INHERITED BEHAVIOUR PATTERN

Class Behavioral

Intent

Extends the behaviour of a method in subclasses by reusing the inherited method and adding to it. This pattern is supported by some object-oriented languages but is lacking in C++.

Also Known As

Callback Method (?), Distributed Method (?)

Motivation

First, note that this is not a design pattern as much as a programming pattern (if there is such a thing). It shows how to address a lack in an object-oriented programming language with regards to the object-oriented paradigm, in this case C++'s lack of behaviour inheritance in methods.

The description of this pattern was deemed necessary in order to properly explain its use in a project I am working in. Many of my colleages had difficulties with my verbal explanations, so I decided to put it in a pattern format so as to explain it more clearly.

So, you often need to extend a class in order to process some new condition. You redefine the methods in order to properly process this new conditions, but when you are faced with a typical case, you would like to fall back onto the previous definition.

Suppose, for instance, that a given class `Parent` provides access to its attributes through a generic `Get()` method*. This `Get()` method would need two parameters: an identifier for the requested attribute and a buffer to put the requested value. We can define a subclass of `Parent`, named `Child`, which adds a number of new attributes to those inherited from `Parent`. Now let's say we invoke the `Get()` method on an instance of `Child`. Clearly, if the requested attributes is specific to the `Child` class, the method should return its value in the buffer. However, if the requested attribute was inherited from `Parent`, we would like to call the `Get()` method of `Parent` in order to fetch it. In addition, we could ask that the `Get()` method of `Parent` handle any erronous call. This way, the `Get()` method in `Child` needs only to check if the requested attribute is specific to `Child`, and if not to pass the request up to `Parent`.

A similar case occurs when you want to extend the behaviour the an existing method. You want to keep the previous behaviour intact and simply add to it. In this case, the first thing the method in `Child` should do is to call the equivalent method in `Parent`. Once that part of the behaviour has been guaranteed, it can then proceed with more specific tasks.

This mechanism can also be applied between `Parent` and its superclass(es), and so on up the inheritance tree. This way, a method's behaviour can be spread across the nodes on the tree. This is why this pattern is named Inherited Behaviour.

In some object-oriented languages, like Simula, this pattern is already part of the language. When you refine a method in a subclass, the parent's behaviour is already

* This is the way in which attributes of managed objects are accessed in OSI network management.

present and your extensions are run after the inherited method has been called (by the execution environment). The order of execution can be modified with some keyword, such as `INNER`, in the parent's method. But this mechanism is missing in other languages such as C++, Eiffel, and Smalltalk, where the subclass' method completely replaces the parent's. This pattern is used to remedy this situation and get the expected object-oriented behaviour from inherited methods when using the C++ language. Note that this mechanism is also lacking in other object-oriented languages like Smalltalk and Eiffel, but the following description only applies to C++.

This pattern can also be useful for someone implementing a compiler for some object-oriented programming language.

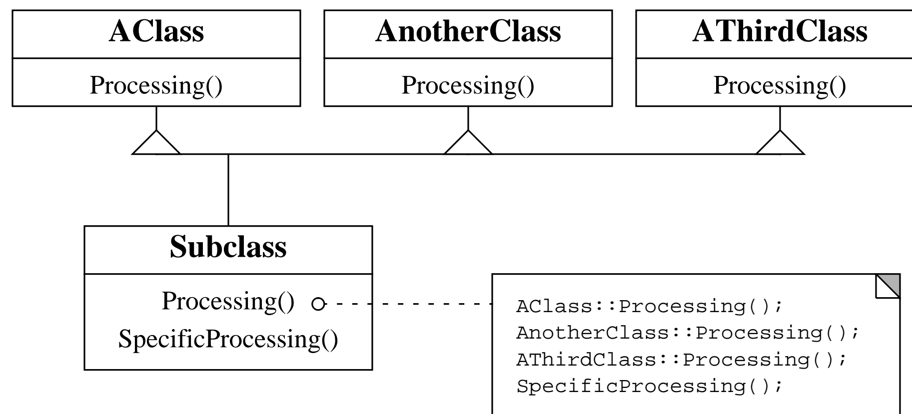
Applicability

The Inherited Behaviour pattern works best when the behaviour of a method is spead across the class's superclass(es), or when a method encapsulates the behaviour of previously defined equivalents in the superclass(es). The Inherited Behaviour pattern works best when:

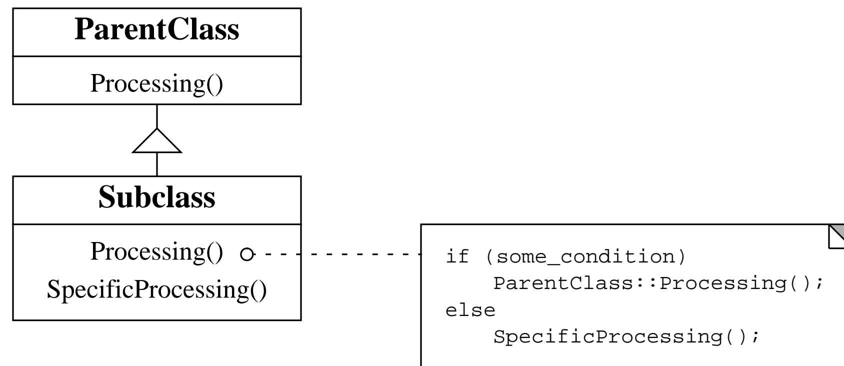
- A subclass only adds some attributes to its superclass(es) without changing the inherited interface.
- A subclass must chose between a generic behaviour or a more specific one based on some local condition.
- A subclass needs to extend the behaviour of a method of a superclass without changing its expected basic behaviour.

Structure

First form, where we simply extend the (composed) behaviour from the parent class(es):



Second form, where we make a choice between the behaviour of the parent class or some other behaviour:



Participants

- **AClass, AnotherClass, AThirdClass, or ParentClass**
  - Define the method.
  - Implement the method's generic behaviour.
- **Subclass**
  - Calls the superclass(es)' generic method at the appropriate time.
  - Implements the method's specific behaviour.

Collaborations

- The superclasses only implement that part of the behaviour that is generic. They also handle most of the error situations.
- Subclass only implements that part of the behaviour that is specific to it. It relies on the superclasses' implementations in all other cases.

Consequences

The Inherited Behaviour pattern has the following benefits and liabilities:

1. *It concentrates the generic behaviour of methods in the superclasses.* There is no need for the methods defined in a subclass to reimplement the behaviour that is expected of a superclass. This duplication of code would be difficult to maintain and the "cut & paste", if not done carefully, could introduce errors in the copy that were not in the original.
2. *The subclass only deals with that part of the behaviour that is specific to it.* This yields smaller, less complex methods that are easier to comprehend and debug. It also keeps the subclass closer to its specification, especially when this specification only deals with the specifics of the subclass and not its generic, inherited nature.
3. *It is easy to modify the behaviour of a class hierarchy by modifying its root.* Since everyone in the hierarchy depend upon its direct superclasses, modifying the behaviour in one class also modifies it in all its descendants, which is what you would expect from inheritance after all.☺

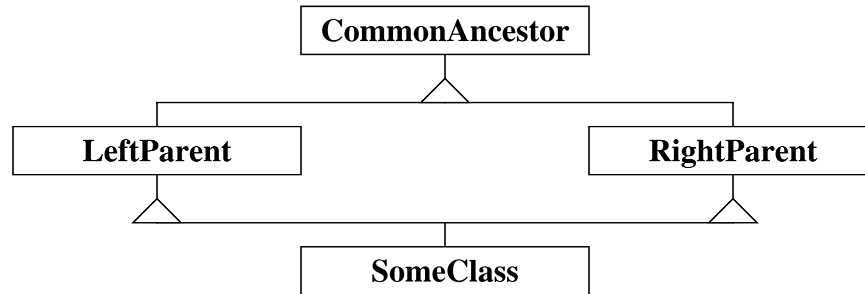
Implementation

Here are some useful technique for implementing the Inherited Behaviour pattern.

1. *Put the handling of all possible error conditions in an abstract class's method.* As much of the error conditions as possible should not be handled by the subclasses. Putting such handling in an abstract superclass that is then inherited by all others

implies that specific subclasses don't have to deal with error cases, unless such cases are specific to them.

2. *Call the parents' method first.* In most cases where you are extending the behaviour of a previously defined method, call the original methods first so as to make sure the object is in a valid state when you start you own processing. This also prevent those methods from undoing some of your work.
3. *Beware of mutiple paths to a common ancestor.* Under multiple inheritance, if two or more parents are descendants from some common ancestor, it may be inappropriate to call its method more than one. You must then use some mechanism to control the method calls through the inheritance tree. Suppose you have the following hierarchy:



A method in SomeClass would inherit the behaviour from the equivalent method in CommonAncestor twice, once through LeftParent and once again through RightParent. One way to ensure that each method is called only once is to use a flag in each class. As the call is passed from one ancestor to the next, the flag is set. This way, if a call finds the flag set, it returns immediately without doing anything. Once the processing is over, the original class can call another method to clear the flags.

This approach has several drawbacks. First, additional space is required for the flags. Second, how can a class along a path know wether or not it initiated the call and wether or not it should clear the flags before it terminates? The latter can be solved by making the object have a public interface, defined in CommonAncestor, that calls a private method to handle the processing and then clears the flags. Each subclass then simply has to redefine the private method.

Sample Code

Let's apply the Inherited Behaviour pattern to the attribute fetching method shown earlier. Suppose we have a generic class `Object` that is a superclass of all classes in a framework. In this system, attributes are identified by numeric constants such as the following:

```
typedef unsigned long ul;

const ul LENGTH_ID            = 10L;
const ul WIDTH_ID             = 11L;
```

`Object` could then look like the following:

```
class Object {
public:
      enum StatusCode {
            success,
            attribute_unknown,
            attribute_unreadable};

      // Constructors, desctructor, and other methods
      virtual ErrorCode GetAttribute (ul, void *);
}
```

Notice that `GetAttribute()` is declared virtual. This is to ensure that when invoking the method through a `Object*`, the actual method of the object is called instead of the expected `Object::GetAttribute()`. This method would look like:

```
Object::StatusCode Object::GetAttribute (ul id, void *buffer)
{
      // Object is an "abstract" class without any attribute

      return (attribute_unknown);
}
```

Now, if we look at the class `Rectangle`, which is a subclass of `Object`, we can see the following:

```
class Rectangle : public Object {
private:
      int   width, length;

public:
      // Constructors, desctructor, and other methods
      StatusCode GetAttribute (ul, void *);
};
```

Note that even though the `GetAttribute()` method is not implicitly declared virtual, it is so nonetheless since is was defined as such in the superclass. This second method could be defined as the following:

```
Object::StatusCode GetAttribute (ul id, void *buffer)
{
      ErrorCode   result;

      switch (id)
      {
            case WIDTH_ID:
                  memcpy (buffer, &width, sizeof (width));
                  result = success;
                  break;
            case LENGTH_ID:
                  memcpy (buffer, &length, sizeof (length));
                  result = success;
                  break;
            default:
                  result = Object::GetAttribute (id, buffer);
                  break;
      }

      return (result);
}
```

Notice how the default case of the switch statement passes control to the previous definition of `GetAttribute()`. The method in a subclass of `Rectangle` would have a similar structure, except that its default action would then be to call `Rectangle::GetAttribute()` instead of `Object::GetAttribute()`.

Known Uses

The C++ Framework for OSI Management interfaces uses the Inherited Behaviour pattern for many of the operations on managed objects. The inheritance tree root, `CMO`, handles errors while subclasses handle specific operations and attributes.

Microsoft Foundation Class Library uses the Inherited Behaviour pattern to ensure that user-defined subclasses behave according to what is expected from their superclasses. For example, to tie an action to the pushing of a button in the GUI, you define your own subclass of `CButton` where you perform the action. But the first thing to do is to call `CButton::OnPush()` so that the proper animation takes place on the screen. Suppose we have a method M of some class C, which is a subclass of some class B. Its definition would then looks like the following:

```
class C : public B {
public:
      void M();
      // other members
};

void C::M ()
{
      B::M();
      // Specific processing
}
```

Related Patterns

Composite: Inherited Behaviour can be used across Leaf and Composite instances in order to handle some specific situation.

Template Method:

Inherited Behaviour can be used to implement the PrimitiveOperation in a TemplateMethod. This is especially true when such a method's behaviour is spread across the inheritance tree instead of neetly contained in only one ConcreteClass. Template Method can also be used to solve problems with calls in multiple inheritance, as was seen earlier.

Contacts

http://www.iro/umontreal.ca/~keller/Layla

Jean Tessier, AT&T Laboratories, Advanced Technologies Division, Holmdel, NJ.

Jean.Tessier@att.com

Rudolf K. Keller, Département d'informatique et de recherche opérationnelle, Université de Montréal, C.P. 6128, succursale Centre-ville, Montréal (Québec) H3C 3J7, Canada.

keller@iro.umontreal.ca