C++ Idioms

by
James Coplien,

Bell Laboratories

Copyright ©1998 Lucent Technologies, Bell Labs Innovations. All rights reserved.

Introduction

This paper attempts to do three things. The first is to re-cast the well-known idioms of *Advanced C++ Programming Styles and Idioms* in pattern form. The second is to organize these idioms, which until now have survived as independent and largely unrelated patterns, into a true pattern language. That means that the patterns form a graph and that they should be applied in an order dictated by the structure of the graph. The third goal is to show that the patterns together (as a pattern language) attack what is metaphorically, conceptually, or actually a structural problem. Structure is a central element in Alexander's theories of aesthetics, a perspective that pervades all his work but which has become more directly articulated in recent works such as Nature of Order. These patterns do piecemeal construction of the structure of an inheritance hierarchy and the structure of the classes within it. The paper tries to explore that geometric nature.

Glossary

This space left blank intentionally as a placeholder for words that we later find we want to define here!

The Pattern Intents

These are the pattern intents, a quick index of the patterns in this paper.

- Handle/Body : Separating Interface from Implementation

Glossary

The Pattern Intents

- **Counted Body Idiom**: Manage logical sharing and proper resource deallocation of objects that use dynamically allocated resources
- **Detached Counted Body:** Adding reference counting to an object to which a reference count cannot directly be added
- **Handle/Body Hierarchy** : To separate representation inheritance from subtyping inheritance
- **Envelope/Letter** : To tie together common semantics of handle and body classes
- **Virtual Constructor:** How to build an object of known abstract type, but of unknown concrete type, without violating the encapsulation of the inheritance hierarchy
- **Concrete Data Type :** Determine whether to allocate an object on the heap or in the current scope
- **Algebraic Hierarchy**: Structuring classes whose type relationships follow classic algebraic types
- **Homogeneous Addition**: Simplify the implementation of operations in an Algebraic Hierarchy
- **Promote and Add:** How to add objects of two different types when only Homogeneous Addition is supported
- **Promotion Ladder**: How to assign type promotion responsibility to classes in an Algebraic Hierarchy
- **Non-Hierarchical Addition:** How to deal with arithmetic operations between types when neither can be promoted to the other
- **Type Promotion:** How to use two type conversion mechanisms—operator functions and constructors—to build a consistent type promotion structure

The Pattern Language Structure

The Pattern Language Structure

The patterns are presented as a pattern language with the structure of Figure 1. All the patterns are contained in Handle/Body and/or Concrete Data Type. Many of the GOF patterns, indicated as rounded boxes instead of rectangles, are also in this pattern language. The GOF patterns are not reiterated here.

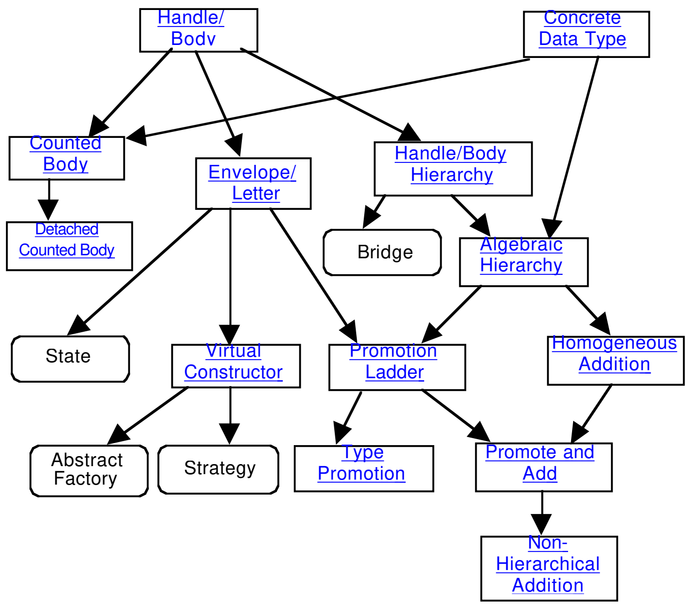


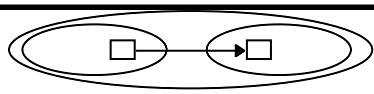
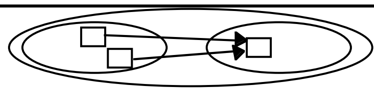
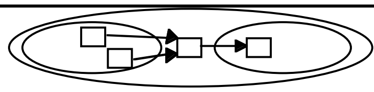
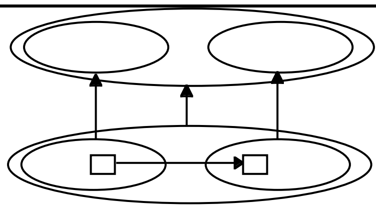
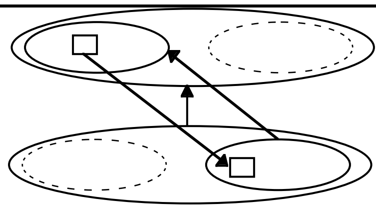
Figure 1 — The Pattern Language Structure

A Spatial Progression

A Spatial Progression

One can view these patterns as a way to guide the geometry of an inheritance hierarchy in C++. Geometry is an essential consideration in patterns, a fact most contemporary software patterns fail to heed. Inheritance structures are one of the most accessible structures of object-oriented software design, though they are less evident in code than implementation hierarchies or other direct geometric properties like indentation (and what it portends for scope and other semantic properties).

Table 1: Progression of Geometric Structure in the Patterns

Pattern Name	Geometry
Handle/Body Idiom	
Counted Handle/Body Idiom	
Detached Counted Handle/Body	
Handle/Body Hierarchies	
Envelope/Letter Idiom Virtual Constructor	

Most of the crisp idioms of *Advanced C++ Programming Styles and Idioms* deal with class structures and, in particular, inheritance structures. They are the foundations of flexible object-oriented programming in C++.

Here, we both develop a pattern language based on problems, solutions, and intents, and we develop the corresponding progression of geometric structures as in Table 1: Progression of Geometric Structure in the Patterns. Only the more basic patterns are depicted here. Each shows two ellipses, one ellipse representing the interface class, and another, the representation class. Objects (the rectangles) are members of the set defined by the class. The arrows show the relationships between objects or classes as appropriate.

Handle/Body Idiom

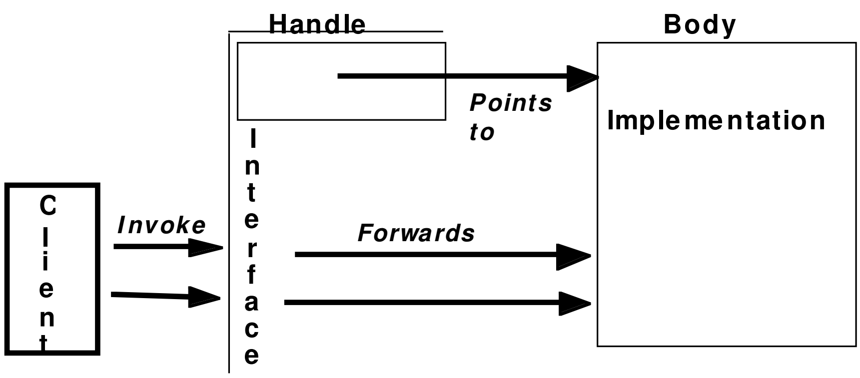
Context:
Advanced C++ programs using user-defined classes which should behave as much like built-in types as possible

Problem:
How do you separate interface from implementation in C++ objects?

Forces:

- C++ public and private sections were designed to separate implementation from interface, but changes even to private data force recompilation
- Changes to a class implementation cause unnecessary recompilation of client code.
- The class implementation is visible (though inaccessible) in a C++ class declaration.

Solution:
Split a design class into two implementation classes. One takes on the role of an identifier and presents the class interface to the user. We call this first class the handle. The other class embodies the implementation, and is called the body. The handle forwards member function invocations to the body.



Example:

```
class StringRep {
// this can be in a separate source file
// than class String, so it can be compiled
// separately, and made invisible to the
// client
friend class String;
    StringRep(const char *s);
    ~StringRep();
    int count; char *rep;
} *rep;

class String {
public:
    String();
    String(const String &s);
    String &operator=(const String &s);
    ~String();
    String(const char *s);
    . . . .
private:
    StringRep *rep;
};
```

Resulting Context:

Data changes can now safely be made to the implementation (body) without recompiling clients of the handle.

The implementation becomes "more hidden" behind a pointer.

This pattern doesn't address the issues of deep versus shallow copy and other run-time dynamics; see Counted Body, Envelope/Letter , and their subtending patterns.

The pattern also makes inheritance less useful; see Handle/Body Hierarchy to overcome this shortcoming.

This pattern has limits in managing the dynamically allocated memory of the body class; see Counted Body. It also introduces the need for occasional redundant update to the handle and body classes, a problem addressed in part by Envelope/Letter . To use this pattern in conjunction with inheritance, see Handle/Body Hierarchy .

Design Rationale:

All interesting problems in computer science reduce to what's in a name, and can be solved by one more level of indirection. In high-level languages like Smalltalk, identifiers and objects are different things. An object can be associated with one or more identifiers. The loose association between identifiers and objects clarifies questions of equality and identity, for example, and lays a foundation for automatic garbage collection. The Counted Body pattern takes advantage of this property as it apes Smalltalk garbage collection in a somewhat indirect way.

Counted Body

Counted Body Pattern

Alias: Counted Handle/Body

Context:

A design has been transformed using Handle/Body class pairs. The pattern may be relevant to other object-based programming languages.

Problem:

Naive implementations of assignment in C++ are often inefficient or incorrect.

Forces:

- Assignment in C++ is defined recursively as member-by-member assignment with copying as the termination of the recursion; it would be more efficient and more in the spirit of Smalltalk—that is, in the spirit of the benefits promised by close adherence to the object paradigm—if copying were rebinding.
- Copying of bodies is expensive.
- Copying can be avoided by using pointers and references, but these leave the problem of who is responsible for cleaning up the object, and leave a user-visible distinction between built-in types and user-defined types.
- Sharing bodies on assignment is usually semantically incorrect if the shared body is modified through one of the handles.

Solution:

Add a reference count to the body class to facilitate memory management; hence the name "Counted Body."

Memory management is added to the handle class, particularly to its implementation of initialization, assignment, copying, and destruction.

It is incumbent on any operation that modifies the state of the body to break the sharing of the body by making its own copy. It must decrement the reference count of the original body.

Resulting Context:

Gratuitous copying is avoided, leading to a more efficient implementation.

Sharing is broken when the body state is modified through any handle. Sharing is preserved in the more common case of parameter passing, etc.

Special pointer and reference types are avoided.

Smalltalk semantics are approximated; garbage collection is driven off of this model.

This pattern presumes that the programmer can edit the source code for the abstraction of interest. When that's not possible, use Detached Counted Body.

Design Rationale:

Reference counting is efficient and spreads the overhead across the execution of real-time programs. This implementation is a variation of shallow copy with the semantics of deep copy and the efficiency of Smalltalk name-value pairs.

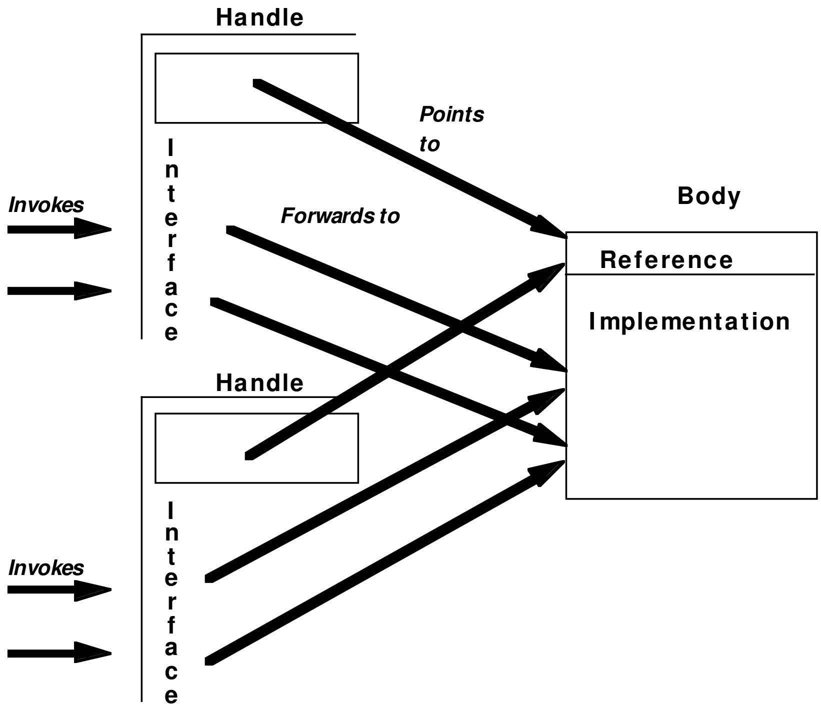
Example:

```
class StringRep {
friend class String;
    StringRep(const char *s): count(1) {
         strcpy(rep=new char[strlen(s)+1],
            s);
    }
    ~StringRep() { delete [] rep; }
    int count; char *rep;
} *rep;

class String {
public:
    String():rep(new StringRep("")) { }
    String(const String &s):
      rep(s.rep) { rep->count++; }
    String &operator=(const String &s){
      s.rep->count++;
      if(--rep->count <= 0) delete rep;
      rep = s.rep;
      return *this;
    }
    ~String() {
      if(--rep->count <= 0) delete rep;
    }
    void putChar(char c) {
      char *newrep=
        new char[strlen(rep->rep)] + 2;
      strcpy(newrep, rep->rep);
      rep->rep[strlen(rep)] = c;
      if (--rep->count <= 0) delete rep;
      rep = new StringRep(newrep);
    }
    String(const char *s):
      rep(new StringRep(s)) { }
    . . . .
private:
    class StringRep *rep;
```

```
};

int main() {
    String a = "hello", b = "world";
    a = b;
    return 0;
}
```



Detached counted body

Detached Counted Body

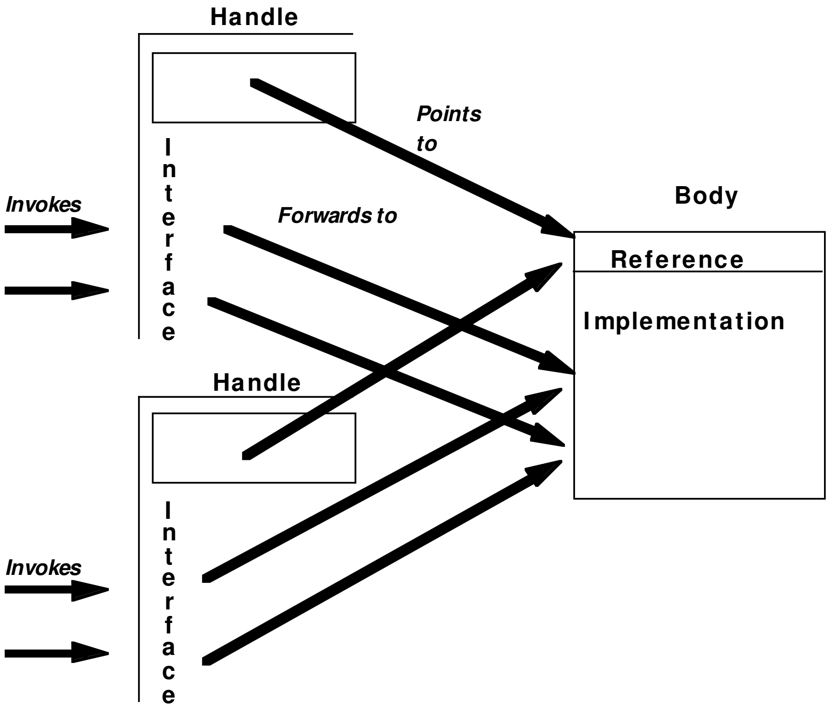
Context:
Many C++ programs use types whose implementations use dynamically allocated memory. Programmers often create such types and put them in libraries without adding the machinery to make these types as well-behaved as built-in types.

Problem:

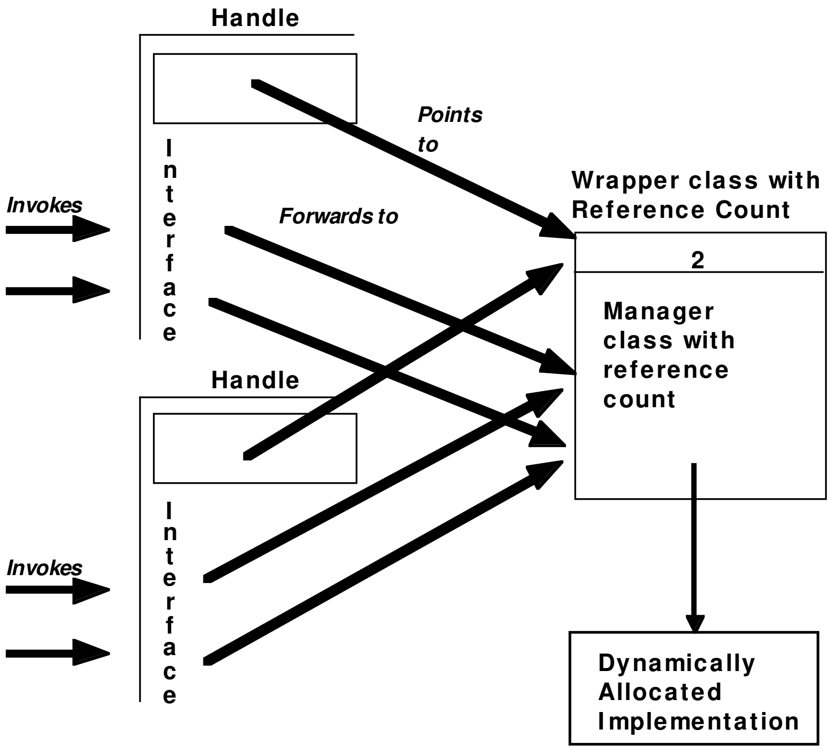
How do you overcome overhead of an additional level of indirection that comes when applying the Counted Body pattern to immutable classes?

Forces:

- The standard solution, Counted Body, embeds a reference count in a shared implementation that is managed by a handle class:



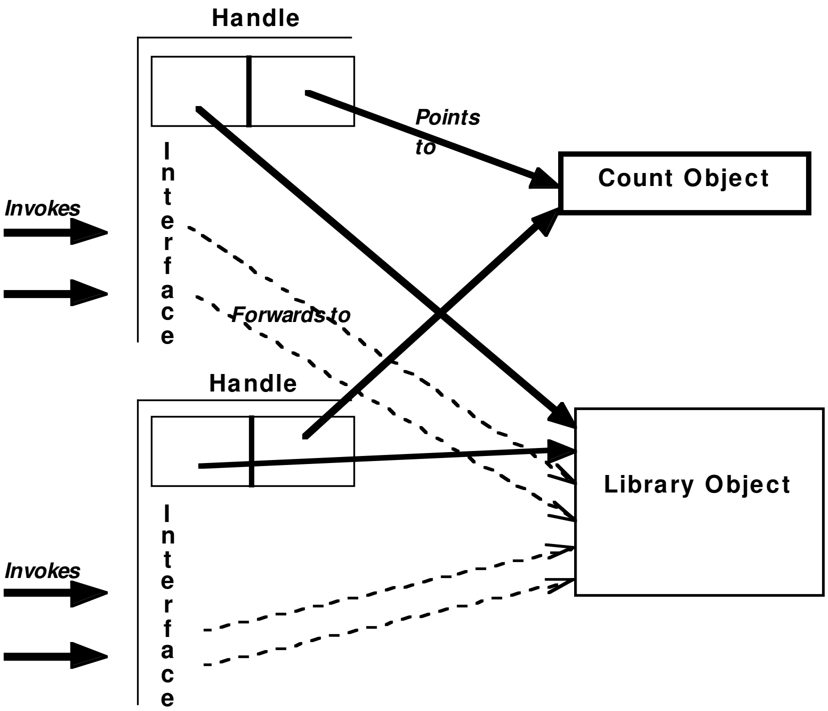
- However, we may not add a reference count to a library abstraction, since we only have object code and a header file. We could solve this with an added level of indirection,



but that adds a cycle to each reference, and may be too expensive.

Solution:

Associate both a shared count, and a separate shared body, with each instance of a common handle abstraction:



Example:

```
class String {
public:
    String():rep(new char[1]),
                count(new int) {
      rep[0] = '\0';
    }
    String(const String &s):
      rep(s.rep), count(s.count) {
        *count++;
    }
    String &operator=(const String &s){
      (*s.count)++;
      if(--*count <= 0) delete [] rep;
      rep = s.rep;
```

```
        return *this;
    }
    ~String() {
        if(--*count <= 0) delete [] rep;
        delete count;
    }
    String(const char *s): count(new int),
        rep(new char[strlen(s)+1]) {
            strcpy(rep,s);
    }
    . . . .
private:
    char *rep;
    int *count;
};

int main() {
    String a = "hello", b = "world";
    a = b;
    return 0;
}
```

Resulting Context:

Now we can access the body with a single level of indirection, while still using only a single indirection for the count.

Handles are slightly more expensive to copy than in Counted Body, memory fragmentation may increase, and initial construction overhead is higher because we are allocating multiple blocks.

Handle/Body Hierarchy (Bridge)

Context:

A C++ program in which the Handle/Body idiom has been applied, in which some classes have subtyping relationships, and implementation-sharing relationships, that do not correspond with each other.

One way this shows up is when a statically typed language that expresses subtyping as inheritance. The base class has an operation whose parameters correspond to degrees of freedom in its state space. The interface of the subtype is more constrained than the interface of the supertype. We want to inherit that operation in the derived class (which takes away at least one degree of freedom present in the base class; see the example). Stated another way, some operations that are closed under the base class are not closed under the derived class.

Another way this shows up is when the base class has a larger state space than the derived class. A derived class should restrict the state space of the base class.

Problem:

C++ inherits ties implementation inheritance and representation inheritance together, and we may want to inherit each separately.

Forces:

- You might want to inherit interface without inheriting implementation.
- In exceptional cases, you might want to inherit implementation without inheriting interface. For example, a base class member function may take one parameter for each of the degrees of freedom in its state space. Because the derived class is a subtype of the base class, it has fewer degrees of freedom than the base class. To inherit a base class operation whose parameters map onto degrees of freedom in the state space, the derived class must elide one argument (or otherwise constrain the arguments). But a base class operation inherited by the derived class should exhibit the same signature in both classes.
- If you inherit from a C++ class, you inherit its implementation.
- We usually use (public) inheritance (in the languages defined in the context) to express subtyping.

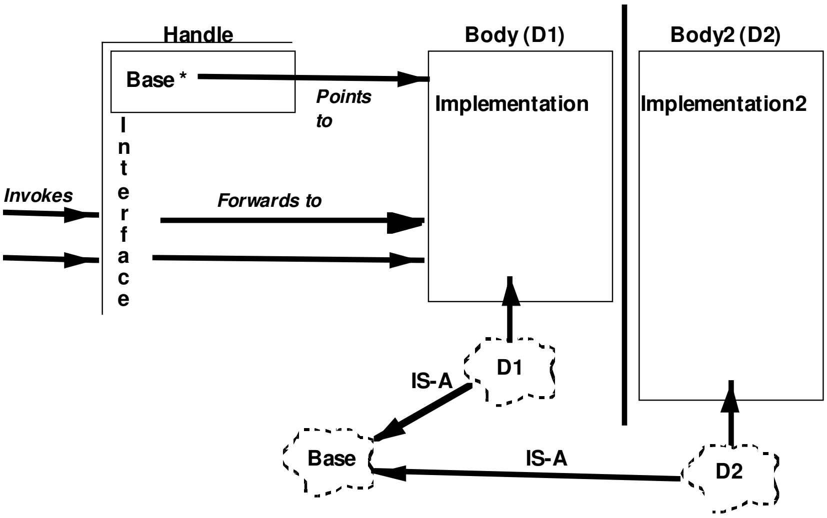
Example:

```
class Ellipse {
public:
    Ellipse(Pdouble majorAxis,
      Pdouble minorAxis, Point center);
    virtual void resize(Pdouble majorAxis,
                        Pdouble minorAxis);
 . . . .
private:
    // representation is in terms of two
    // foci, the constant sum of the distance
    // between any point on the ellipse and
    // each of the two centers
    Point center1, center2, sum;
};


class Circle: public Ellipse { // because a
                              // Circle IS-A
                              // Ellipse
public:
    Circle(Pdouble radius, Point center):
         Ellipse(radius, radius, center) { }
    void resize( ? );
private:
    // can reuse sum as the radius, and one
    // of the centers as the center, but the
    // other center is wasted
};
```

Solution:

Maintain separate inheritance hierarchies for handle classes and body classes. The base interface class contains a reference to the base implementation class.



Resulting Context:
Interface and implementation can be separately inherited. A compact representation can be used, independent of the interface presented to the application programmer. Though this saves the memory of the extra datum inherited from the base class, it adds space for a pointer.

Design Rationale:
Much of the power of object-oriented programming comes from separating identifiers and their semantics, from objects and their semantics. In C++, objects adopt the semantics of the identifiers to which they are bound. Identifiers adopt the semantics of their compile-time type.

These semantics include object size, permissible operations, and compatibility with other types.

Because objects adopt the semantics of the identifiers through which they are manipulated, they carry the semantics of a compile-time type.

This chain of dependency can be broken with pointers (which relax object size restrictions) and virtual functions (which retain the semantics of a fixed set of permissible operations, but allow for variant implementations of an operation). However, no single language mechanism combines the implementation de-coupling of pointers with the sharing of type semantics provided by inheritance.

In pure object-oriented languages, identifiers and objects have distinct semantics. There is a chain of dependency from type to object, but no compile-time dependency from identifier to type.

Though this seriously weakens compile-time type checking, it addresses the forces described above.

At first glance, resize(Pdouble,Pdouble) seems to apply to Ellipses but not to Circles. It is easy to conclude this, because resize is closed under Ellipse, but not under Circle. However, resize applies equally to Circles as to Ellipses, and is closed under Ellipses in general. This means that any attempt to resize a Circle changes it into an Ellipse. Such dynamic retyping is difficult to support in the given context of statically typed languages. To overcome this restriction, use idioms (like the Envelope/Letter idiom) or design patterns (like the GOF Bridge) that allow dynamic retyping of an object at run-time. Here, the type is a function of the number of degrees of freedom in the state. This is a subtle difference from the GOF State pattern alone, where type depends on value.

Envelope/Letter Idiom

Envelope/Letter Idiom

Context:

A program using Handle/Body pairs, and/or Counted Body pairs

Problem:

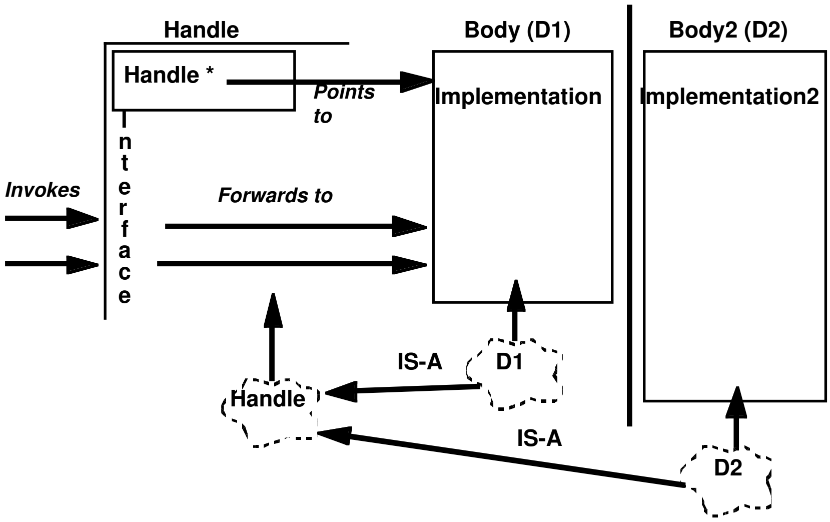
Supporting multiple implementations of a single ADT instance across its lifetime

Forces:

- Multiple implementations of an ADT share signature: we want to capture that in the design and code.
- All implementations (bodies) of a given ADT share signature with the ADT's interface (handle): we want to capture that
- Adding a new operation to a handle/body pair causes redundant update to both classes
- The handle implementation is coupled to the implementations of all possible bodies

Solution:

Derive all solution body classes from a common base class. To reflect the commonality in signature between the handle and the body, use the handle class as the common base class for alternative bodies. Make handle member functions virtual. Each alternative implementation derived from the handle class (in its role as the class defining the interface to the implementations) overrides suitable virtual functions. The base class implementation of these member functions defines the handle class functionality: it forwards requests to its associated body class instance.



The solution is weakly reflexive.

Resulting Context:

The ADT instance can now "metamorphize" between different body classes at run-time.

All bodies share a common signature, and share the signature of the handle.

New signatures need be added in one less place than if the information were duplicated.

The handle class can de-couple itself from the implementation of alternative body classes, if its public member functions are virtual.

Resulting Context:

This pattern is the basis for Virtual Constructors.

In Algebraic Hierarchy, this pattern forms the basis for a Promotion Ladder.

To vary the implementation at run time, consider the State pattern.

Design Rationale:

Others prefer to use a distinct base class for alternative implementations; this is O.K., depending on the forces one want to resolve.

Virtual Constructors

Virtual Constructor

Context

A particular client wants to create an object of an unspecified class chosen from a specified class hierarchy. Once created, any object created from the classes in the hierarchy can be used interchangeably by the client. The particular class is chosen from arbitrary global context according to the needs of the client. The Handle/Body pattern has been applied.

Problem

How do you create an object whose general type is known by the client requesting it, but whose specific subtype characteristics must be chosen from context?

Forces

- You want to hide the implementation of the inheritance hierarchy from the user of the hierarchy's objects: Only the base class interface should be published.
- Code must be written to select the most appropriate derived class. The code should be associated with the abstraction that minimizes coupling between all classes involved.
- The client must be able to use the services of any derived class object.

Solution

Just use the Envelope/Letter pattern structure. The base class is an intermediary agent that selects the appropriate derived type from context. The notion of supporting multiple implementations of an object across its lifetime generalizes to selecting the appropriate initial implementation, even in the degenerate case that the body instance doesn't change over the lifetime of the Handle/Body pair.

Resulting Context:

This pattern is the basis for Abstract Factory.

When the letter class variations are largely algorithmic, especially if the letter class contains only one member function, refine this pattern using Strategy.

Concrete Data Type

Context

Your design enumerates system classes, and you need to establish the lifetime and scope of the objects for those classes. This is particularly important for languages (like C++) where the object representation is visible to the user; that is, there are important distinctions between pointers and instances. Most of these languages lack garbage collection and other important finalization constructs.

Problem

How do you decide when to use operator new to allocate an object instead of allocating it within some program scope?

Forces

- Unlike the procedural paradigm, where lifetime follows scope, the object paradigm leads to designs where lifetime and scope are decoupled. Dynamic (heap) allocation is the usual mechanism to separate lifetime and scope.
- Per the context, the programming language can't make the difference between these two cases transparent.
- It's better for lifetime to follow scope, because the compiler can generate code to clean up the object when the scope closes. Dynamically allocated objects must be cleaned up by the programmer.
- Yet if every object is bound to an identifier declared in some scope (whether procedure or object scope), and its lifetime is bound to some identifier in that scope, it restricts the object lifetime to be different than the corresponding real-world abstraction. Software objects should capture important real-world abstractions, and must pay homage to such properties as lifetime and scope.
- On the other hand, not all classes reflect real-world objects; many are artifacts of the implementation.

Solution

Objects that represent "real-world" entities that live outside the program should be instantiated from the heap using operator new. The lifetime of these objects is likely to be independent of the lifetime of any procedure or class scope, so it is unwise to declare them as objects in any scope. Window is an example of such a class.

Objects that represent abstractions that live "inside" the program that are closely tied to the computational model, the implementation, or the programming language, Collection classes (string, list, set) are examples of this kind of abstraction (though they may use heap data, they themselves are not heap objects). They are concrete data types—they aren't "abstract," but are as concrete as int and double.

Resulting Context

Objects allocated from the heap can follow the lifetimes of the application abstractions they represent. Internal abstractions such as strings, which have no direct external world counterparts, can be automatically managed by the compiler.

To deal with dynamically allocated representations, see Counted Body.

For user-defined types that behave like built-in algebraic types, see Algebraic Hierarchy.

Algebraic Hierarchy

Algebraic Hierarchy

Context

Designing a system with user-defined types that support binary operations. Consider, for example, a number hierarchy with Complex as the base class. Complex is the most general of the number types under consideration. A Complex number has two scalar floating point numbers as its representation.

Problem

How do you construct the inheritance hierarchy for algebraic types?

Forces

- If we use C++ inheritance to express subtyping relationships, class Real would be derived from Complex, and Integer from Real. Each of these specific types supports all the operations of its base class.

- However, C++ inheritance also bestows all base class data on the derived class. That means that Real has at least two scalar floating point numbers as its representation. And likewise for Integer.
- We could minimize this waste by making the base class fields protected instead of private (so the derived class could "reuse" one of the base class fields). . Even so, the designer would prefer to use a single integer as the representation of Integer than to use a floating point number, in the interest of space and computational efficiency. And Real still has an extra floating point number that is waste.

Solution

Use the Bridge pattern [GOF95] to separate interface from implementation. The visible part of the Bridge is called class Number. It contains a pointer to a representation part, which contains the representation and operations of the specific type (Complex, Real, Integer, Imaginary, etc.).

```
class Number {
public:
    virtual Number add(const Number &n) {
      return rep->add(n);
    }
    virtual Number div(const Number&);
    . . . .
private:
    NumberRep *rep;
};


class Complex: public Number {
    . . . .
};


class Real: public Complex {
    . . . .
};


class NumberRep {
friend Number;
    virtual Number add(const Number&);
```

```
    virtual Number div(const Number&);
    . . . .
};

class ComplexRep: public NumberRep {
    virtual Number add(const Number &n) {
      . . . .
    }
    double rpart, ipart;
};

class RealRep: public NumberRep {
    virtual Number add(const Number &n) {
    . . . .
    }
    double rpart;
};
```

Resulting Context

Interface and implementation are now separate, and we can capture the subtyping semantics in the C++ inheritance hierarchy. Commonality between the implementation parts can be captured in a separate inheritance hierarchy if desired.

One can also combine the State pattern so given Numbers can change type over time:

```
class Number { . . . . };
class Complex: public Number { . . . . };
class Real: public Complex { . . . . };
class Rational: public Real { . . . . };


int main() {
     Complex i(3, -2), j(3, 2);
     i *= j;  // i.e., 13
     return 0;
}
```

In fact, this use of Bridge and State and other patterns forms a small pattern language in its own right. In particular, Homogeneous Addition, Promote and Add, Promotion Ladder, Non-Hierarchical Addition, Type Promotion fill out the structure of an Algebraic Hierarchy.

Homogeneous Addition

Context

You have built a hierarchy of classes (Algebraic Hierarchy) whose objects are to participate in binary operations. Each class is implemented using the Bridge pattern.

Problem

You need to distribute responsibilities to the objects; i.e., the addition operation. How many addition operations are there, and where do they belong?

Forces

- One of the main reasons for types in programming languages is efficiency. Good type design teaches the compiler how to generate efficient code for common operations.
- In general, efficient code must know the types of both operands involved.
- However, this leads to a combinatorial explosion in algorithms, n^2 for n types [Ingalls1986]. This violates the evolution law of continuity: it should be cheap to add a new type, not as expensive as adding code to every type that already exists!

Solution

Addition can always be expressed in terms of homogeneous operations (for example, Promote and Add). Each type should support only homogeneous algebraic operations, unless there are substantial performance gains to be realized by doing otherwise.

The solution generalizes to other binary operations.

```
class Complex: public Number {
public:
        . . . .
        // this must deal only with
        // Complex numbers
        Number add(const Number&) const;
};
```

```
class Imaginary: public Complex {
public:
    . . . .
    // this must deal only with
    // Imaginary numbers
    Number add(const Number&) const;
};
```

Promote and Add

Promote and Add

Context

You have an Algebraic Hierarchy). Each type knows how to add itself to an object of the same type (Homogeneous Addition). Each class knows important properties of its base class.

Problem

How do you do heterogeneous addition?

Forces

- Having Homogeneous Addition is fine, but even the most basic languages support polymorphic addition.
- The type of a result will, in general, be at least as general as the type of the more general of the two operands. For example, the result from adding a Complex and an Integer cannot be an Integer.
- Note that the result of multiplying two Complex numbers can be an integer. However, this knowledge is sophisticated enough that it belongs in Complex, not in Integer.

Solution

Using RTTI, it is straightforward to establish which of two object types is the more general Promote and Add the object of the more specific type to the type of the more general object, using Promotion Ladder. Then, use Homogeneous Addition to satisfy the request.

C++ in particular is rich in language features that support promotions involving both user-defined types and built-in types.

The pattern generalizes beyond addition to other binary operations.

```
class Number {
public:
    virtual Number promote() const;
};

Number operator+(const Number &n1,
     const Number &n2)
{
    if (n1.isA(n2)) {
      do { n1 = n1.promote(); }
        while (n1.isA(n2) &&
          n1.type != complexExemplar);
        return n2.add(n1);
    } else if (n2.isA(n1)) {
      do { n2 = n2.promote(); }
        while (n2.isA(n1) &&
          n2.type != complexExemplar);
        return n1.add(n2);
    }
}
```

Resulting Context:
This pattern fails if one of the operands is not a proper subtype of the other; to solve that, use Non-Hierarchical Addition.

Promotion Ladder

Promotion Ladder

Context
Types know how to promote themselves (Promote and Add), making heterogeneous addition possible.

Problem
Where do you put the knowledge of type promotion?

Forces
- You want to minimize cohesion and coupling between classes, even along an inheritance hierarchy. In particular, base classes shouldn't know about their derived classes.

- However, to do promotion from one type to another, each type must know something about the other.
- You might do this with casting and conversion operators, but in general the types can't be foreknown at compile time.
- Putting the knowledge of promotion in the base class might make it necessary to expose the derived class implementation if promotion is to be efficient. But putting knowledge of promotion in the derived class would likewise expose the implementation of the base class, which is even worse.
- Knowledge of promotion needn't be replicated in each pair of classes; once is sufficient. However, there must be some convention that points to the knowledge (e.g., in the more general or more derived class) to avoid ambiguity.

Solution

Each class should know how to promote itself to its own base class type. Promotions that involve more than two levels of the inheritance hierarchy can be handled by multiple successive promotions.

```
class Imaginary: public Complex {
public:
    . . . .
    Number promote(const Number& n) const {
        // always returns a Complex
        return Number(0, n.ipart())
    }
    Number add(const Number&) const;
};


class Complex: public Number {
public:
    . . . .
    // no promote:  nothing is promoted
    // to a Complex
    Number add(const Number&) const;
};
```

Non-Hierarchical Addition

Context

You've built a Promotion Ladder of types, each of which support Homogeneous Addition, that makes overall heterogeneous addition possible.

Problem

Sometimes, two objects are involved in a binary computation where neither can be promoted to the type of the other. Consider the addition of an Imaginary and an Integer. Neither knows how to promote itself to the type of the other.

Forces

- You could handle such exceptions as special cases, but that would clutter the code, and it would be difficult to present a convincing case that all cases were covered.
- You could build a full promotion matrix that would map any given pair of types onto the right promotion algorithm, but that would lead to a combinatorial explosion in conversion functions.

Solution

Detect cases where neither type can be promoted to the other. Promote both to a more general type and retry the operation.

```
Number operator+(const Number &n1,
          const Number &n2) {
    if (n1.isA(n2)) {
      do { n1 = n1.promote(); }
        while (n1.isA(n2) &&
          n1.type != complexExemplar);
        return n2.add(n1);
    } else if (n2.isA(n1)) {
      do { n2 = n2.promote(); }
        while (n2.isA(n1) &&
          n2.type != complexExemplar);
        return n1.add(n2);
    } else {
        // promote both to Complex & retry
```

```
                . . . .
        }
    }
```

Type Promotion

Type Promotion

Context:

The pattern applies to C++ and potentially to other object-oriented programming languages.

The decision of which promotion to apply is made at compile time.

The context is inadequate for the compiler to apply built-in translation rules, as would be possible for conversion between built-in types, or between a derived class and one of its base classes. Promotion Ladder may not apply.

Problem

Promotion between objects of different but related C++ types, zero or one of which is a built-in type.

Forces:

- The implementation of promotion from an object of one type to an object of another type is usually coupled to the implementation of both types.
- The C++ language lets the programmer associate such an implementation with only one of the participating types.
- The type containing the conversion implementation must be a class object type, since the implementation of built-in types cannot be redefined by the programmer.
- Two language mechanisms support user-defined conversions: constructors and conversion operators.
- Individually, each is an equally suitable solution in some circumstances, but use of both leads to an irreconcilable ambiguity.

Solution:

A program should promote a class object type to a built-in type using a member conversion operator:

```
class RationalNumber {
public:
    operator float() const;
    . . . .
};
```

A program should use constructors (as in Promotion Ladder) for all other promotions:

```
class Complex {
public:
    Complex(const RationalNumber&);
    Complex(double);   // no double::operator
                       // Complex, so do it
                       // here
    . . . .
};
```

Resulting Context

Coupling between types (and, in general, friend relationships) is still necessary between types; the force is resolved only to the extent that the conversion is firmly associated with a single type. The pattern does guarantee that the type bearing the conversion is always a class object type, however.

The pattern avoids most conversion ambiguities. An additional pattern must deal with the case:

```
struct S {
        operator int() const;
        operator float() const;
};


void f( int );
void f( float );
```

```
main() {
        S s;
        f( s ); // error: ambiguous call:
                  //  f ( struct S )
}
```

Design rationale:

A given type cannot know about every (more general) type in the universe whose instances might be created as generalizations of itself; the onus is on the type of the newly created object to understand its own initialization parameters.

Primitive types form an exceptional sub-pattern because their semantics are built into the compiler for efficiency, and their semantics are not as generally subject to change as for user-defined types.