External Polymorphism

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

Chris Cleeland and Douglas C. Schmidt
chris@cs.wustl.edu and schmidt@cs.wustl.edu
Department of Computer Science, Washington University
St. Louis, MO 63130, (314) 935-7538

This paper appeared in the September 1998 issue of the C++ Report magazine. An earlier subset of this paper appeared as a chapter in the book "Pattern Languages of Program Design 3" ISBN, edited by Robert Martin, Frank Buschmann, and Dirke Riehle published by Addison-Wesley, 1997.

Introduction

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. This pattern combines C++ language features with patterns like Adapter and Decorator [1] to give the appearance of polymorphic behavior on otherwise unrelated classes. The External Polymorphism pattern has been used in a number of C++ frameworks such as ACE [2] and the OSE class library.

This article is organized as follows. Section 1 describes the External Polymorphism pattern in much greater detail than an earlier version appearing in [3], Section 2 describes an example implementation using C++, and Section 3 presents concluding remarks.

1 The External Polymorphism Pattern

1.1 Intent

Allow classes that are not related by inheritance and/or have no virtual methods to be treated polymorphically.

1.2 Motivation

Debugging applications built using reusable class libraries can be hard. For example, when an error occurs in the library, developers often don't know the names of all the relevant objects comprising their application. This makes it hard to display the current state of these objects in a debugger or in print statements.

It is often useful, therefore, for class libraries to enable applications to dump the content of some or all objects that are "live" at any given point. In object-oriented languages like C++, live objects include (1) all global objects, (2) initialized static objects, (3) dynamically allocated objects that have not yet been freed, and (4) all automatic objects that are in valid activation records on the run-time stack of active threads.

To motivate the External Polymorphism pattern, consider the following code that uses the `SOCK_Stream`, `SOCK_Acceptor`, and `INET_Addr` library classes, which encapsulate the socket network programming interface within type-safe C++ wrappers [4]:

```
1. // In-memory Singleton object database.
2. class Object_DB { /* ... */ };
3. SOCK_Acceptor acceptor; // Global storage
4.
5. int main (void) {
6.    SOCK_Stream stream; // Automatic storage
7.    INET_Addr *addr =
8.      new INET_Addr; // Dynamic storage.
9.    Object_DB::instance ()->dump_all (cerr);
```

If the state of this program were dumped when reaching line 13, we might get the following output:

```
Sock_Stream::this = 0x47c393ab,
                    handle_ = {-1}
SOCK_Acceptor::this = 0x2c49a45b,
                      handle_ = {-1}
INET_Addr::this = 0x3c48a432,
                  port_ = {0},
                  addr_ = {0.0.0.0}
```

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

`Object_DB` is an in-memory database Singleton[1], *i.e.*, there's only one copy per-process. To preserve encapsulation, the `Object_DB::dump_all` method could access the state information of the `SOCK_Stream`, `SOCK_Acceptor`, and `INET_Addr` objects by calling a `dump` method defined by these classes. These objects register and unregister with `Object_DB` in their constructors and destructors, respectively, as illustrated below:

```
SOCK_Stream::SOCK_Stream (void)
{
  Object_DB::instance ()->
        register_object ((void *) this);
  // ...
}

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

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 having no virtual methods, *i.e.*, "concrete data types" [5], must not be forced to add a virtual table pointer (`vptr`).
2. *Polymorphism* – all library objects must be accessed in a uniform manner.

The remainder of this section describes and evaluates three solutions for implementing the `Object_DB` facility. The first two solutions exhibit several common traps and pitfalls. The third solution employs the External Polymorphism pattern to avoid the problems with the first two approaches.

1.3 Common Traps and Pitfalls

The limitations with two "obvious" ways of implementing the functionality of `Object_DB` for statically-typed object-oriented programming languages (such as C++ or Eiffel) are presented below.

1.3.1 Tree-based Class Library Solution

"Tree-based" class libraries [6] have a common class, such as 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, as shown in Figure 1. Using this approach, implementing

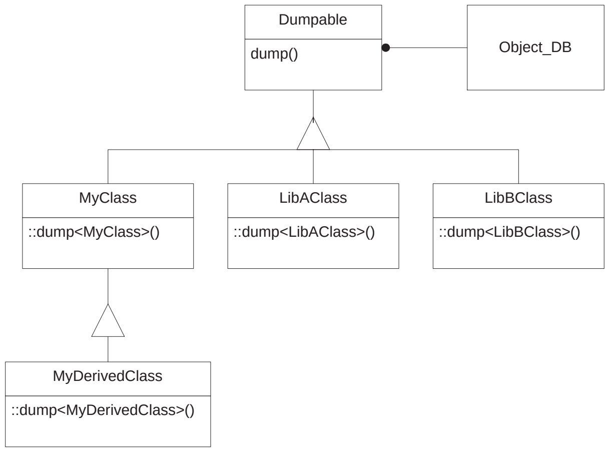

Figure 1: Object Model for Tree-based Solution

`Object_DB::dump_all` is straightforward:[1]

[1]Standard Template Library [7] classes are used wherever possible in examples, based on information in [8].

```
void
Object_DB::dump_all (void)
{
  struct DumpObject {
    void operator ()(const Object &obj) {
      obj->dump ();
    }
  };
  // Dump all the objects in the table.
  for_each (this->object_table_.begin (),
            this->object_table_.end (),
            DumpObject ());
}
```

There are several drawbacks to the tree-based solution, however:

1. It requires access to the source code: It also requires the ability to modify 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. Other OO 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. It requires all classes to inherit from a common root class: Conventional wisdom deprecates single root tree-based class library design strategy in languages like C++ [6, 7]. For instance, inheriting from a common root object complicates integration with third-party libraries. Moreover, the tree-based approach makes it hard to use subsets of library functionality without including many unnecessary headers and library code modules. For these reasons, the Standard Template Library [7] 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 with virtual methods 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, and interprocess communication (IPC) interfaces [4]. 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 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 be 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++.

1.3.2 Static Type Encoding Solution (Brute-Force)

One way to avoid the drawbacks with the tree-based class library design is to modify the interface of `Object_DB`. The revised approach is shown in Figure 2. As shown below, the brute-force approach explicitly allows objects of each different type in the class library to register and remove themselves, as follows:

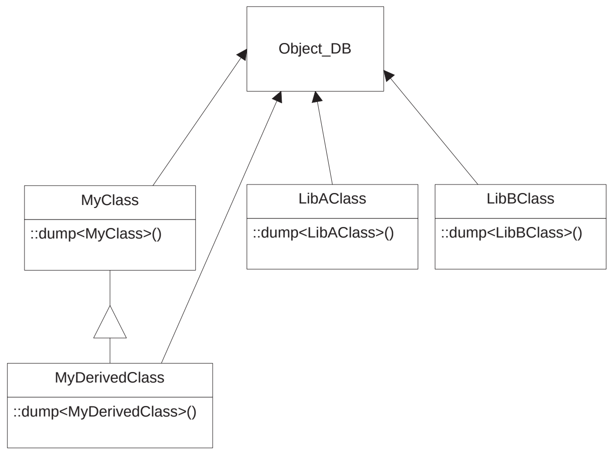

Figure 2: Object Model for “Brute-Force” Solution

```
class Object_DB
{
public:
  void register_SOCK_Stream (SOCK_Stream*);
  void register_SOCK_Acceptor (SOCK_Acceptor*);
  void register_INET_Addr (INET_Addr *);
  // ...

private:
  list<SOCK_Stream> SOCK_stream_table_;
  list<SOCK_Acceptor> SOCK_Acceptor_table_;
  list<INET_Addr> INET_Addr_table_;
  // ...
};
```

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 (SOCK_stream_table_.begin (),
            SOCK_stream_table_.end (),
            Dump<SOCK_Stream> ());

  for_each (SOCK_Acceptor_table_.begin (),
            SOCK_Acceptor_table_.end (),
            Dump<SOCK_Acceptor> ());

  for_each (INET_Addr_table_.begin (),
            INET_Addr_.end (),
            Dump<INET_Addr> ());

  // ...
}
```

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

- *Tedious maintenance* – 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.
- *Error-prone* – 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.
- *Integration difficulties* – The brute-force solution does not simplify integrating separately developed libraries because `Object_DB` must be re-written for each combination of libraries.

1.4 Solution: the External Polymorphism Pattern

A more efficient and transparent way to extend concrete data types is to use the *External Polymorphism pattern*. This pattern allows classes that are not related by inheritance and/or have no virtual methods to be treated polymorphically. It resolves the 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 rendered the previous solutions inadequate. By using this pattern, we’ll be able to reclaim the conceptual elegance of the polymorphic solution in Section 1.3.1, while still maintaining the storage efficiency and performance benefits of the solution in Section 1.3.2.

Figure 3 shows the object model for the External Polymorphism solution. Notice that it combines the best aspects of the strategies discussed in Sections 1.3.1 and 1.3.2. Using the External Polymorphism pattern, `Dumpable` and `Dumpable_Adapter` combine to form the Tree model’s `Dumpable` (see Figure 1). The template function `::dump<`*AnyType*`>` (shown in Figure 3 as a globally-scoped member function on each class) replaces the overloading of the virtual `dump` method in the Tree model, thus eliminating the `vtbl` for *AnyType*.

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

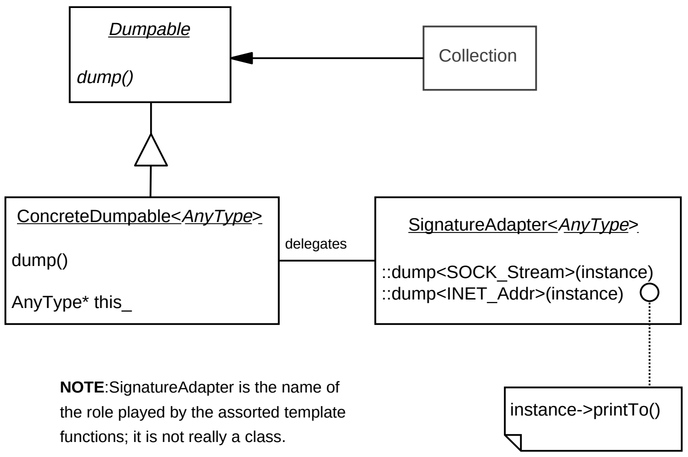
Dumpable
dump()
Collection
ConcreteDumpable<AnyType>
dump()
AnyType* this_
delegates
SignatureAdapter<AnyType>
::dump<SOCK_Stream>(instance)
::dump<INET_Addr>(instance)
instance->printTo()
NOTE:SignatureAdapter is the name of the role played by the assorted template functions; it is not really a class.

Figure 3: Object-Model of External Polymorphism Solution

```
// 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 third-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.

By applying the External Polymorphism pattern, the `Object_DB::dump_all` method looks almost identical to the original polymorphic one shown in Section 1.3.1:

```
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 ());
}
```

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 key benefit is that the flexibility provided by a `vptr` can be added transparently without changing the storage layout of the original objects.

1.5 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.

1.6 Structure and Participants

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

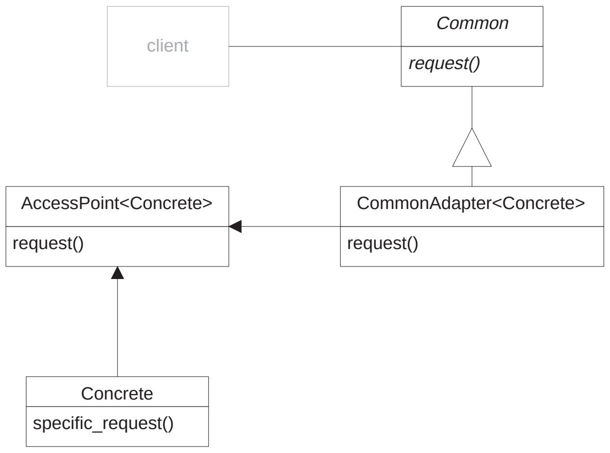

Figure 4: Structure of Participants in the External Polymorphism Pattern

Common (`Dumpable`):

- 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> (`Dumpable_Adapter`):

- 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 Method (`::dump<>`):

- 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 method can be used to insulate such differences from the `Common_Adapter`.

Concrete (`SOCK_Stream`, `SOCK_Acceptor`):

- The `Concrete` classes in this pattern define one or more `specific_request` methods that perform the desired tasks. Although `Concrete` classes are not related by inheritance, the External Polymorphism pattern make it possible to treat all or some of their methods polymorphically.

Collection (`Object_DB`):

- 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).

1.7 Collaborations

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 diagram for this collaboration.

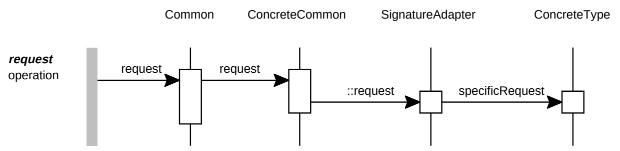

Figure 5: Interaction Diagram for Collaborators in External Polymorphism Pattern

1.8 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 otherwise 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 classes, it's easy to use conditional compilation to remove all trace of this pattern. This feature is particularly useful for frameworks that use the External Polymorphism pattern solely for debugging purposes.

However, this pattern has the following drawbacks:

Unstable: All of 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 to insert/remove pointers to `Common` classes.

Inefficient: Extra overhead is increased 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. For instance, in the object model in Figure 3 it's impossible to access `dump` through a pointer to `SOCK_Stream`. In addition, it is not possible to access other methods from the "polymorphized" classes through a pointer to `Dumpable_Adapter`.

1.9 Known Uses

The External Polymorphism pattern has been used in the following software systems:

- The ACE framework uses the External Polymorphism pattern 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.

- The External Polymorphism pattern also 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 unified interface to classes from a locally-developed library, the ACE library, and various other "commercial" libraries.

- The idea for the "access method (see Section 2.1) came from usage in the OSE class library, by Graham Dumpleton.[2] In OSE, template functions are used to define collating algorithms for ordered lists, etc.

1.10 Related Patterns

The External Polymorphism 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 `ConcreteCommon` class in the External Polymorphism pattern need only define the methods in the Concrete class it wants to treat polymorphically.

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

The External Polymorphism pattern is similar to the GoF Adapter pattern. However, there are subtle but important differences:

Intents differ: An Adapter *converts* an interface to something directly usable by a client. External Polymorphism has no intrinsic motivation to convert an interface, but rather to provide a new substrate for accessing similar functionality.

Layer vs. Peer: The External Polymorphism pattern creates an entire class hierarchy outside the scope of the concrete classes. Adapter creates new layers within the existing hierarchy.

Extension vs. Conversion: The External Polymorphism pattern extends existing interfaces so that similar functionality may be accessed polymorphically. Adapter creates a new interface.

Behavior vs. Interface: The External Polymorphism pattern concerns itself mainly with behavior rather than the names associated with certain behaviors.

Finally, the External Polymorphism pattern is similar to the *Polymorphic Actuator* pattern documented and used internally at AG Communication Systems.

2 Implementing External Polymorphism in C++

The steps and considerations necessary to implement the External Polymorphism pattern are described below.

2.1 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 performing the following steps:

1. Identify common polymorphic functionality and define it in an abstract base class: The key to polymorphic behavior is a common ancestor. Inheritance is typically used when polymorphic behavior is desired. It's not always possible or desirable, however, to use the implementation language's inheritance to achieve polymorphism. For instance, in C++, polymorphic behavior generally requires addition of a `vptr` a class' internal data structure. To avoid this, 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. This class simply specifies an *interface* for the behaviors, not an implementation, as follows:

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

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

2. Define an access method for each behavior method: The abstract base defined in Step #1 above defines the signatures of the behaviors. The actual implementation of the behavior will differ (as one might expect) from concrete class to concrete class. Likewise, 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->operation1_impl (...someargs...);
}
```

which provides a generic, default access method to an implementation named `operation_impl`. Likewise, the approach would be applied 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 method can be provided. Consider a class `T1` implementing the required functionality through an interface named *some_impl*. The special-case access method would be defined as

```
void
operation1<T1> (const T1 *t)
{
  t->some_operation1_impl (...args...);
}
```

3. Define a parameterized adapter, inheriting from the abstract base: Step #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 implementation for each interface defined in the base class `Polymorphic_Object`. Each implementation calls the corresponding access method defined in Step #2, delegating to the access method the task of calling the actual 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:
  // Make the constructor private to ensure
  // that this_ is always set.
  Polymorphic_Adapter ();

  T *this_;
}
```

4. Change the 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` that are parameterized over different types `T`, and managing those instances solely through a pointer to the abstract base, `Polymorphic_Object`.

It should be noted that the External Polymorphism pattern is really no different from managing concrete descendants in "normal" inheritance/polymorphism. The main differences is that the parameterization and additional layer of indirection is provided by the access method template function.

2.2 Implementation Considerations

The following issue arises when implementing the External Polymorphism pattern.

Transparency: The scheme shown in Section 1.4 is not entirely transparent to the concrete class `T`. In particular, the `SOCK_Stream`'s constructor and destructor must be revised slightly to register and de-register instances with `Object_DB`, as follows:

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

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

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 `SOCK_Stream` to inherit from a common class. Neither do they change the storage layout of `SOCK_Stream` 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 */

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

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

2.3 Sample Code

The following code was adapted[3] from the ACE framework, which is an object-oriented toolkit for developing communication software [2]. 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. Applications 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 with virtual methods. This design is crucial to avoid unnecessary overhead. In addition, there is no additional space added to ACE objects. This design is crucial to maintain binary layout compatibility.
- This mechanism can be conditionally compiled to completely disable External Polymorphism 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.[4] Note, however, that this scheme doesn't generalize to work with multiple-inheritance or virtual base classes.

2.4 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_;
};
```

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

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

The implementations of these methods are relatively straightforward:

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

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

2.5 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. Therefore, on destruction of the subobject its handle won't exist any more, so we'll have to check for that.

The `Dumpable_Ptr` methods are implemented below. Once again, these are not tricky:

```
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;
  }
}
```

2.6 The Object Database (Object_DB) Class

The `Object_DB` class is the Singleton object database that keeps track of all live objects. Instances must be registered with the database using the `register_object` method, and subsequently removed using `remove_object`. The entire database can be traversed and registered objects dumped using `dump_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 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_);

  // Factory method to get the singleton database
  static Object_DB *Object_DB::instance (void);

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

  // Ensure we have a Singleton (nobody
  // can create instances but this class)
  Object_DB (void);

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

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

  typedef vector<Tuple> TupleVector;

  // Holds all registered C++ objects.
  TupleVector object_table_;
};
```

The `instance` method, along with the private constructor, enforces the policy that `Object_DB` is a singleton. Note that this implementation does not protect itself against concurrent access; however, we can easily apply the *Double-Checked Locking Pattern*[9] to achieve that.

```
Object_DB *
Object_DB::instance (void)
{
  // For thread safety we would employ
  // double-checked locking, but not now.
  if (Object_DB::instance_ == 0)
    Object_DB::instance_ = new Object_DB;
  return Object_DB::instance_;
}
```

The `dump_objects` method traverses the database and calls the `dump` method on each registered instance.

```
// Dump all the live objects registered
// with the Object_DB Singleton.
void
Object_DB::dump_objects (void)
{
  // A "funcstruct" to dump what's in a tuple
  struct DumpTuple {
    bool operator ()(const Tuple &t) {
      t.dumper_->dump ();
    }
  };
  for_each (this->object_table_.begin (),
            this->object_table_.end (),
            DumpTuple ());
}
```

An object's lifecycle with respect to the database is managed by the following methods which register and remove instances from the database. An STL-style predicate function is used to compare for equality (see code comments for details).

```
// STL 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 ())
    // Reached the end and didn't find it, so append
    this->object_table_.push_back (*dumper);
  else
    // Found this already--replace
    *slot = *dumper; // Silently replace the duplicate
}

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;
```

2.7 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 critical "glue" between the external class hierarchy and the existing class hierarchy is the *access method*, which is defined for the `dump` method as follows:

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

Since it may not always be desireable to have this debugging hierarchy compiled in, we created some useful macros for conditionally compiling this implementation of 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 */
```

2.8 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;
}
```

2.9 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 the `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 can 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 2.3 since it allows the other methods besides `dump` to be invoked when iterating over the `Object_DB`.

3 Concluding Remarks

External Polymorphism is likely not destined for the design patterns "Hall of Fame." In particular, it is not as broadly applicable as the *Singleton* or *Adapter* patterns [1]. However, External Polymorphism does solve a very subtle, real-world problem encountered by developers implementing complex software in statically-typed languages like C++.

Re-use, and thus integration, typically occurs at source level. While patterns cannot change fundamental linkage styles of languages or environments from "source" to "binary," the External Polymorphism pattern enforces software integration at a different conceptual level. In particular, it encourages a component-like "black-box" style of development and integration, as opposed to a "white-box" approach [10]. Therefore, substituting one set of library components for another can be simplified. Likewise, bringing in new, externally-produced libraries is also easier.

The following analogy is offered in closing: automobiles are complex interworking systems. Many repairs or maintainence tasks can be performed by using general-purpose tools such as screwdrivers or a socket set. However, there are some automotive subsystems such as mounting a tire or performing a wheel alignment, that require the application of highly specialized tools and techniques to efficiently complete the job. In the world of design patterns, External Polymophism is definitely a special-purpose tool. Hopefully, our description of this pattern will enable you to apply it and "effectively complete the job."

Acknowledgements

Many people have given comments on earlier versions of this paper including: Alex Maclinovsky, Ralph Johnson, Jody Hagins, Christian Millour, Phil Brooks, Phil Mesnier, Bill Hess, and Jerome Zingg. The authors also wish to thank everyone from PLoP '96 who provided substantive comments on content and form for this pattern.

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

[3] C. Cleeland, D. C. Schmidt, and T. Harrison, "External Polymorphism – An Object Structural Pattern for Transparently Extending Concrete Data Types," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[4] D. C. Schmidt, "IPC_SAP: An Object-Oriented Interface to Interprocess Communication Services," *C++ Report*, vol. 4, November/December 1992.

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

[6] D. Lea, "libg++, the GNU C++ Library," in *Proceedings of the 1st C++ Conference*, (Denver, CO), pp. 243–256, USENIX, Oct. 1988.

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

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

[9] D. C. Schmidt and T. Harrison, "Double-Checked Locking – An Object Behavioral Pattern for Initializing and Accessing Thread-safe Objects Efficiently," in *The 3rd Pattern Languages of Programming Conference (Washington University technical report #WUCS-97-07)*, February 1997.

[10] H. Hueni, R. Johnson, and R. Engel, "A Framework for Network Protocol Software," in *Proceedings of OOPSLA '95*, (Austin, Texas), ACM, October 1995.