The Object Recursion Pattern

Bobby Woolf
SilverMark, Inc.
woolf@acm.org

OBJECT RECURSION — Object Behavioral

Intent

Distribute processing of a request over a structure by delegating polymorphically. Object Recursion transparently enables a request to be repeatedly broken into smaller parts that are easier to handle.

Also Known As

Recursive Delegation

Motivation

Consider the need to determine if two objects are equivalent. Simple objects and primitives are easy to compare; just use native operations. The difficulty lies in comparing arbitrarily complex objects.

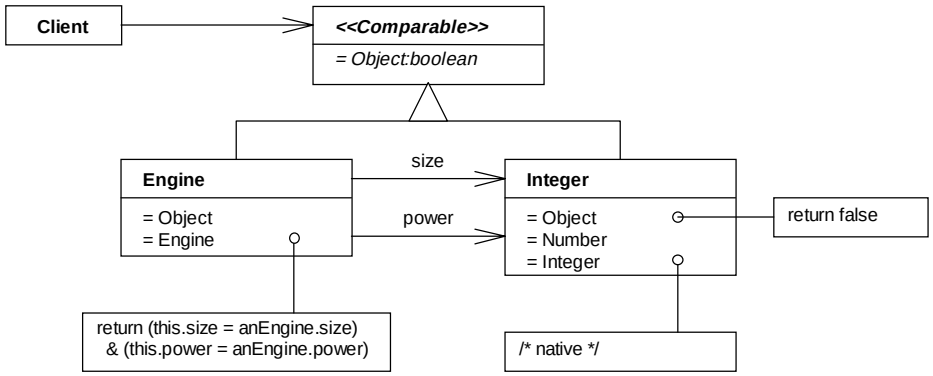
One approach is to employ a Comparer object that accepts any two arbitrarily complex objects and answers whether the subjects are equivalent. The Comparer takes the subjects, breaks each one into pieces, and compares the pieces to determine if they're equivalent. If any of the pieces are too complex to compare, the Comparer repeats the process by breaking it into pieces, and so on until all of the pieces are simple enough to compare.

This Comparer approach has several undesirable consequences. The Comparer must recognize what kind of object the subjects are and know how to decompose those kinds of objects into simple parts that can be compared. The more complex a subject is, the more complex the code for comparing it needs to be. Every time a developer implements a new class whose instances might need to be compared, he also needs to add code to Comparer (or a subclass) to handle the new class. The decomposition code in Comparer depends heavily on the class' implementation, so whenever that implementation changes, the code in Comparer must be changed accordingly. The subjects need to provide extra protocol for decomposing, protocol that exposes the objects' implementation. All in all, the Comparer requires lots of complex code that is difficult to develop and maintain.

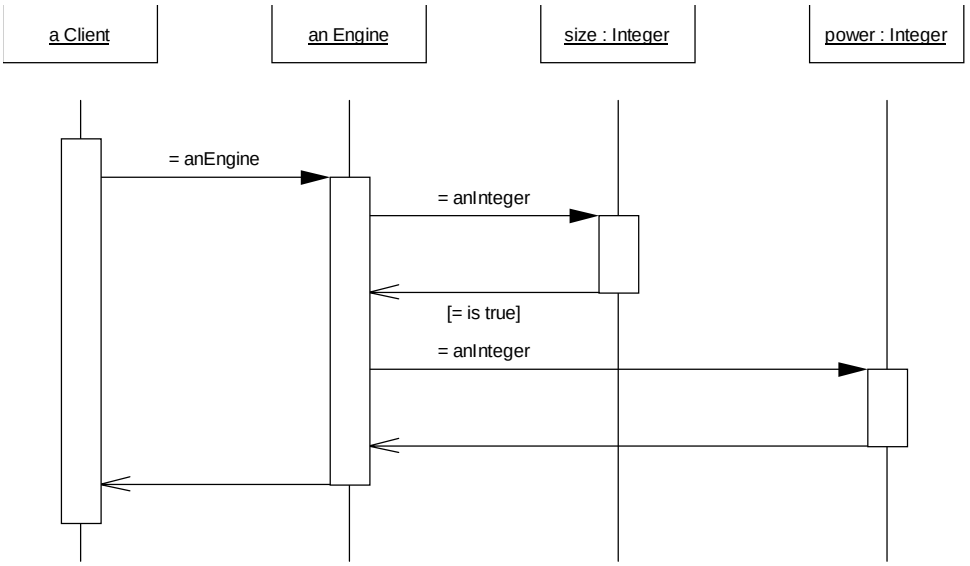
A more object-oriented approach is for the Comparer to tell the subjects to compare themselves and let them decide how to do that. This way, the Comparer is telling the subjects *what* to do, but not *how* to do it. With this approach, the Comparer object isn't even needed because any Client can simply ask one subject to compare itself with another.

So how do the subjects compare themselves? One determines if the other is equivalent to itself. It considers which parts of its state must be equivalent, then compares those. Each of those parts considers which of its parts must be equivalent and compares those, and so on until all of the parts are simple objects. At each step, the comparison process is relatively simple. Even a highly complex object doesn't need to know how to compare itself entirely; it just needs to know what its parts are and they need to know how to compare themselves.

For example, consider an `Engine` object that knows its internal displacement and total horsepower. To compare it to another engine, a client must verify that both engines are the same size and same power. This diagram shows the classes involved and how to implement `=`:



The `Client` sends = to the first `Engine` with the second `Engine` as an argument. The first `Engine` compares itself to the second one by comparing their `size` and `power`. The parts are `Integer`s, simple objects that the native system can compare. If the parts were complex objects, they would continue the comparison process by comparing their parts. Eventually, the simplest parts are either equal or they're not.



This comparison algorithm, where an object compares itself to another by telling their parts to compare themselves and so on, is an example of Object Recursion. An implementation of a recursive message sends the same message to one or more of its related objects and so on. The message surfs through the linked structure until reaching objects that can simply implement the message and return the result.

Keys

A system that incorporates the Object Recursion pattern has the following features:

- Two polymorphic classes, one of which handles a request recursively and another which simply handles the request without recursing.
- A separate message, usually in a third class that is not polymorphic with the first two, to initiate the request.

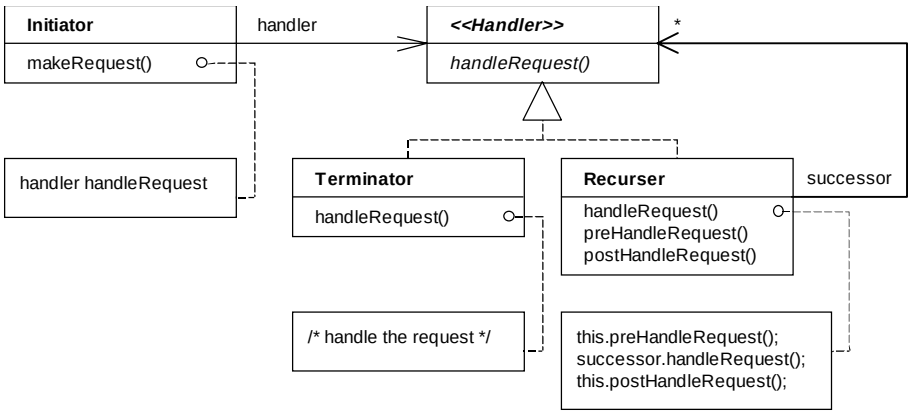
Applicability

Use the Object Recursion pattern when:

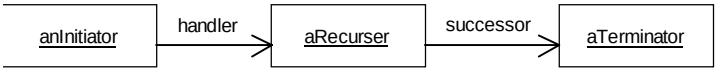
- passing a message through a linked structure where the ultimate destination is unknown.
- broadcasting a message to all nodes in part of a linked structure.
- distributing a behavior's responsibility throughout a linked structure.

Structure

The following class diagram shows the roles of the participants:



A typical object structure might look like this:



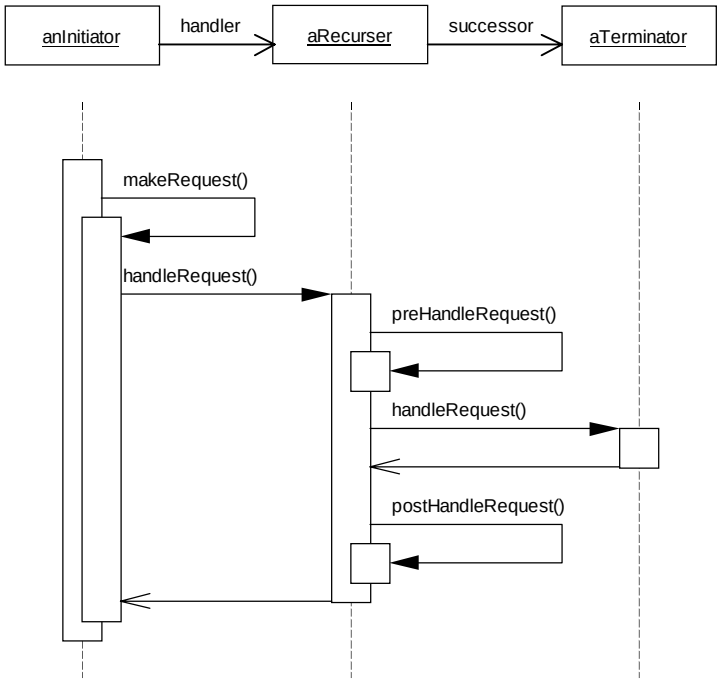
Participants

- **Initiator** (`Client`)
  - initiates the request.
  - usually not a subtype of Handler; makeRequest() is a separate message from handleRequest().
- **Handler** (`Comparable`)
  - defines a type that can handle requests that initiators make.
- **Recurser** (`Engine`)
  - defines the successor link.
  - handles a request by delegating it to its successors.
  - successors relevant to a request can vary by request.
  - can perform extra behavior before or after delegating the request.
  - may be a terminator for a different request.
- **Terminator** (`Integer`)
  - finishes the request by implementing it completely and not delegating any of its implementation.
  - may be a recurser for a different request.

Collaboration

- The Initiator needs to make a request. It asks its Handler to handle the request.

- When the Handler is a Recurser, it does whatever work it needs to do, asks its successor—another Handler—to handle the request, and returns a result based on the successor's result. The extra work can be done before and/or after delegating to the successor. If the Recurser has multiple successors, it delegates to each of them in turn, perhaps asynchronously.
- When the Handler is a Terminator, it handles the request without delegating the request to any other successors and returns the result (if any).



Consequences

The advantages of the Object Recursion pattern are:

- *Distributed processing.* The processing of the request is distributed across a structure of handlers that can be as numerous and arranged as complexly as necessary to best complete the task.
- *Responsibility flexibility.* The Initiator does not need to know how many Handlers there are, how they're arranged, or how the processing is distributed. It simply makes the request of its handler and lets the Handlers do the rest. The Handler arrangement can change at runtime to dynamically reconfigure the handling responsibilities.
- *Role flexibility.* A handler that acts like a Recurser for one request may act as a Terminator for another request, and visa versa.
- *Increased encapsulation.* Encapsulates the decisions for how to handle a request within the object doing the handling.

The disadvantages of the Object Recursion pattern are:

- *Programming complexity.* Recursion, procedural or object-oriented, is a difficult concept to grasp. Overuse can make a system more difficult to understand and maintain.

Implementation

There are a couple of issues to consider when implementing the Object Recursion pattern:

1. *Separate Initiator type.* The Initiator.makeRequest() message must not be polymorphic with the Recursor.handleRequest() message. Beginner programmers quickly learn that if a method has no

senders, it can be deleted. But they mistakenly assume that as long as a method has senders, it shouldn't be deleted. This is not true when the only senders of a method are other implementors of the same message, as is the case with object recursion. Unless there is another, non-polymorphic method to start off the recursion, none of the implementors will ever be run and they can all be deleted.

2. *Defining the successor.* The Recurser needs one or more successors, but the Terminator does not. If the Terminator implements the successor link (usually by inheriting it from the Handler), it ignores the link when implementing the handleRequest() messages. If all of the Terminator's messages ignore the successor link, then its value can always be null and the link is not needed.

Sample Code

Let's look at how to implement equals recursively.

All objects, no matter how complex, are ultimately composed of simple objects (i.e., primitives) such as integers, floats, booleans, and characters. Determining the equality of two simple objects (of the same type) is a trivial task performed natively by the operating system or CPU. For example:

```
5 == 5         // integer comparison (true)
5.25 == 5.15   // float comparison (false)
true == false  // boolean comparison (false)
'a' == 'b'     // character comparison (false)
```

There's no recursion here, but these simple comparisons form the terminating case for recursion.

Comparing two ordered collections is nearly as simple: Are each of the elements equal? Thus two strings are equal if each of their characters are equal. For example, look at the implementation of `String.equals(String)`:

```
public class String {
   private char value[];
   private int count;

   public boolean equals(String anotherString) {
      int n = count;
      if (n == anotherString.count) {
         char v1[] = value;
         char v2[] = anotherString.value;
         int i = 0;
         int j = 0;
         while (n-- != 0) {
            if (v1[i++] != v2[j++]) {
               return false;
            }
         }
         return true;
      }
      return false;
   }
}
```

Thus if each of the characters in the strings are equal, the strings are equal. For the purposes of implementing equality recursively, a string is a terminating object.

Now consider a name object that stores the first name and last name separately. Two names are equal if their first and last names are both equal:

```
public class PersonName {
   private String firstName, lastName;

   public boolean equals (PersonName anotherName) {
```

```
        return firstName.equals(anotherName.firstName)
           && (lastName.equals(anotherName.lastName));
    }
}
```

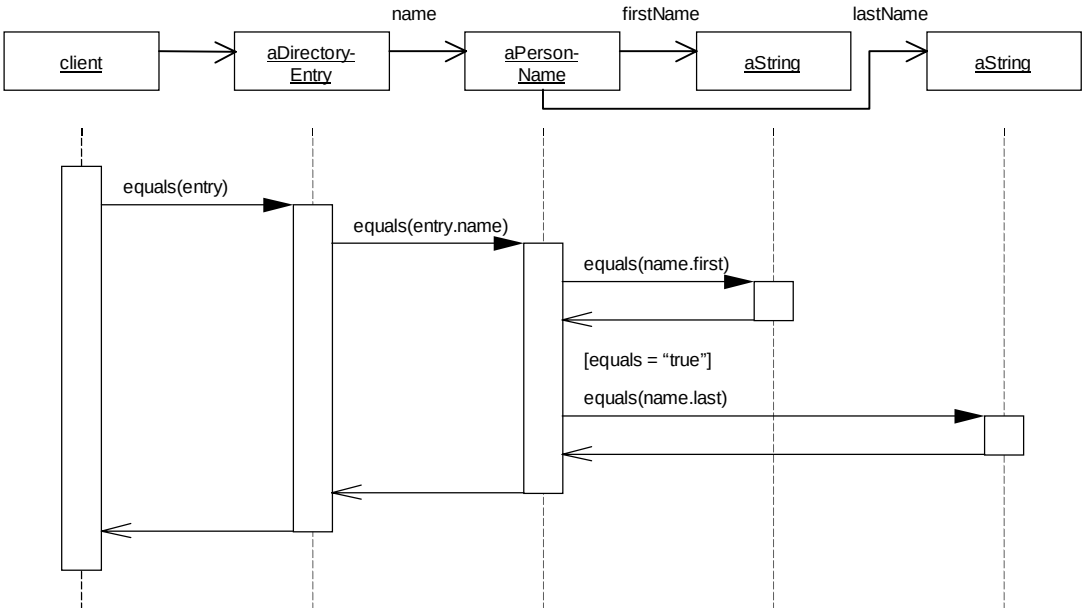
This contains one level of recursion—`PersonName.equals()` calls `String.equals()`—plus the primitive operation.

Now consider a phone directory object that stores a person's name and phone number. Two entries are considered duplicates if their names are equal:

```
public class DirectoryEntry {
   private PersonName name;

   public boolean equals (DirectoryEntry anotherEntry) {
      return name.equals(anotherEntry.name);
   }
}
```

Here we have two levels of recursion— `DirectoryEntry.equals()` calls `PersonName.equals()`, which calls `String.equals()`.



The client doesn't really care which implementor of `equals()` it's calling; it just knows that it has two objects of the same type and so it compares them using `equals()`. If the two objects were simple ones, there would be no recursion necessary. When the two objects are complex, `equals()` is implemented recursively using other implementors of `equals()`.

Known Uses

Numerous programming examples use Object Recursion.

The equality example described in the Motivation uses one-step recursion: The recursive implementation of the message sends the same message to its successor directly. Each implementor of equal also needs a corresponding implementor of hash, which is also implemented recursively.

A copy or clone message is often implemented using two-step recursion: The recursive implementation sends a second message to the receiver, which in turn sends the original message to its successor. Thus the two messages together implement the recursion. In the case of `copy()`, the

two messages are `simpleCopy()`—which only copies the root of the object—and `postCopy()`—which copies the object's parts as necessary. `copy()` sends `simpleCopy()` and `postCopy()`; `postCopy()` in turn sends the parts `copy()`, propagating the recursion.

Serialization algorithms, whether they produce text or binary output, usually use recursion. The algorithm serializes an object by serializing the root and then recursively serializing its (persistent) parts. Each branch of the recursion ends when a simple object serializes itself and is finished.

An algorithm to display an object as a string (e.g., Java's `toString()` and Smalltalk's `printString`) is usually recursive. The algorithm displays the root as a string, then recursively tells the interesting parts to display themselves.

A tree structure can use recursion to pass a message from any one of its leaves up to its root. Each node in the path recursively passes the message to its parent, doing any extra work along the way as necessary. The tree can similarly use recursion to broadcast a message from its root through all of its nodes to its leaves. These recursive techniques are frequently used in graphics trees, for example, to register invalidation requests and broadcast redisplay opportunities.

See the Related Patterns section for more known uses.

Related Patterns

Object Recursion vs. Composite and Decorator

When a Decorator [GHJV95, p. 175] delegates to its component or a Composite [GHJV95, p. 163] delegates to its children, this might be considered an example of Object Recursion. However, Composite and Decorator are structural (data structure) patterns, whereas Object Recursion is a behavioral (algorithm) pattern. If the structural patterns do embody recursion, it is only explicitly one level deep (a composite or decorator delegating to its component), whereas recursion's depth is unlimited. At best, Object Recursion is a pattern that both Composite and Decorator contain, but that can also be used independently of Composite or Decorator.

Object Recursion vs. Chain of Responsibility

Chain of Responsibility [GHJV95, p. 223] contains the Object Recursion pattern. Chain of Responsibility uses a linked-list or tree organized by specialization or priority. When a request is made, the structure uses Object Recursion to find an appropriate handler.

Object Recursion and Adapter

A chain of Adapters [GHJV95, p. 139] can seem to use a non-polymorphic form of Object Recursion, where the behavior is recursive even though the message isn't, as each adapter delegates to the next until the delegation terminates at the adaptee and unwinds. However, the lack of polymorphism goes against the spirit of Object Recursion.

Object Recursion and Interpreter

In Interpreter [GHJV95, p. 243], the `interpret()` message traverses the abstract syntax tree using Object Recursion. Client is the Initiator, AbstractExpression is the Handler, NonterminalExpression is the Recurser, and TerminalExpression is the Terminator.

Object Recursion and Iterator

Some implementations of Iterator [GHJV95, p. 257] use Object Recursion. External iterators on any structure are not recursive; they use while loops instead. Internal iterators on array structures also use while loops.

An internal iterator on a linked-list structure is recursive. If the terminating node of the list is a real object with the same type as other list nodes, since the internal iteration makes the `next()` and `isDone()` messages private, the algorithm is object recursion. If the end of the list is marked by null, the recursion is procedural.

An internal iterator on a branching structure (i.e., a tree or graph) must be implemented recursively. If the terminating points—the leaves and/or root—are objects, then the recursion is object recursion.

Object Recursion and Delegation

Other patterns contain an object that implements a message by delegating the same message to a collaborator of the same type: Proxy is a good example. Any such example is a one-level-deep example of Object Recursion, which is not a very interesting example, but a simple example nonetheless.

References

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

Acknowledgments

Thanks to Eugene Wallingford for his help in writing this paper.