Patterns for polymorphic operations

Three small object structural patterns for dealing with polymorphism

Alexander A. Horoshilov

hor@epsylontech.com

Abstract

Polymorphism is one of the main elements of the object paradigm. It is considered to be a requirement for any true object-oriented language. Like any powerful tool, it should be used properly. This article presents design patterns to impose additional constraints on polymorphic operations. The patterns can be applied to a wide range of object-oriented languages. They help to produce additional benefits from polymorphism and to avoid common pitfalls.

Introduction

Polymorphic methods are widely used when designing class hierarchies to provide flexible manners, to partially change object behavior and to avoid complex error prone *switches*/*cases*. With polymorphism, new derived classes can be added without the need to change the existing hierarchy.

Polymorphism provides a flexible means to redefine parts of object behavior but not to break existing code. To succeed with well-designed class hierarchies, in certain cases other constraints should be applied. When a polymorphic method is designed, its signature (name and argument types) is fixed in base class. So, it is important to provide the method with proper parameters. Patterns will help choose the right set of parameters.

Below, three patterns of polymorphism use are presented. They address to different cases. *Functional polymorphism* and *procedural*

polymorphism patterns describe two different scenarios. *Constant polymorphism* is a special case of functional polymorphism. *Functional* and *procedural* polymorphism got their names from languages like Pascal where two distinct syntaxes are used for subroutines. If a routine returns some value, it is called *function*. If it returns nothing, it is called *procedure*. As will be shown later, these kinds of routines are usually used specifically for their case of polymorphism. There are a lot of exceptions as well. These names may seem a little confusing for C/C++ programmers (where all routines are called functions), but they are more descriptive terms then other alternatives. These patterns have no relation to functional programming languages, or to functional programming.

For simplicity and clearness of interpretation, in the rest of the article the following terms are used:

- Operation whose implementation may be changed in derived classes called *polymorphic operation*. Available in many languages, the notion of *virtual* method is not used in the description in order to be more language-neutral.
- Parameters of the operation are divided into *input parameters* (supplied by the callee) and *output parameters* (returned from as result of its execution).
- Polymorphism is closely related to the inheritance. *You* are expected to provide the proper design for the base class. The developer of inherited classes is called the *programmer*.
- *Private*, *protected* and *public* scopes are used. They are available in many languages and have their usual meaning.

Pattern: Functional Polymorphism

Context

You are designing a class to be the root of a hierarchy. To implement its functionality additional information is needed. But only derived classes

can provide this information. So, you introduce a polymorphic operation. It is expected to provide the data required and called from base class methods.

Problem

How can *you* force the *programmer* to supply all information needed for base class?

Forces

- Base class can not provide default values, so if the programmer misses this information, run-time error or even unstable behavior will occur.
- This operation is not expected to change the state of the base object. It is allowed only to select from the set of the predefined alternatives.
- Two consequent calls of the operation return the same value. In some cases the result is always the same.

Solution

Make the operation return the information in *output parameters*. If only one value is needed, return it as result of the function execution. Several values can be grouped in complex data; for example, structure or array. Since some languages like Pascal call routine with return value *function*, we will name this case *functional polymorphism*.

Consequences

Modern compilers effectively warn the *programmer* if he misses the result value of the operation. These types of errors can be discovered at compile-time, which is always better than run-time testing.

Example

Traditional use of functionally polymorphic operations - is access to some information, specific for each class; for example, to class-IDs in environments with no RTTI support. The following code uses functional polymorphism to provide classes with a unique string identifier.

```
class Base
{
public:
  virtual const std::string get_name() = 0;
};
```

Derived classes should override such an operation and return the data (class name in this case) to correspond to the concrete class:

```
class Derived : public Base
{
public:
  virtual const std::string get_name()
  {
    return std::string("Derived");
  };
};
```

In C++ the pattern may be extended with the notion of *pure virtual* function. The *programmer* will be forced to redefine such a function in inherited classes. Because of that, he can not forget to implement the operation.

Known Uses

Good examples of these types of methods taken from Java are *java.lang.Object.hashCode* or *java.lang.Object.toString* [5].

Another example is *Object::is_a* operation from CORBA specification [6], which definition in many mapping languages uses functional polymorphism.

Pattern: Constant Polymorphism

Context

You have applied the functional polymorphism pattern. Because of some design issues, the result of the operation should preserve the same value during the object lifetime.

Problem

Functionally polymorphic operations can return different values each time they are called. Nothing prevents them from changing the return value. How can *you* guarantee time-invariantness of the value after initial object initialization?

Forces

- Value returned may vary between different instances of the same class. It depends on the concrete class instance.
- The operation may require some additional data accessible only in limited scope; for example, only during object initialization.
- Value requires complex calculations, so it causes significant overhead to call the operation many times.

Solution

Call the operation during object initialization and store value returned as object attribute. Put the attribute in *private* scope to prevent it from changing. Provide *public* non-polymorphic read accessor. The operation itself may have *protected* scope.

Consequences

This solution allows, on the one hand, derived class to implement their own version of polymorphic operation and, on the other hand, force the value returned to be the same during the object lifetime.

Another benefit is very effective access to the operation result. Many OO languages support the concept of field accessor to provide read-only access to field member. This implementation is equal to direct reading of the data field.

If value returned requires a lot of memory, application of this pattern increases the amount of dynamic memory needed for the object or the size of the object itself.

Special cases of constant polymorphism are operations called during object initialization when they should return another (usually especially created) object to be used later.

Example

The following code is the previous example modified to guarantee object that the class name will not changed.

```
class Base
{
public:
  const std::string name() const { return name_; };
  void init() { name_ = get_name(); };
protected:
  virtual const std::string get_name() = 0;
private:
  std::string name_;
};
```

In C++ such accessors are usually implemented as *inline* functions. Read accessor does not entail any overhead, it just imposes read-only constraint on the field.

Known Uses

In Visual Component Library shipped with Borland Delphi and Borland C++Builder development tools, *TCustomGrid* component has *CreateEditor* method. It is called during object initialization and expected to create and return the contained object. This object is accessible later by *InplaceEditor* property.

Pattern: Procedural Polymorphism

Context

You should allow the *programmer* to take control when a certain event occurs. When an event is triggered, a polymorphic operation is called. Derived class should override the operation to process this event. This type of event may be a public method that is called from any code, not only from methods of the base class.

Problem

Complex base object can have many events. Full implementation of all these handlers can require a lot of coding. If the event does not require obligatory processing, how *you* can allow the *programmer* not to implement some handlers?

Forces

- Base class can provide default behavior useful for a wide range of ancestors without overriding.
- If an operation is overridden, the inherited code of the base class may still be usable to be called from the derived one, may be more then one time.
- No return values are expected.

Solution

Design your operation not to return any values. Require the operation to perform all the jobs needed. If the operation requires additional information, pass it as operation parameters.

If possible, provide default implementation, suitable for derived classes.

Consequences

The Programmer is now has full control of the event handling. It is possible to reuse base class functionality, so time for adding a new class to the existing hierarchy is reduced. When overridden, an inherited operation can be called before, after, or in the middle of the new logic introduced. If it is appropriate, an inherited operation can be called several times, perhaps with different parameters. The *Programmer* can effectively reuse already existing implementations.

Example

This simple example shows how operation prints object state can be designed. By default, it prints only empty string. Ancestors add their own logic.

```
class Printable
{
public:
  virtual void print_state(std::ostream& os)
  {
    os << endl;
  };
};

class IntHolder : public Printable
{
```

```
public:
  virtual void print_state(std::ostream& os)
  {
    os << i;
    Printable::print_state(os);
  };
private:
  int i;
};
```

Known Uses

Most common examples of procedurally polymorphic operations are virtual destructors available in many languages.

Two Kinds of Polymorphism

It should be noted that two kinds of polymorphism – *procedural* and *functional* - are totally different. Some forces of these patterns are exclusive. So methods applicable to both patterns usually can not be created.

If, in the common case of providing the object with properties (or attributes - some named values, associated with this object), the read and write of the value of this property is performed by polymorphic methods. Then the read accessor is functionally polymorphic and the write accessor is procedurally polymorphic.

The bad design of operation signature and functionality expected may lead to an error-prone programming style. Operations designed with these two patterns, instead, improve purity and robustness of the design. Language specific features, as noted in pattern description, can increase benefits of the patterns.

Usually functionally polymorphic operations do not change the object state (and are declared as *const* in C++), while procedurally polymorphic operations do. Result of execution of functionally polymorphic operation may

be used to change object state (which is done in constant polymorphism pattern). Procedurally polymorphic operation takes full control of event handling.

Related Patterns and Additional Information

Patterns described are small-grained recommendations. They are widely used in other patterns which cover more complex scenarios. Because of this, concrete examples can be easily found in many cases.

Since [1] is generally accepted as design patterns foundation, it is good to highlight the use of described patterns in it. They will help the reader that is familiar with [1] to get additional ideas about pattern applicability.

- *Abstract Factory* uses functional polymorphism to creation new objects. It is illustrated in its code example by *MazeFactory* class with *MakeMaze*, *MakeWall*, *MakeRoom*, and *MakeDoor* operations.
- *Composite* uses procedural polymorphism to provide flexible hierarchy extension. In its code example, *Equipment* class has procedurally polymorphic operations *Add* and *Remove*. The same functional polymorphism is applied to *Power*, *NetPrice*, *DiscountPrice*, and *CreateIterator* operations of the same class.
- *Observer* (see *Observer::Update* method in the code example) and *Visitor* (see *Visitor::VisitElementA* and *Visitor::VisitElementB* methods in the code example) both have procedurally polymorphic handlers to perform a job.
- In most cases, all patterns described are used as *Template Method*.
- *Factory Method* is a clear case of functional polymorphism, as is shown in code example by *Creator::CreateProduct* method.

In [2] many recommendations about object-oriented design are given. They show how to design a high-quality, robust class hierarchy and describe various useful principles that can be combined with polymorphism patterns discussed here.

As described in [3], when *External Polymorphism* pattern is applied, all patterns introduced can be translated from polymorphism based on virtual methods to C++ templates and operator overloading, and be suitable for dealing with built-in C++ types and classes unrelated with inheritance.

An important part of the pattern is proper design of arguments passed to the operation and result values returned. In [4] additional recommendations about operation parameters design can be found.

References

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

[2] Robert C. Martin, "Engineering Notebook" Column, C++Report '96-97

[3] Chris Cleeland, Douglas C. Schmidt, and Tim Harrison, "External Polymorphism", in proceedings of PLOP'96.

[4] James Noble, "Arguments and Results", in proceedings of PLOP '97.

[5] Java™ 2 SDK, Standard Edition Documentation. http://java.sun.com/j2se/1.3/docs/api/index.html

[6] The Common Object Request Broker: Architecture and Specification. OMG document formal/01-02-33, http://www.omg.org/technology/documents/formal/corbaiiop.htm