void) const {
    cerr << "hello from SOCK = "
         << this << endl;
  }

  // ...
};

class SOCK_Acceptor : public SOCK
{
public:
  SOCK_Acceptor (void) {
    REGISTER_OBJECT (SOCK_Acceptor);
  }
  ~SOCK_Acceptor (void) { REMOVE_OBJECT; }

  void dump (void) const {
    cerr << "hello from SOCK_Acceptor = "
         << this << endl;
  }

  // ...
};

class SOCK_Stream : public SOCK
{
public:
  SOCK_Stream (void) {
    REGISTER_OBJECT (SOCK_Stream);
  }
  ~SOCK_Stream (void) { REMOVE_OBJECT; }

  void dump (void) const {
    cerr << "hello from SOCK_Stream = "
         << this << endl;
  }

  // ...
};

int
main (void)
{
  SOCK sock;
  // Note that the SOCK superclass is *not*
  // printed for SOCK_Stream or SOCK_Acceptor.
  // because of the smart pointer Dumpable_Ptr.
  SOCK_Stream stream;
  SOCK_Acceptor acceptor;
  Object_DB::instance ()->dump_objects ();
  {
    SOCK sock;
    // Note that the SOCK superclass is *not*
    // printed for SOCK_Stream or SOCK_Acceptor.
    SOCK_Stream stream;
    SOCK_Acceptor acceptor;
    Object_DB::instance ()->dump_objects ();
  }
  Object_DB::instance ()->dump_objects ();
  return 0;
}
```

10 Variants

The Object_DB that maintains the live objects can be implemented using the GoF Command Pattern. In this case, the Object_DB::register_object method is implemented by "attaching" a new Command. This Command contains the object and its dump method. When an Object_DB::dump_all is invoked all the Commands are "executed." This solution allows the Command executor to iterate through a collection of unrelated objects with no vtables and pick out the right method for each one.

For example, assume that Object_DB had a Command_List, as follows:

```
class Object_DB
{
public:
  void register_object (Command_Base *base) {
    dumpables_.attach (base);
  }

  void dump_all (void) {
    dumpables_.execute ();
  }
  // ...

private:
  // List of Commands_Base *'s.
```

```
  Command_List dumpables_;
};
```

Individual objects could be registered as follows:

```
SOCK_Stream *ss = new SOCK_Stream;
SOCK_Acceptor *sa = new SOCK_Acceptor;

Object_DB::register_object
  (new Command0<SOCK_Stream>
    (ss, &SOCK_Stream::dump));

Object_DB::register_object
  (new Command0<SOCK_Acceptor>
    (sa, &SOCK_Acceptor::dump));
```

This implementation is more flexible than the one shown in Section 9 since it allows the other methods besides `dump` to be invoked.

11 Known Uses

- This pattern has been used in the ACE framework to allow all ACE objects to be registered with a Singleton in-memory "object database." This database stores the state of all live ACE objects and can be used by debugger to dump this state. Since many ACE classes are concrete data types it was not possible to have them inherit from a common root base class containing virtual methods.

- This pattern has been used in custom commercial projects where code libraries from disparate sources were required to have a more common, polymorphic interface. The implementation of the pattern presented a united interface to classes from a locally-developed library, the ACE library, and various other "commercial" libraries.

- The idea for the "access point" (see Section 8.1) came from usage in the OSE class library.[7]. In OSE, template functions are used to define collating algorithms for ordered lists, etc.

12 Related Patterns

This pattern is similar to the Decorator and Adapter patterns from the Gang of Four (GoF) design patterns catalog [1]. The Decorator pattern dynamically extends an object transparently without using subclassing. When a client uses a Decorated object it thinks it's operating on the actual object, when in fact it operates on the Decorator. The Adapter pattern converts the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

There are several differences between these two GoF patterns and the External Polymorphism pattern. The Decorator pattern assumes that the classes it adorns are already abstract (*i.e.,* they have virtual methods, which are overridden by the Decorator). In contrast, External Polymorphism adds polymorphism to concrete classes (*i.e.,* classes without virtual methods). In addition, since the Decorator is derived from the class it adorns, it must define all the methods it inherits. In contrast, the `Common_Adapter` class in the External Polymorphism pattern need only define the methods in the Concrete class it wants to treat polymorphically.

The External Polymorphism pattern has the same basic structure as the object form of the GoF Adapter pattern. Just as the Adapter pattern's `Adapter` adapts an `Adaptee` class to the `Target` interface, the External Polymorphism pattern's `Common_Adapter` class adapts a `Concrete` class to the `Common` interface. One difference is that the implementation of the External Polymorphism pattern shown above uses parameterized types rather than mixin private inheritance to compose the `Adaptee` with the `Adapter`. The intents of the two patterns also differ. The goal of the External Polymorphism pattern is to avoid changing the storage layout of concrete data types, yet still be able to treat all objects of these types polymorphically in some circumstances. Although this usage is a logical extension of the Adapter's intent, the scope of the Adapter focuses more on the relation between `Client`, `Target`, `Adapter`, and `Adaptee`, rather than on the relation between `Clients` and *all* `Adaptees`.

Acknowledgements

Many people have given comments on earlier versions of this paper including: Alex Maclinovsky, Ralph Johnson, Jody Hagins, Christian Millour, Phil Brooks, and Bill Hess. In addition, many thanks to David Delano, who was the paper's "shepherd" for PLoP '96.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[2] Bjarne Stroustrup, *The C++ Programming Language, 2^{nd} Edition*. Addison-Wesley, 1991.

[3] A. Stepanov and M. Lee, "The Standard Template Library," Tech. Rep. HPL-94-34, Hewlett-Packard Laboratories, April 1994.

[4] D. L. Musser and A. Saini, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, 1995.

[5] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the 6^{th} USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[7] The OSE class library is written and distributed by Graham Dumpleton. Further information can be found at `http://www.dscpl.com.au/`