Technical Report TR02-040

Department of Computer Science
Univ. of North Carolina at Chapel Hill

An Elemental Design Pattern Catalog

Jason McC. Smith

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175

smithja@cs.unc.edu

December 10, 2002

An Elemental Design Pattern Catalog

Jason McC. Smith

December 10, 2002

There are sixteen Elemental Design Patterns in this initial catalog, broken down into three main groupings: Object Elements, Type Relation, and Method Invocation. I expect that this collection will be added to and refined as time passes, as with any design patterns catalog.

Object Elements are those patterns that deal with the creation and definition of objects: CreateObject, AbstractInterface, and Retrieve. CreateObject describes when, how, and why we instantiate objects, what makes them special over procedural systems, and why they are not merely syntactic sugar. AbstractInterface offers a solution for needing to defer implementations to a later date, and Retrieve outlines how and why one uses objects as data fields inside an enclosing object.

Type Relation has one simple pattern: Inheritance, providing a discussion of the primary method by which typing information (*and* method body definitions!) are reused effectively in object-oriented systems.

The last grouping, Method Invocation, contains the final twelve patterns in this document. These were created from an inspection of the most common method call patterns in the Gang of Four text, which led to an expanded view of orthogonal invocation issues. Two axis of similarity and one of various typing relationships led to the twelve patterns described here. A more complete treatment of this discovery process and their derivation can be found in UNC-CS TR-02-011, May 2002.

These patterns, while small and precise, are important exactly because of their ubiquitousness in object-oriented programming, and their possible formalization. These patterns are those that every programmer uses on a daily basis, usually without conscious effort, and it is the very purpose of patterns to bring into conscious awareness that which is subconsciously and reflexively seen as useful. Other patterns have been produced that provide insights into programming concepts in general, but these are those that are unique to object-oriented systems, and, I believe, are a necessary if not sufficient set of design patterns for object-oriented programming from which all other design patterns can be built and composed. I follow the basic format used for the Gang of Four text, in an effort to create a common framework for discussion of these issues, and how they relate to the larger design patterns in use today.

CreateObject — Object Creation

Also Known As

Instantiation

Intent

To ensure that newly allocated data structures conform to a set of assertions and pre-conditions before they are operated on by the rest of the system, and that can only be operated on in pre-defined ways.

Motivation

An object is a single indivisible unit comprised of data and applicable methods that are conceptually related. We wish to ensure that this unit is in a particular coherent and well defined state before we attempt to operate on the object, and we wish to ensure that only well-defined operations can be performed on the object. In procedural languages, we can emulate an object rather well using records (C structs), and groupings of functions (libraries, perhaps with namespace encapsulation). We cannot, however, ensure either of our conditions of initial coherence or restricted operations.

We cannot, for example, guarantee that at the time of allocation of the record that the record's contents conform to *any* specific assertion we may choose to make. We can make certain that particular classes of static assertions will hold ("All records will have their third entry be set to '5'"),

```
typedef struct {
        int a;
        float b;
        short c = 5;
} myStruct;
```

but others are beyond our grasp ("Every third record created on a Tuesday will have the first entry set to '1', otherwise '0'.").

We can get around this problem in imperative languages by creating an initializer function that is to be called on all new allocated data records before use.

```
void initializeMyStruct( myStruct* ms ) {
        static int modulo3 = 0;
        if (modulo3 == 2) {
                modulo3 = 0;
                if (checkDay('tue')) {
```

```
                        ms->a = 1;
                }
        } else {
                modulo3++;
                ms->a = 0;
        }
}
```

This will be an effective solution, but not an enforceable one. Enforcement relies on policy, documentation, and engineer discipline, none of which have proven to be ultimately accurate or reliable. Therefore, we are back to the original problem of not being able to guarantee that any given assertion holds true on the newly allocated data. A malicious, careless or lazy programmer could allocate the structure, and then fail to call the proper initialization procedure, leading to possible catastrophic consequences.

Object-based systems (and class-based systems) provide an alternative. When an object is allocated by a runtime, it is initialized in a well-formed way that is dependent on the language and environment. All object-oriented environments provide some analogous mechanism as a fundamental part of their implementation. This mechanism is the hook at which the implementor can create a function (usually called the initializer or constructor) that performs the appropriate setup on the object. In this way *any* specific assertion can be imposed on the data before it is available for use by the rest of the system.

More formally, we can say that by using the **CreateObject** pattern, for some assertion A, and some object o, $A(o)$ is always true immediately following the creation of o. Without the **CreateObject** pattern, we can only state that either $A(o)$, or $D(o)$, where D is the default state of allocated storage in our system. Obviously the former is preferred for reducing errors.

There is no way for a user of the object to bypass this mechanism, it is enforced by the language and runtime environment. The hypothetical malicious, careless or lazy programmer is thwarted, and a possible error is avoided. Since this type of error is generally extremely difficult to track down and identify, avoidance is preferred.

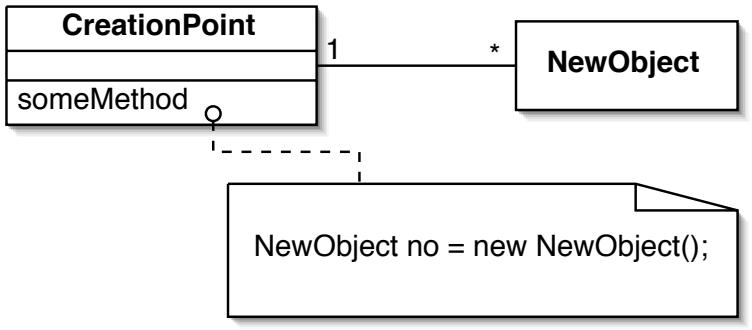
Similarly, we could emulate an object by the record and library approach of an imperative language and document that only certain operations from a pre-defined library may be used on that data structure, but there is no way to enforce such a policy. An object has a pre-defined set of methods that correspond to the functionality of that object, and only they may interact directly with the embedded data structure. In this way objects are more than just mere syntactic sugar or convenience, and provide a strong policy enforcement mechanism at the language level.

Applicability

Use **CreateObject** when:

- You wish to provide a data representation and enforce that only certain operations can be performed on any particular instance.
- You wish to provide allocated instances of a data representation and ensure that some set of pre-conditions are met before use.

Structure



Participants

CreationPoint
The object that requests the creation of a new object of type NewObject

NewObject
The object type to be created

Collaborations

An instance of CreationPoint requests that a new instance of object type NewObject be created. The exact mechanism for this will vary from language to language, but it generally consists of making the request 'of' the object type itself. (Naturally, the request is actually given to the runtime environment, with the object type as an argument, but the syntax of most languages creates the appearance that the request is made directly to the object type.) This is, as an astute reader will notice, a self-referential definition - you must have an object to be the CreationPoint before making a new object. So where does the initial object that kicks off this whole chain come into play? This is done by the run-time mechanism of the language being used, either implicitly (such as in dynamic languages such as SmallTalk) or explicitly (such as for hybrid class-based languages such as C++ that requires a top-level procedural `main`.)

Consequences

Most object-oriented languages do not allow for the creation of data structures using any other method, but do not require the definition of a developer-supplied initialization routine. A default one is generally supplied that performs a minimum of setup.

Some languages, while object-oriented, allow the creation of non-object data structures.

C++ and Objective-C are two examples, both derived from the imperative C language. Python, Perl 6 and other languages have similar historical reasons for allowing such behaviour.

Once an object is created, only the set of methods that were supplied by the developer of the original class are valid operations on that object. See the **Inheritance** pattern for an example of how to alter this state of affairs.

Objects have a time at the end of their existence when they are disposed of. This deallocation has an analogous function (the deallocator or destructor) that is called before the storage space of the data is finally released. This allows any post-conditions to be imposed on the data and resources of the object. While also a best practice to have a well-formed and definite deconstruction sequence for objects, it turns out that from a computational standpoint this is a matter of convenience only. If we had infinite resources at our disposal, objects could continue to exist, unused, for an indefinite amount of time. It is only because we have finite resources that destruction of objects is required essential in most systems. Construction of objects, however, injects them into the working environment so they can be used in computation, and is therefore a requirement, not a convenience. Conceptually, some object types may rely on the destruction of objects to enforce certain abstract notions (fixed elements of a set, etc), but this is a matter for the class designer.

Implementation

In C++:

```
class Bunny {
public:
        Bunny( int earLength );
};

int
main(int argc, char** argv) {

        Bunny b = new Bunny(1);

}
```

Abstract Interface — Object Structural

Intent

To provide a common interface for operating on an object type family, but delaying definition of the actual operations to a later time.

Also Known As

Virtual Method, Polymorphism, Defer Implementation

Motivation

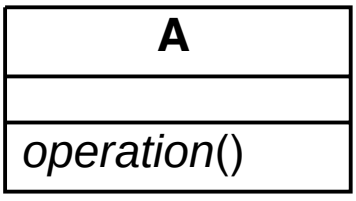
Often we will find that while we have a hierarchy of classes using the **Inheritance** pattern that conforms well to our conceptual design, occasionally we see that we run into a situation where we simply cannot provide a meaningful method implementation. We know *conceptually* what we want to do, we're just not entirely sure exactly how to go about doing it.

Applicability

Use Abstract Interface when:

- Implementation of a method can't be known until a later date
- The interface for that method *can* be determined
- You expect subclasses to handle the functionality of the method

Structure



Participants

A
The class type that declares an interface for *operation*

Collaborations

A defines an interface for a method, that some unknown later subclass will implement.

Consequences

The **CreateObject** pattern lets us instantiate objects, and the **Retrieve** pattern shows how to fill in the fields of that newly created object. **AbstractInterface** is unusual in that it indicates the *absence* of a method implementation; instead of showing us to fill in the method, it shows us how to defer the method definition until a later date. Note that this pattern does not actually provide the fill picture, including the later definition of the method.

The method is declared, to define the proper interface for our conceptual needs, yet the method body is left undefined. This does *not* mean that we simply define an empty method, one that does nothing... the method has *no definition at all*. This is a critical point, and one that is often missed by new object-oriented programmers. How this is done will vary from language to language. For instance, in C++ the method is set 'equal to zero' as shown in the example code.

Implementation

In C++:

```
class AbstractOperations {
public:
     virtual void operation() = 0;
};

class DefinedOperations : public AbstractOperations {
public:
     void operation();
};

void
DefinedOperations::operation() {
        // Perform the appropriate work
}
```

Inheritance — Type Relation

Intent

To use all of another classes' interface, and all or some of its implementation.

Also Known As

IsA

Motivation

Often times an existing class will provide an excellent start for producing a new class type. The interface may be almost exactly what you're looking for, the existing methods may provide *almost* what you need for your new class, or at the very least, the existing class is conceptually close to what you wish to accomplish.

In such cases, it would be highly useful and efficient to reuse the existing class instead of rewriting everything from scratch. One way of doing so is by the use of **Inheritance**. Every object-oriented language supports this approach of code reuse, and it is usually a core primitive of that language.

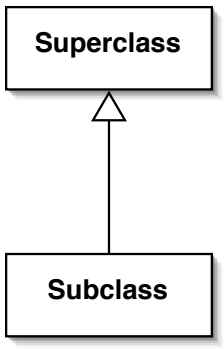
At its most basic, this pattern offers a relationship between a *superclass* or *base class*, and a *subclass* or *derived class*. The superclass (let's call it Superclass) is an existing class in the system, one that provides at a minimum an interface of concepts for methods, and/or data structures. A second class can be defined as being *derived* from Superclass, let's call it Subclass. The Subclass class *inherits* the interface and implementations of all methods and fields of Superclass, and this provides a starting point for a programmer to begin work on Subclass.

Applicability

Use Inheritance when:

- Two classes are related conceptually, providing the same services.
- One of those classes' implementation or interface is a superset of the other.
- *or* one class is mostly the same as the other.

Structure



Participants

Superclass
An existing class in a system that is used as a basis for producing a further class

Subclass
The secondary class that relies on the first for its basic interface and implementation

Collaborations

The Superclass class creates a basic set of method interfaces and possibly accompanying method implementations. Subclass inherits all the interface elements of Superclass, and, by default, all of the implementations as well, which it may choose to override with new implementations.

Consequences

Inheritance is a powerful mechanism, but has some interesting limitations and consequences. For one thing, a subclass may not *remove* a method or data field in most languages. (There a few exceptions, but these are rare, and beyond the scope of this pattern, as this gets to the heart of a subtlety of object-oriented theory.) A subclass is limited to overriding a method with a new implementation, or adding new methods or data fields.

It may seem that overriding is a waste of good code in the base class, and in many cases this is true. Look to the **ExtendMethod** pattern for a solution to this problem.

In some cases, one may not wish to inherit an entire existing class, but instead just small pieces of functionality may be desired. This may be due to a lack of confidence in the actual implementations of the method bodies (often when the original source code is unavailable), a reluctance to absorb a large class when only a small segment is needed, or other scenarios. In such situations, consider instead the **Delegate** pattern.

Implementation

The mechanism for creating an inheritance relationship will vary from language to language,

but is almost always readily apparent. Assume that we are modeling a rabbit farm, and wish to keep track of the health of the bunnies on hand. We make a class Bunny that contains the basic info for all rabbits, of any type. In addition, though, for lop-eared rabbits, we want to keep track of the ear length for breeding purposes. We can subclass a new class LopEaredBunny from Bunny, gaining all the necessary data and methods to model a rabbit, and we can then add additional information as we see fit:

In C++:

```
class Bunny {
public:
        Bunny( );
};

class LopEaredBunny : public Bunny {
public:
        LopEaredBunny( );
        float   getEarLength();
        void    setEarLength(float newEarLength);
private:
        float earLength;
};
```

Retrieve — Object Structural

Intent

To use an object from another non-local source in the local scope, thereby creating a relationship and tie between the local object and the remote one.

Motivation

Objects are a solid mechanism for encapsulating common data and methods, and enforcing policy, as shown in **CreateObject**. Singular objects, however, are of extremely limited power. In fact, if there were only one object in a system, and nothing external to it, it could be considered a procedural program - all data[1] and methods are local and fully exposed to one another. It is therefore critical that a well formed methodology be put into place for transporting objects across object boundaries. There are two situations where this is applicable and they differ only slightly.

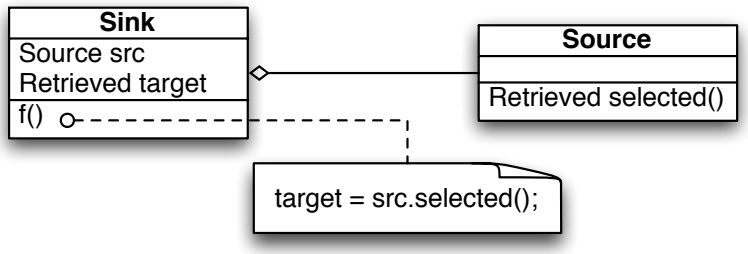
First, there is the simplest case where an external object has an exposed field that is being accessed, and secondly, where an external object has a method which is called, and the return value of that method is being used in the internal scope.

Applicability

Use Retrieve when:

- A remote object provides a value object that is required for local computation and is either:
- provided by a method call's return value or,
- provided by an exposed field object.

Structure



[1] Assuming that non-object data are allowed in said system.

Participants

Source
The object (or class) type that contains *selected*.

Sink
The object (or class) type that includes the item, *target*, to be given a new value.

Retrieved
The type of the value to be updated, and the value that is returned.

target
The field that is given a new value.

selected
The method or field that produces the new value.

Collaborations

This is a very simple relationship, consisting of two objects, and two methods. The distinguishing factors are the transferral of a return value into the local object space, and an update to a local field using that retrieved object.

Consequences

Most object-oriented languages do not allow the updating of methods with new values (method bodies), instead only data fields can be updated in this manner. From a theoretical point of view, there is no appreciable difference between these two scenarios, and we allow for both, instead relying on the language semantics themselves to govern which cases are valid and which are not. By deferring our definition until we have a set of well formed ρ-calculus facts, we avoid much of the complexity of handling the myriad of language quirks.

Tying two objects and/or types like this is an everyday occurrence, but it is one that should not be done without thought.

Implementation

In C++:

```
class Source {
public:
        Retrieved giveMeAValue();
};

class Sink {
        Retrieved target;
        Source srcobj;
public:
```

```
        void operation() { target = srcobj.giveMeAValue(); }
};
```

Delegate — Object Behavioural

Intent

To parcel out, or delegate, a portion of the current work to another method in another object.

Also Known As

Messaging, Method Invocation, Calls

Motivation

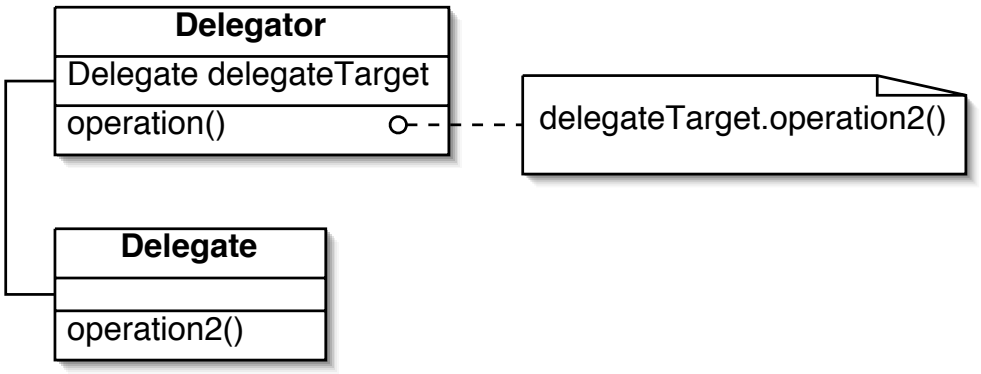
In the course of working with objects, it invariably comes about that 'some other object' can provide a piece of functionality we want to have. **Delegate** embodies the most general form of a method call from one object to another, allowing one object to send a message to another, to perform some bit of work. The receiving object may or may not send back data as a result.

Applicability

Use Delegate when:

- Another object can perform some work that your current method body wishes to have done.

Structure



Participants

Delegator
The object type sending the message to the Delegate

operation
The method within the Delegator that is currently being executed when the message is sent - the point of invocation of the *operation2* method call

Delegate
The object type receiving the message, with an appropriate method to be invoked

operation2
The method being invoked from the call site

Collaborations

A simple binary relationship, one method calls another, just as in procedural systems, and with the same sorts of caveats and requirements. Since we are working in an object-oriented realm, however, we have a couple of additional needs. First, that the object being called upon to help with the current task must be visible at the point of invocation. Second, that the method being invoked must be visible external to the enclosing object.

Consequences

All operations between any two objects can be described as an instance of the **Delegate** pattern, but it is much more useful to be able to describe further attributes of the relationship. See the further Method Invocation EDPs for refinements of **Delegate** that will be more useful.

Implementation

The most generalized and basic style of method invocation in object-oriented programming, **Delegate** is how two objects communicate with each other, as the sender and receiver of messages, performing work and returning values.

In C++:

```
class Delegator {
public:
        Delegatee       target;
        void operation() {target.operation2();};
}

class Delegatee {
public:
        void operation2();
}
```

Method Calling Classification

Other, Dissimilar

Redirect — Object Behavioural

Intent

To request that another object perform a tightly related subtask to the task at hand, perhaps performing the basic work.

Motivation

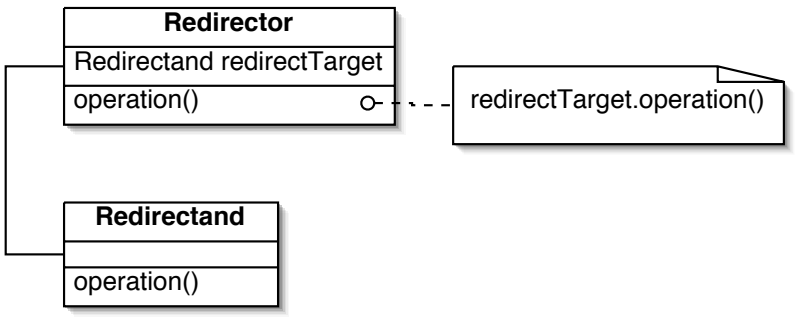
A small refinement to **Delegate**, **Redirect** takes into consideration that methods performing similar tasks are often named similarly. We can take advantage of this to provide more clarity as to the intent of this pattern over the more general form.

Applicability

Use Redirect when:

- A task to be performed by a method can be broken down into subtasks.
- One of those subtasks can be achieved by using another object.
- That target object has a method that has a similar intent, expressed through its signature name after Kent Beck's **Intention Revealing Selector** pattern.
- There is no distinct type relationship between the two objects.

Structure



Participants

Redirector
The originating site of the method call, contains a method named *operation* which has a subtask to be parceled out to another object. This object, redirectTarget, is a field element of type Redirectand. Redirector's *operation* calls redirectTarget.*operation* to perform a portion of its work.

Redirectand
The receiver of the message, which performs the subtask asked of it.

Collaborations

As with **Delegate** (and indeed all the Method Invocation EDPs), **Retrieve** is a binary relationship, between two objects and their enclosed methods.

Consequences

Almost identical to the **Delegate** pattern, the seemingly small equivalence of method names has some far-reaching effects, particularly when this pattern is combined with other EDPs and typing information to form complex interactions, as we will see in later patterns.

By leveraging the fact that both methods have the same name, and that this is a common way of declaring the intent of a method, we can deduce that this is an appropriate way of indicating that our originating call site method is requesting the invoked method to do some portion of work that is tightly related to the core functionality of the original method.

Implementation

In C++:

```
class Foo {
public:
        operation();
};

class Bar {
        Foo f;
public:
        operation() { f.operation(); };
};
```

Method Calling Classification

Other, Similar

Conglomeration — Object Behavioral

Intent

To bring together, or conglomerate, diverse operations and behaviours to complete a more complex task within a single object.

Also Known As

Decomposing Message

Motivation

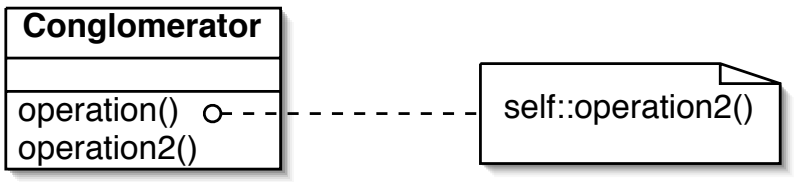
Often times an object is asked to perform a task which is too large or unwieldily to be performed within a single method. It usually makes conceptual sense to break the task into smaller parts to be handled individually as discrete methods, and then built back into a whole result by the method responsible for the larger task. (Kent Beck refers to this as the **Decomposing Message** pattern.) It may also happen that related subtasks can be unified into single methods, resulting in reuse of code inside a single object.

Applicability

Use Conglomeration when:

- A large task can be broken into smaller subtasks
- That task can, or must, be performed within a single object
- Several subtasks may be unified into a single method body

Structure



Participants

Conglomerator
Enclosing object type

operation

Master controlling method that is parcelling out subtasks

operation2
Subservient method performing a particular subtask

Collaborations

A specialization of **Delegate**, the object is calling a method of itself.

Consequences

As with **Delegate**, this pattern ties two methods into a reliance relationship, with *operation* relying on the behaviour and implementation of *operation2*. In this case, however, there may be immediate side effects to shared data within the confines of the object that they share.

Implementation

In C++:

```
class Conglomerate {
public:
void operation() { operation2(); };
void operation2();
}
```

Method Calling Classification

Self, Different

Recursion — Object Behaviour

Intent

To accomplish a larger task by performing many smaller similar tasks, using the same object state.

Motivation

Sometimes we run across a problem that, after analysis, can be broken down into smaller subtasks that are identical to the original task, except at a smaller scale. Sorting an array of items using the Merge Sort algorithm is one such example. Merge Sort takes an array and divides it into two halves, sorting each individually, then merging the two sorted arrays into a unified whole. Of course, each half sorted is done so by calling Merge Sort again, so they are again halved, and so on. Eventually arrays of a single item are reached, at which point the merging begins.

The process by which a method calls itself is known as *recursion*, and is ubiquitous in general programming. In object-oriented programming, the same principle applies, except that we have the added requirement that the object must be calling on itself, through an implicit or explicit use of *self*. (See **Redirect** for an example of calling another object, and **RedirectedRecursion** for calling another object of the same type.)

Recursion is, generally speaking, a way of folding a large amount of computation into a small conceptual space. Say we want to sort an array, and we decide that the easiest way to sort a large array is to split it into two roughly equal sized arrays, and then merge the two subarrays. The merging will be easy - if the head of array A is less than the head of array B, then the head of A is copied to the new, larger array, otherwise, copy the head of B. Remove the copied item from the appropriate array, and repeat until both arrays are merged.

Now, since this Merge Sort scheme relies on the subarrays being properly sorted, we correctly surmise that we can perform the sort algorithm on the two subarrays, splitting, sorting, then merging each in turn. Again, we are faced with the same sorting problem, so we continue in the same manner until we reach the smallest indivisible array, namely a single item. At that point, merging of sorted subarrays at each step can begin.

Assume we have an Array class that has the usual methods of add and delete. If we knew that the length of the beginning array was 4 items, then we could hardcode the entire sorting in pseudo-code like so:

```
Array
sort_merge(Array a) {
        a11 = a[0];
```

```
        a12 = a[1];
        a21 = a[2];
        a22 = a[3];
        // Sort first half
        if (a11 < a12) {
                a1.add(a11);
                a1.add(a12);
        } else {
                a1.add(a12);
                a1.add(a11);
        }
        // Sort second half
        if (a21 < a22) {
                a2.add(a21);
                a2.add(a22);
        } else {
                a2.add(a22);
                a2.add(a21);
        }
        // Merge
        if (a1[0] < a2[0]) {
                res.add(a1[0]);
                a1.delete(0);
        } else {
                res.add(a2[0]);
                a2.delete(0);
        }
        if (a1[0] < a2[0]) {
                res.add(a1[0]);
                a1.delete(0);
        } else {
                res.add(a2[0]).
                a2.delete(0);
        }
        if (a1.length == 0) {
                res.add(a2[0]);
                res.add(a2[1]);
        }
        if (a2.length == 0) {
                res.add(a1[0]);
                res.add(a1[1]);
        }
        if (a1.length == 1 && a2.length == 1) {
                res.add(a1[0]);
                res.add(a2[0]);
        }
}
```

While highly efficient (more on this later), this has an obvious drawback... it is limited to only working for arrays of length 4. A much more general version of this is one using looping. Here, we illustrate a for loop implementation (assuming for simplicity that the length of the array is a power of 2):

```
sort_array(Array a) {
        // Slice a into subarrays
        subarray[0][0] = a
        for (i = 1 to log_2(a.length())) {
                for (j = 0 to 2^i) {
                        prevarray = subarray[i-1][floor(j/2)]
                        subarray[i][j] = prevarray.slice(
                                (j mod 2) * (prevarray.length() / 2),
                                ((j mod 2) + 1) * (prevarray.length() / 2))
                }
        }
        // Sort subarrays and merge
        for (i = log_2(a.length()) to 0) {
                for (j = 1 to 0) {
                        subarray[i][j]
        }
        return subarray[0][0]
}
```

This succeeds in making our algorithm much more flexible, but at a cost of making it almost unreadable due to the overhead mechanisms. Note that we are looping inward to the base case (length of array is a single unit), storing state at every step, then looping outward using the state previously stored. This process of enter, store, use store, and unroll is precisely what a function call accomplishes. We can use this fact to vastly simplify our implementation using Recursion:

```
Array
sort_array(Array a, int beg, int end) {
        if (a.length() > 1) {
                return merge(sort_array(a, beg, end-beg/2),
                                sort_array(a, (end-beg/2) + 1, end));
        } else {
                return a
        }
}

Array
merge(Array a, Array b) {
        Array res;
        while (b.length() > 1 || a.length() > 1) {
                // If a or b is empty, then a[0] or b[0] returns a min value
                if (a[0] < b[0]) {
```

```
                        res.add(a[0]);
                        a.delete(0);
                } else {
                        res.add(b[0]);
                        b.delete(0);
                }
        }
        return res;
}
```

This is a much cleaner conceptual version that performs the same task as our looping variation. We are letting the runtime of the language handle the overhead for us.

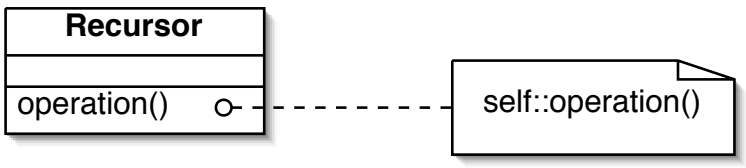
There are times, however, when that overhead handling may be determined to be excessive for our needs, such as in a high-performance embedded system. In such cases, optimizing away the recursion into loops, or further into linear code, is a possibility. These instances are extreme however, and tend to be highly specialized. In most cases the benefits of Recursion (conceptual cleanliness, simple code) greatly outweigh the small loss of speed.

Applicability

Use Recursion when:

- A task can be divided into highly similar subtasks.
- The subtasks must be, or are preferred to be, performed by the same object, usually due to a necessity of access to common stored state.
- A small loss of efficiency is overwhelmed by the gain in simplicity.

Structure



Participants

Recursor
The only participant, Recursor has a method that calls back on itself, within the same instantiation.

Collaborations

Recursor's method *operation* collaborates with itself, requesting smaller and smaller tasks

to be performed at each step, until a base case is reached, at which point results are gathered into a final result.

Consequences

The reliance of Recursion on a properly formed base case for termination of the recursion stack is the weakest point of this pattern. It is nearly impossible in most modern systems to wantonly consume all available resources... but recursion can do it easily by having a malformed base case that is never satisfied.

Implementation

In C++:

```
class SortedArray {
sort_merge() {

}
```

Method Calling Classification

Self, Same

RevertMethod — Object Behavioural

Intent

Bypass the current class' implementation of a method, and instead use the superclass' implementation, reverting to an 'earlier' method body.

Motivation

There are times when polymorphism works against us. One such instance would be when providing multiple versions of classes for simultaneous use within a system. Imagine a library of classes for an internet data transfer protocol. A base library is shipped as 1.0. With the 1.1 library, changes to the underlying protocol are made, and an application using the 1.1 protocol must be able to fall back to the 1.0 protocol when it detects that the application at the other end of the connection is only 1.0 enabled.

Now, one approach would be to use polymorphism directly, and have a base class that abstractly provides the protocol's methods, as in Figure 1, and create the proper class item on protocol detection. Unfortunately, this doesn't give us a lot of dynamic flexibility... what if the protocol needs to adapt on the fly, so that the two ends can handle graceful degradation of the connection, for instance. You could instantiate objects of each protocol type, and swap back and forth as needed, but there may be protocol state issues that would be troublesome.

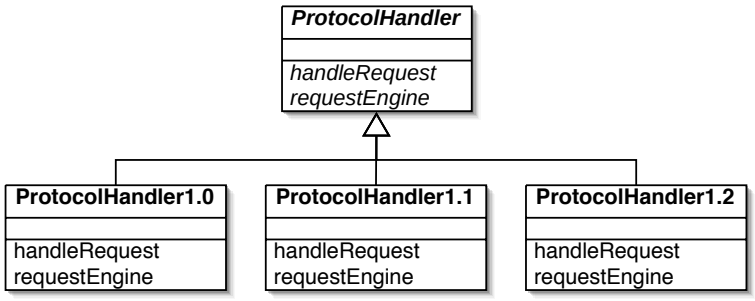


Figure 1: Polymorphic approach

So instead we may elect to have the base protocol class subclassed by a 1.1 version class, and just instantiate that instead. We now have only a single class to deal with, but we still need to be able to revert to the previous version. In Figure 2 we show an extended variant of this approach, with several versions.

In this case we can instantiate an object of just the last class in the chain, ProtocolHandler1.2, and conditional statements in the code will pass the protocol handling back up the chain to the appropriate version if needed. Note that we can make a simple test for whether or not the current object's version is appropriate for the protocol, and if not, pass

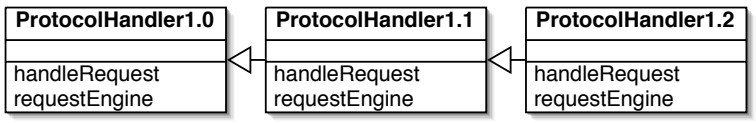


Figure 2: Subclassing approach

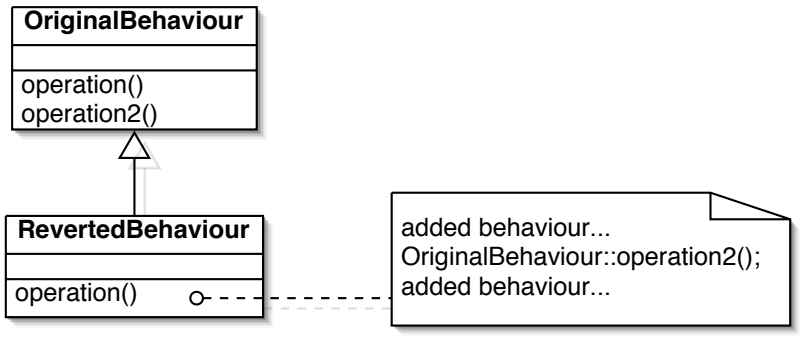
it simply to *super*, and let the test be re-enacted there, and so on. This vastly simplifies further maintenance of code that supports this protocol, since only the instance(s) of the ProtocolHandler need to be changed to the latest version anytime a new update to the library comes out, and all the graceful degradation is handled automatically. (Or, a specific version can be hardcoded for various reasons - it is up to the application developer.)

Applicability

Use RevertMethod when:

- A class wishes to a prior implementation of a method that it has overridden.

Structure



Participants

OriginalBehaviour
A base class, defining two methods, *operation* and *operation2*.

RevertedBehaviour
A subclass of OriginalBehaviour, with *operation* and *operation2* overridden. *operation* calls the OriginalBehaviour implementation of *operation2* (when one would normally expect it to call its own implementation of that method.)

Collaborations

In most cases when a subclass overrides a parent class' method, it is to replace the functionality, but the two method definitions can work together to allow an extension of the

behaviour. RevertedBehaviour relies on OriginalBehaviour for a core implementation.

Consequences

There is an odd disconnect conceptually between the overriding of a base class' method, and the utilization of that same method that can be confusing to some students. Overriding a method does not erase the old method, it merely hides it from public view for objects of the subclass. The object still has knowledge of its parent's methods, and can invoke them internally without exposing this to the external world.

Implementation

In C++:

```
class OriginalBehaviour {
public:
        void operation() { operation2(); };
protected:
        void operation2();
};

class RevertedBehaviour: public OriginalBehaviour {
public:
        void operation() {
                if (oldBehaviourNeeded) {
                        OriginalBehaviour::operation2();
                } else {
                        operation2();
                }
                                };
private:
        void operation2();
};
```

Method Calling Classification

Super, Dissimilar

Extend Method — Object Behavioral

Intent

Add to, not replace, behaviour in a method of a superclass while reusing existing code.

Also Known As

Extending Super

Motivation

There are many times when the behaviour of a method needs to be altered or extended, for reasons such as fixing a bug in the original method when the source code is unavailable, adding new functionality without changing the original method, etc.

One of the most common ways of doing this is, of course, to cut and paste the old code into the new method, but this presents a host of problems, including consistency of methods, and results in a potential maintenance morass. It is much better to adhere to the Single Point Principle and instead reuse the existing code, tweaking the results as needed.

It is possible, of course, to create a reference to a delegate object with the original behaviour, and then call into it when needed. This is the approach taken in **Redirect**.

In other cases, it is necessary or desired to subclass directly off of the original class. In such cases, we have two options for extending the original method. One is to cut and paste the old code into the new subclass' method, but this is not only undesirable from a maintenance standpoint, but also may not be possible, as in cases where the original source code is unavailable.

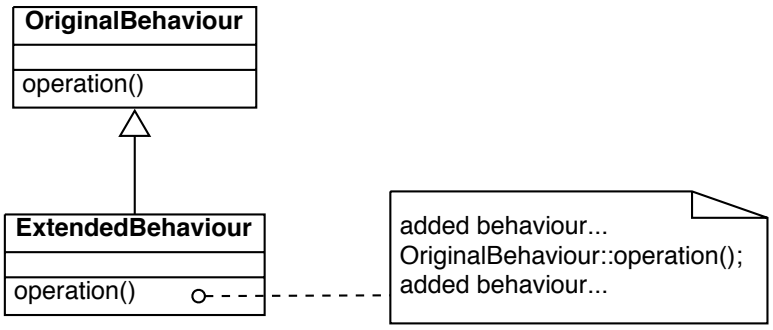
The second is **ExtendMethod**. In this case, we use **Inheritance** to provide the mechanism for reuse. We then override the original method, but make a call back to the superclass' implementation of the method. This provides us with simple maintenance, reuse, and encapsulation of the altered behaviour.

Applicability

Use Extend Method when:

- Existing behaviour of a method needs to be extended but not replaced.
- Reuse of code is preferred or necessitated by lack of source code.
- Polymorphic behaviour is required.

Structure



Participants

OriginalBehaviour
Defines interface, contains a method with desired core functionality.

ExtendedBehaviour
Uses interface of OriginalBehaviour, re-implements method as call to base class code with added code and/or behaviour.

Collaborations

In most cases when a subclass overrides a parent class' method, it is to replace the functionality, but the two method definitions can work together to allow an extension of the behaviour. ExtendedBehaviour relies on OriginalBehaviour for both interface and core implementation.

Consequences

There is an odd disconnect conceptually between the overriding of a base class' method, and the utilization of that same method that can be confusing to some students. Overriding a method does not erase the old method, it merely hides it from public view for objects of the subclass. The object still has knowledge of its parent's methods, and can invoke them internally without exposing this to the external world. Code reuse is optimized, but the method Operation in OriginalBehaviour becomes somewhat fragile - its behaviour is now relied upon by ExtendedBehaviour::Operation to be invariant over time. Behaviour is extended polymorphically and transparently to clients of OriginalBehaviour.

Implementation

In C++:

```
class OriginalBehaviour {
public:
     virtual void operation();
};

class ExtendedBehaviour : public OriginalBehaviour {
public:
     void operation();
};

void
OriginalBehaviour::operation() {
     // do core behaviour
}

void
ExtendedBehaviour::operation() {
     this->OriginalBehaviour::operation();
     // do extended behaviour
}
```

This pattern should translate very easily to most any object-oriented language that supports inheritance and invocation of a superclass' version of a method.

Method Calling Classification

Super, Same

DelegatedConglomeration — Object Behavioural

Intent

A **Conglomeration** pattern is appropriate, but we need to work with a distinct instance of our object type, resulting in a need for the **Delegate** pattern to be used.

Motivation

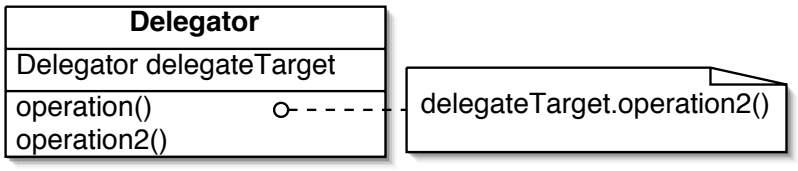
Many times we have objects of the same type working in concert to perform tasks. User interfaces are frequently coded as a collection of like objects collaborating to produce a larger functionality.

Applicability

Use DelegatedConglomeration when:

- A task can be broken into subtasks that are properly handled by the same object type
- Many objects of the same type work in concert to complete a task
- A single object is unable to complete the task alone

Structure



Participants

Delegator
The object type that contains references to other instances of its own type

delegateTarget
The enclosed instance that is called upon to perform a task

operation
The calling point within the first object

operation2
The subtask to be completed

Collaborations

Consequences

Implementation

In C++:

```
class Delegator {
        Delegator       delegateTarget;
public:
        void operation() { delegateTarget.operation2(); };
        void operation2();
}
```

Method Calling Classification

Same, Dissimilar

RedirectedRecursion — Object Behavioural

Intent

To perform a recursive method, but one that requires interacting with multiple objects of the same type.

Motivation

Frequently we will wish to perform an action that is recursive in nature, but it requires multiple objects working in concert to complete the task. Imagine a line of paratroopers getting ready for a jump. Space it tight, so instead of the commander indicating to each trooper to jump at the door, he stands at the back of the line, and when time has come, taps the last trooper on the shoulder. He knows to tap the shoulder of the trooper in front of him, and when that soldier has jumped, jump himself. This can continue on down a line of arbitrary length, from 2 to 200 troopers. All they have to do is when they feel a tap on their shoulder, tap the next person in line, wait, shuffle forward as space is available, and when they see the soldier in front of them go, jump next. The commander issues one order, instead of one to each soldier. A sample coding of this might look like:

```
Paratrooper::jump() {
        nextTrooper->jump();
        while (! nextTrooper->hasJumped() ) {
                shuffleForward();
        }
        leap();
}
```

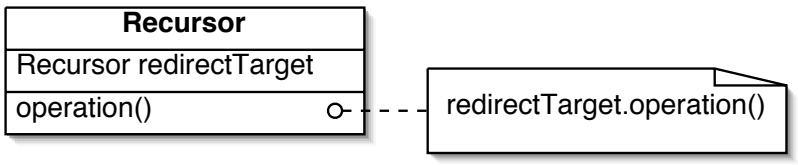
The current paratrooper cannot jump until the trooper in front has completed their task, and so on, and so on.

Applicability

Use RedirectedRecursion when:

- Recursion is a clean way of breaking up the task into subparts
- Multiple objects of the same type must interact to complete the task

Structure



Participants

Recursor
An object type that holds a reference to another instance of its own type

redirectTarget
The enclosed instance

operation
A method within Recursor that is recursive on itself... but through redirectTarget

Collaborations

Consequences

Implementation

In C++:

```
class Recursor {
        Recursor        redirectTarget;
public:
        void operation() { redirectTarget.operation(); };
}
```

Method Calling Classification

Same, Similar

DelegateInFamily — Object Behavioural

Intent

Related classes are often defined as such to perform tasks collectively. In such cases, multiple objects of related types can interact in generalized ways to delegate tasks to one another.

Motivation

User interfaces are a familiar type of system in which to find **DelegateInFamily** and related patterns (**DelegateInFamily**, **RedirectInFamily**, **RedirectInLimitedFamily**). This pattern allows one to parcel out tasks within a family of classes (often called a class cluster) when the interface, and method name, are known, but the precise object type (and therefore method body) may not be. It is a form of polymorphic delegation, where the calling object is one of the polymorphic types.

Consider a windowing system that includes slider bars and rotary dials as input controls, and text fields and bar graphs as display widgets. A input control is tied to a particular display widget, and sends it updates of values when the control is adjusted by the user. The input controls don't need to know precisely what kind of display widget is at the other end, they just need to know that they must call the *updateValue* method, with the appropriate value as a parameter. Since input controls also display a value implicitly, it is possible to programmatically change their adjustment accordingly, so they too need an *updateValue* method. By our **Inheritance** pattern, it looks as if the input controls and display widgets are of the same family, and in fact we want to make sure that they can all interact, so we create a class hierarchy accordingly:

```
class UIWidget {
        void updateValue( int newValue );
};

class InputControl : UIWidget{
        UIWidget*       target;

        void userHasSetNewValue()  {...  target->updateValue(myNewValue); ...}
};

class SliderBar : InputControl {
        void updateValue( int newValue );       // Moves the slider bar accordingly
        void acceptUserClick() {... userHasSetNewValue(); ...};
};

class RotaryKnob : InputControl {
        void updateValue( int newValue );       // Rotates the knob image accordingly
```

```
        void acceptUserClick() {... userHasSetNewValue(); ...};
};

class DisplayWidget : UIWidget {
        GraphicsContext gc;
};

class TextField : DisplayWidget {
        void updateValue( int newValue ) { gc.renderAsText( newValue ); };
};

class BarGraph : DisplayWidget {
        void updateValue ( int newValue ) { gc.drawBarLengthOf( newValue ); };
};
```

In the above example, the InputControl and UIWidget classes are the ones fulfilling roles in a **DelegateInFamily** pattern.

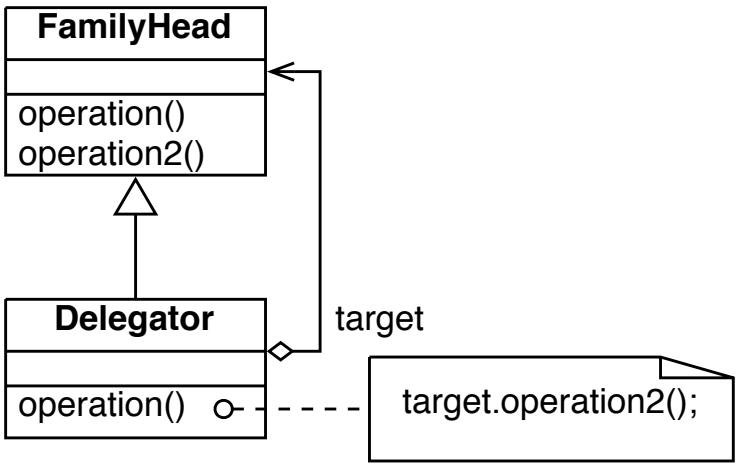
The SliderBar and RotaryKnob objects don't have to know anything about where their value is going, and in fact, they could be tied to each other, with each adjusting the other in sync. (You could even have two objects of the same InputControl subclass tied together, such as two SliderBar instances.) We have separated the concerns of who is sending what data, and who is receiving it. All that is of concern is that the data is being sent to a properly receiving client polymorphically, and that the current calling object is *of that polymorphic family.* This allows for a single unified interface for many classes that can work in tandem to perform many tasks.

Applicability

Use DelegateInFamily when:

- Delegation is appropriate, with not necessarily related subtasks to be performed
- Polymorphism is required to properly handle the message request
- The calling object is of a type in the polymorphic hierarchy

Structure



Participants

FamilyHead
The base class for a polymorphic class cluster

Delegator
A subclass of FamilyHead

target
A polymorphic instance of FamilyHead that is contained by Delegator

operation
The calling site

operation2
The called subtask

Collaborations

Consequences

Implementation

The *target* can be defined in either the base class, or the subclass, it just needs to be accessible from with the subclass.

In C++:

```
class FamilyHead {
        void operation();
        void operation2();
};

class Delegator : public FamilyHead {
```

```
        FamilyHead      target;
        void operation() { target->operation2(); };
};
```

Method Calling Classification

Parent, Dissimilar

RedirectInFamily — Object Behavioral

Intent

Redirect some portion of a method's implementation to a possible cluster of classes, of which the current class is a member.

Motivation

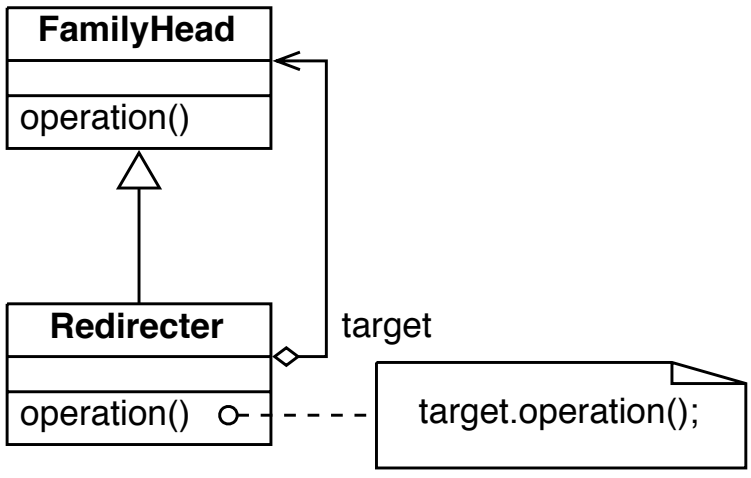
Frequently a hierarchical object structure of related objects will be built at runtime, and behaviour needs to be distributed among levels.

Applicability

Use RedirectInFamily when:

- An aggregate structure of related objects is expected to be composed at compile or runtime.
- Behaviour should be decomposed to the various member objects.
- The structure of the aggregate objects is not known ahead of time.
- Polymorphic behaviour is expected, but not enforced.

Structure



Participants

FamilyHead
Defines interface, contains a method to be possibly overridden.

Redirecter
Uses interface of FamilyHead, redirects internal behaviour back to an instance of FamilyHead to gain polymorphic behaviour over an amorphous object structure.

Collaborations

Redirecter relies on the class FamilyHead for an interface, and an instance of same for an object recursive implementation.

Consequences

Redirecter is reliant on FamilyHead for portions of its functionality.

Implementation

In C++:

```
class FamilyHead {
public:
        virtual void operation();
};

class Redirecter : public FamilyHead {
public:
        void operation();
        FamilyHead*     target;
};

void
Redirecter::operation() {
        // preconditional behaviour
        target->operation();
        // postconditional behaviour
}
```

Method Calling Classification

Parent, Similar

DelegateInLimitedFamily — Object Behavioural

Intent

When **DelegateInFamily** is too generalized, and it is necessary to pre-select a sub-tree of the class hierarchy for polymorphism.

Motivation

Static typing is a way of pre-selecting types from a well defined pool, and forming more concrete notions of an object's type at runtime. Polymorphism is a technique for abstracting out typing information until runtime. Sometimes we need a balance of the two. This pattern and the related **RedirectInLimitedFamily** both weigh these opposing forces but for slightly different outcomes.

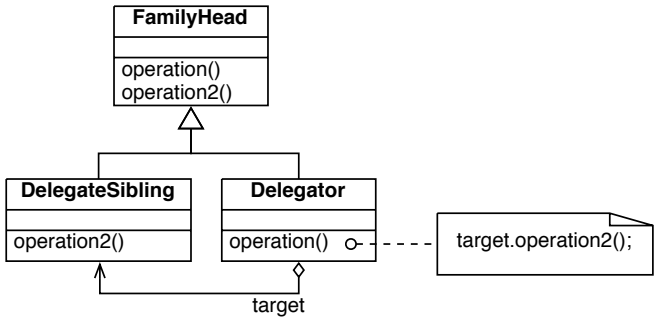
This pattern is concerned with *Delegation*, the more generalized form of computational subtasking.

Applicability

Use DelegateInLimitedFamily when:

- **DelegateInFamily** is the appropriate general pattern...
- ... but greater control over the possible types of objects is required

Structure



Participants

FamilyHead
The base class for a polymorphic class cluster

Delegator

A subclass of FamilyHead

DelegateSibling
Another subclass of FamilyHead

target
A polymorphic instance of DelegateSibling that is contained by Delegator

operation
The calling site

operation2
The called subtask

Collaborations

Consequences

Implementation

In C++:

```
class FamilyHead {
        void operation();
        void operation2();
};

class DelegateSibling : public FamilyHead {
        void operation2();
};

class Delegator : public FamilyHead {
        DelegateSibling*        target;
        void operation() { target->operation2(); };
};
```

Method Calling Classification

Sibling, Dissimilar

RedirectInLimitedFamily — Object Behavioural

Intent

When **RedirectInFamily** is too generalized, and it is necessary to pre-select a sub-tree of the class hierarchy for polymorphism.

Motivation

Static typing is a way of pre-selecting types from a well defined pool, and forming more concrete notions of an object's type at runtime. Polymorphism is a technique for abstracting out typing information until runtime. Sometimes we need a balance of the two. This pattern and the related **DelegateInLimitedFamily** both weigh these opposing forces but for slightly different outcomes.

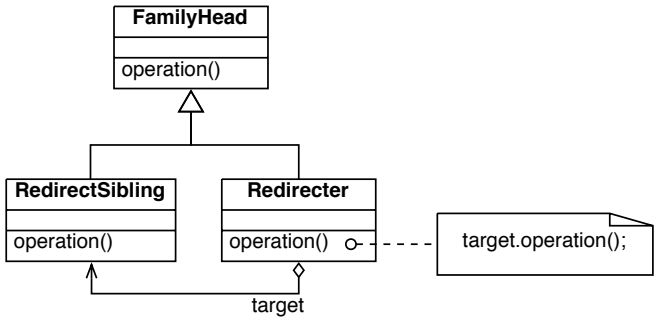
This pattern is, obviously, concerned with *redirection*, having some advance knowledge of the intimacies of the task at hand to be able to determine that similarly named methods will be calling each other in a chain of subtasking.

Applicability

Use RedirectInLimitedFamily when:

- **RedirectInFamily** is the appropriate general pattern...
- ... but greater control over the possible types of objects is required

Structure



Participants

FamilyHead
Defines interface, contains a method to be possibly overridden, is the base class for both Redirecter and RedirectSibling

Redirecter
Uses interface of FamilyHead, redirects internal behaviour back to an instance of RedirectSibling to gain polymorphic behaviour over an amorphous *but limited in scope* object structure.

RedirectSibling
The head of a new class tree for polymorphic behaviour

Collaborations

Consequences

Implementation

In C++:

```
class FamilyHead {
public:
        virtual void operation();
};

class RedirecterSibling : public FamilyHead {
        void operation();
}

class Redirecter : public FamilyHead {
public:
        void operation();
        RedirecterSibling*      target;
};

void
Redirecter::operation() {
        // preconditional behaviour
        target->operation();
        // postconditional behaviour
}
```

Method Calling Classification

Sibling, Similar