External Polymorphism

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

Chris Cleeland
chris@envision.com
Envision Solutions, St. Louis, MO 63141

Douglas C. Schmidt and Timothy H. Harrison
schmidt@cs.wustl.edu and harrison@cs.wustl.edu
Department of Computer Science
Washington University, St. Louis, Missouri, 63130

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

1 Intent

Allow C++ classes unrelated by inheritance and/or having no virtual methods to be treated polymorphically. These unrelated classes can be treated in a common manner by software that uses them.

2 Motivation

Working with C++ classes from different sources can be difficult. Often an application may wish to "project" common behavior on such classes, but is restricted by the classes' existing design. If only the class interface requires adaptation an obvious solution is to apply an object structural pattern such as Adapter or Decorator [1]. Occasionally there are more complex requirements, such as the need to change both underlying interface and implementation. In such cases, classes may need to behave as if they had a common ancestor.

For instance, consider the case where we are debugging an application constructed using classes from various C++ libraries. It would be convenient to be able to ask any instance to "dump" its state in a human-readable format to a file or console display. It would be even more convenient to gather all live class instances into a collection and iterate over that collection asking each instance to dump itself.

Since collections are homogeneous, a common base class must exist to maintain a single collection. Since classes are already designed, implemented and in use, however, modifying the inheritance tree to introduce a common base class is not an option – we may not have access to the source, either! In addition, classes in OO languages like C++ may be *concrete data types* [2], which require strict storage layouts that could be compromised by hidden pointers (such as C++'s virtual table pointer). Re-implementing these classes with a common, polymorphic, base class is not feasible.

Thus, projecting common behavior on unrelated classes requires the resolution of the following forces constraining the solution:

1. *Space efficiency* – The solution must not constrain the storage layout of existing objects. In particular, classes that have no virtual methods (*i.e.*, concrete data types) must not be forced to add a virtual table pointer.

2. *Polymorphism* – All library objects must be accessed in a uniform, transparent manner. In particular, if new classes are included into the system, we won't want to change existing code.

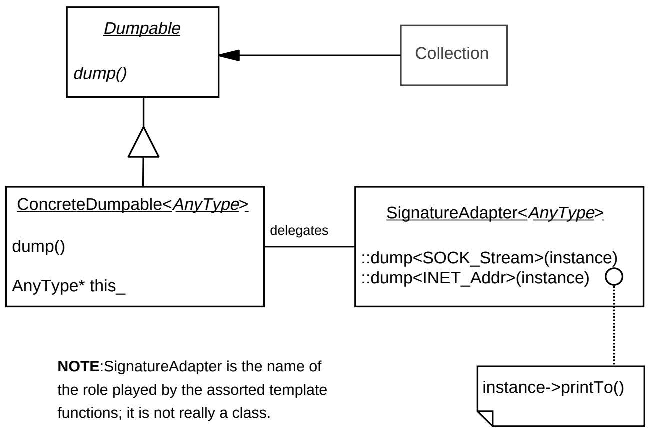
Consider the following example using classes from the ACE network programming framework [3]:

```
1. SOCK_Acceptor acceptor; // Global storage
2.
3. int main (void) {
4.    SOCK_Stream stream; // Automatic storage
5.    INET_Addr *addr =
6.      new INET_Addr // Dynamic storage.
7.    ...
```

The `SOCK_Stream`, `SOCK_Acceptor`, and `INET_Addr` classes are all concrete data types since they don't all inherit from a common ancestor and/or they don't contain virtual functions. If during a debugging session an application wanted to examine the state of all live ACE objects at line 7, 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}
```

An effective way to project the capability to dump state onto these class without modifying their binary layout is to use the *External Polymorphism pattern*. This pattern constructs a parallel, external inheritance hierarchy that projects polymorphic behavior onto a set of concrete class that need not be related by inheritance. The following OMT diagram illustrates how the External Polymorphism pattern can be used to create the external, parallel hierarchy of classes:



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.

As shown in the figure, we define an abstract class (`Dumpable`) having the desired `dump` interface. The parameterized class `ConcreteDumpable`<> inherits from `Dumpable` and contains a pointer to an instance of its parameter class, *e.g.*, `SOCK_Stream`. In addition, it defines a body for `dump` that delegates to the `dump`<> template function (shown in the figure as the `SignatureAdapter`<> pseudo-class and parameterized over the concrete class). The `dump` template function calls the corresponding implementation method on the concrete class, *e.g.*, `SOCK_Stream::dump` or `INET_Addr::printTo`.

Using the External Polymorphism pattern, it is now possible to collect `Dumpable` instances and iterate over them, calling the `dump` method uniformly on each instance. Note that the original ACE concrete data types need not change.

3 Applicability

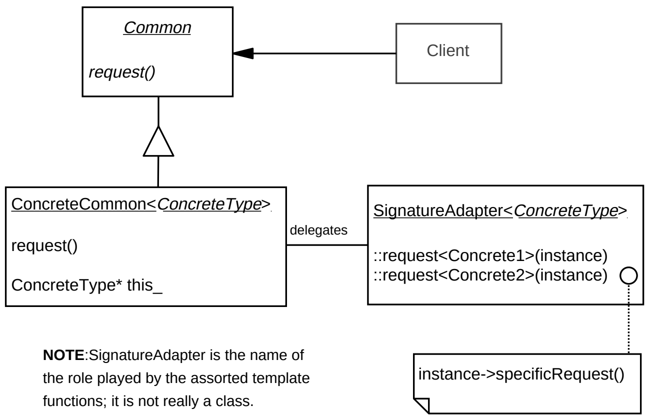
Use the External Polymorphism pattern when:

1. Your class libraries contain concrete data types that cannot inherit from a common base class containing virtual methods; and
2. The behavior of your class libraries or applications 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; or
2. Your programming language or programming environment allows methods to be added to classes dynamically.

4 Structure and Participants



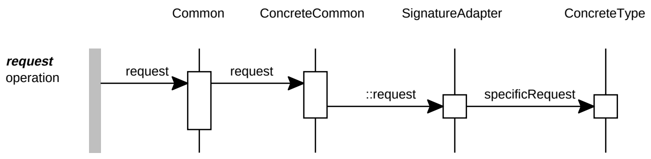
Common
request()
Client
ConcreteCommon<ConcreteType>
request()
ConcreteType* this_
delegates
SignatureAdapter<ConcreteType>
::request<Concrete1>(instance)
::request<Concrete2>(instance)
instance->specificRequest()
NOTE:SignatureAdapter is the name of the role played by the assorted template functions; it is not really a class.

- **Common** (`Dumpable`)
  - This abstract class forms the base of the external, parallel hierarchy and defines the interface(s) whose behaviors will be polymorphically projected and used by clients.
- **ConcreteCommon<ConcreteType>** (`ConcreteDumpable`)
  - This parameterized subclass of `Common` implements the interface(s) defined in `Common`. A typical implementation will simply forward the call to the appropriate `SignatureAdapter` template function.
- **SignatureAdapter::request<ConcreteType>** (::dump<>)
  - The template function adapter forwards requests to the object. In some cases, *e.g.*, where the signature of `specificRequest` is consistent, this feature may not be needed. However, if `specificRequest` has different signatures within several `Concrete` classes, the SignatureAdapter can be used to insulate `ConcreteCommon` from the differences.
- **ConcreteType**
  - The `ConcreteType` classes define `specificRequest` operations that perform the desired tasks. Although `Concrete` classes need not be related by inheritance, the External Polymorphism pattern allows you to treat all or some of their methods polymorphically.

5 Collaborations

The External Polymorphism pattern is typically used by having an external client make requests through the polymorphic

Common*. The following is an interaction diagram for this collaboration.



Many applications of the External Polymorphism pattern maintain a collection of objects over which the program iterates, treating all collected objects uniformly. Although this is not strictly part of the pattern, it is a common use-case that bears mentioning.

6 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 particular, the object layout of existing classes need not change 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 classes, it's easy to use conditional compilation to remove all trace of this pattern. This feature is particularly useful for systems that use the External Polymorphism pattern only for debugging purposes.

This pattern has the following drawbacks:

- *Instability* – The methods in the `Common` and `ConcreteCommon` must track changes to methods in the `Concrete` classes.
- *Inefficient* – Extra overhead due to multiple forwarding from virtual methods in the `ConcreteCommon` object to the corresponding methods in the `Concrete` object. However, judicious use of inline (*e.g.*, within `SignatureAdapter` and `Concrete`) can reduce this overhead to a single virtual method dispatch.

There is another consideration when using this pattern:

- *Possibility of inconsistency* – Externally Polymorphic methods are not accessible through pointers to the concrete classes, *e.g.*, using the example in Section 2 it's not possible to access `dump` through a pointer to `SOCK_Stream`. In addition, it is not possible to access other methods from the concrete class through a pointer to `ConcreteCommon`.

7 Implementation

The following issues arise when implementing this pattern.

- *Arguments to* `specificRequest`. Projecting polymorphic behavior will usually require adaptation of the signatures of the various `specificRequest`s. This can be complicated when, for instance, some require arguments, whereas others do not. The implementer must decide whether to expose those arguments in the polymorphic interface or to insulate the client from them.
- *Where does the code go?* As mentioned in Section 6, the external class hierarchy must be maintained in parallel with the original classes. The implementer must carefully select the source files in which co-dependent code, such as signature adaptation, is implemented.
- *External Polymorphism is not an Adapter.* The intent of the Adapter pattern is to *convert* an interface to something usable by a client. External Polymorphism, on the other hand, focuses on providing a new base for existing interfaces. A "serendipitous" use of External Polymorphism would find all signatures for `specificRequest` identical across all disparate classes, and thus not require the use of *SignatureAdapter*; it is in this situation where it is most apparent that External Polymorphism is not an Adapter.

8 Sample Code

To look at an example implementation, recall the motivating scenario: there are classes whose assistance we wish to enlist to create a flexible debugging environment. This implementation uses the External Polymorphism pattern to define a mechanism by which all participating objects (1) can be collected in a central "in-memory" object collection and (2) can dump their state upon request.

The `Dumpable` class forms the base of the hierarchy and defines the desired polymorphic interface which, in this case, is for dumping:

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

  // This pure virtual method must be
  // implemented by a subclass.
  virtual void dump (void) const = 0;
};
```

`ObjectCollection` is the *client*–a simple collection that holds handles to objects. The class is based on the STL `vector` class [4].

```
class ObjectCollection : public vector<Dumpable*>
{
public:
  // Iterates through the entire set of
  // registered objects and dumps their state.
  void dump_objects (void);
};
```

The `dump_objects` method can be implemented as follows:

```
void
ObjectCollection::dump_objects (void)
{
  struct DumpObject {
    bool operator()(const Dumpable*& dp) {
      dp->dump();
    }
  };
  for_each(begin(), end(), DumpObject());
}
```

Now that the foundation has been provided, we can define `ConcreteDumpable`:

```
template <class ConcreteType>
class ConcreteDumpable : public Dumpable
{
public:
  ConcreteDumpable (const ConcreteType* t);
  virtual void dump (void) const;
// Concrete dump method

private:
  const ConcreteType* realThis_;
// Pointer to actual object
};
```

The `ConcreteDumpable` methods are implemented as follows:

```
template <class ConcreteType>
ConcreteDumpable<ConcreteType>::ConcreteDumpable
  (const ConcreteType* t)
  : realThis_ (t)
{
}

template <class ConcreteType> void
ConcreteDumpable<ConcreteType>::dump (void) const
{
  dump<ConcreteType>(realThis_);
}
```

All that's left are the signature adapters. Suppose that `SOCK_Stream` and `SOCK_Acceptor` both have a `dump` method that outputs to `cerr`. `INET_Addr`, on the other hand, has a `printTo` method that takes the output stream as the sole argument. We could define two signature adapters. The first is a *generic* signature adapter that works with any concrete type that defines a `dump` method:

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

whereas the second is *specialized* signature adapter that is customized for `INET_Addr` (which does not support a `dump` method):

```
void
dump<INET_Addr>(const INET_Addr* t)
{
  t->printTo(cerr);
}
```

The `ObjectCollection` instance can be populated by instances of `ConcreteDumpable<>` by making calls such as:

```
...
ObjectCollection oc;
// Have instances of various SOCK_Stream, etc., types
...
oc.insert(oc.end(), aSockStream);
oc.insert(oc.end(), aSockAcceptor);
oc.insert(oc.end(), aInetAddr);
...
```

Then, later, we could query the state of those objects simply by calling

```
...
oc.dump_objects();
...
```

9 Known Uses

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

- The ACE framework uses the pattern to register ACE objects in 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 *DV-Centro* C++ Framework for Visual Programming Language development from DV Corporation uses the External Polymorphism pattern to create a hierarchy around unrelated internal system classes.
- The *Universal Streaming System* from ObjectSpace's Systems<Toolkit> uses the External Polymorphism pattern to implement object persistence via streaming.
- A variation of this pattern was independently discovered and is in use at Morgan Stanley, Inc. in internal financial services projects.
- 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 signature adapter came from usage in the OSE class library.[1] In OSE, template functions are used to define collating algorithms for ordered lists, etc.

10 Related Patterns

This pattern is similar to the Decorator and Adapter patterns from the Gang of Four (GoF) design patterns catalog [1].

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

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.

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

1. *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.

2. *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.

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

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

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

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, 2nd Edition*. Addison-Wesley, 1991.

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

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