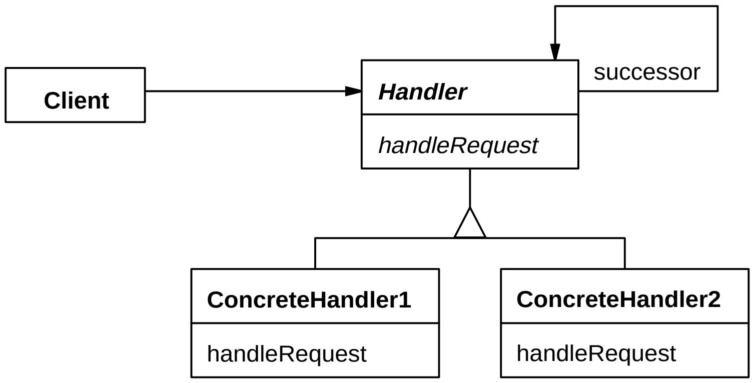
CHAIN OF RESPONSIBILITY (DP 223) Object Behavioral

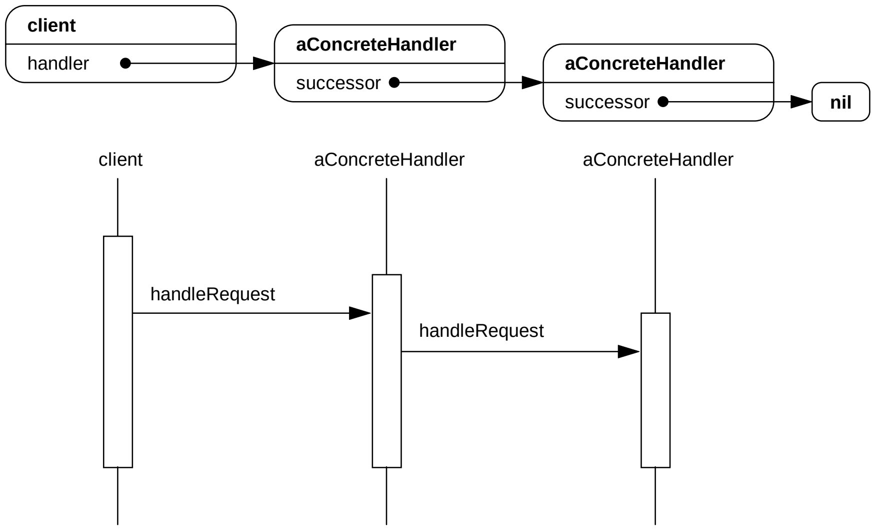
Intent

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Structure



A typical object structure and message interaction might look like this:



Discussion

Chain of Responsibility is the object-oriented version of recursion, also called *recursive delegation*.[1] In procedural recursion, a function calls itself with a different parameter each time. Eventually the parameter is a base case that the function simply performs, and then the recursion unwinds back to the original call. In recursive delegation, a method polymorphically sends its message to a different receiver. Eventually the method invoked is a base case implementation that simply performs the task, and then the recursion unwinds back to the original message send. (Beck, 1997) Chain of Responsibility explains how to design object structures that use recursive delegation.

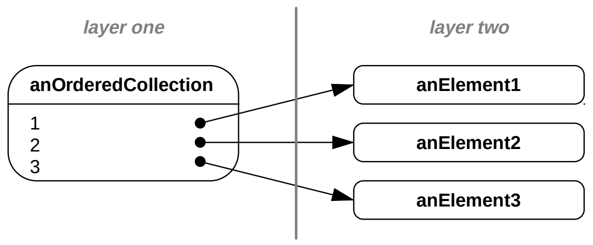
The key to Chain of Responsibility is a linked series of objects that recursively pass responsibility for an operation from one object to the next. When a client makes a request of one of the objects in the series, that object does its part to implement the behavior, then passes the buck to the next object in the series. Each object does its part and passes the buck until eventually an object in the series completes the behavior and returns the result. The client receives the result without knowing which objects contributed to it.

Tree Chains

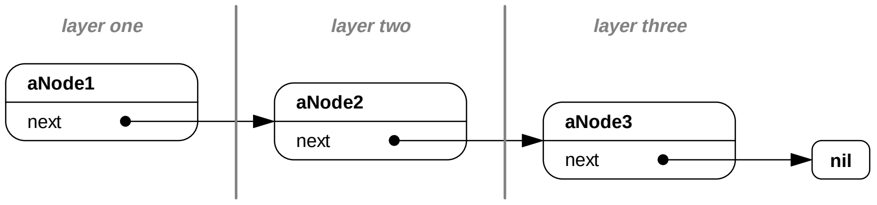
Chain of Responsibility occurs most often in Smalltalk in trees, because trees are much more common in Smalltalk than linked-lists. The structures are related because the path between the root node in a tree and any leaf node is a linked list. A tree can be linked in three different ways: the parent points to its children, each child points to its parent, or both. (See the discussion of tree structures in Composite.) The chain is formed by the pointers, so the Chain of Responsibility can only travel in the direction of the pointers.

A linked-list or tree might seem like just another way to collect elements, but these linked structures are quite different from a `Collection`. A `Collection` is a two-layer structure where one object, the `Collection` itself, knows what all of the elements in the structure are. Thus is can easily list how many elements there are, what each of them is, and so forth. A `Collection` and its elements are related like this:

[1] Not all recursive techniques in Smalltalk use Chain of Responsibility, and not all examples of the pattern are recursive, but recursive delegation is the most common form of object-oriented recursion and is the Chain of Responsibility pattern.



A linked-list or a path in a tree is a multi-layer structure where no one node in the structure knows what all of the other nodes are. Instead, each node knows that the next node is, so they can all be traversed eventually. But this makes common tasks like determining how many elements there are and what each one is a multiple step process. This is how the nodes in a linked list are related:



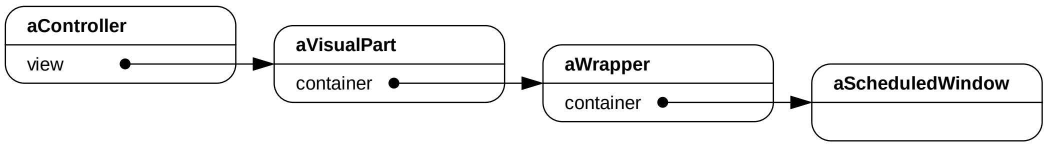
The elements in the linked-list form a chain that Chain of Responsibility recurses through. The elements in the collection do not form a chain, so Chain of Responsibility cannot traverse the elements.

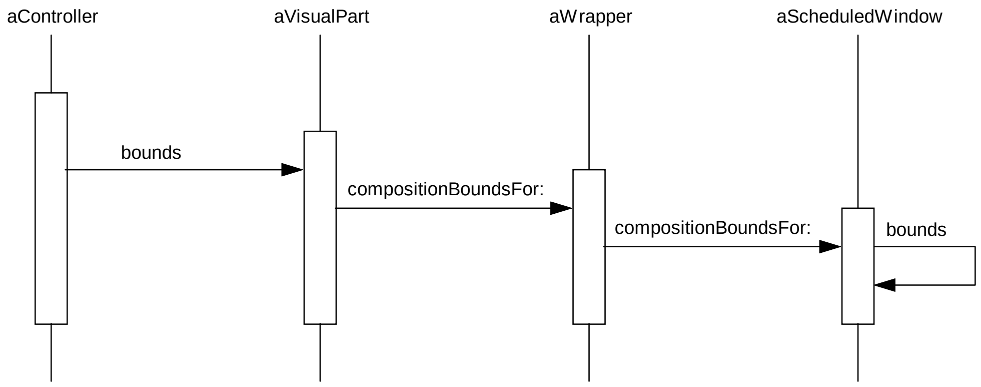
Visual Trees

The visual tree structure used to implement a window in VisualWorks contains several examples of Chain of Responsibility. For more details about the structure of a visual tree, see Composite.

When a visual computes its bounds (`VisualComponent>>bounds`), it uses Chain of Responsibility. Most visuals will draw themselves however large or small they need to be to fill the area they have. Thus their preferred bounds are unlimited. Yet a visual's actual bounds are constrained by the bounds of its container visual, which is constrained by its container, and so on. Ultimately the window frame constrains the bounds of all of its components.

Because of this container relationship, a visual does not know its own bounds. Its bounds depend on what it's contained in and what the container's bounds are. The diagram below shows a typical example what happens when a client asks a visual for its bounds.





Let’s look at the implementation of the messages in the diagram. Sending `bounds` to the visual runs the implementor in `VisualPart`.

```
VisualPart>>bounds
   "Answer the receiver's compositionBounds if there
   is a container, otherwise answer preferredBounds."
   ^container == nil
      ifTrue: [self preferredBounds]
      ifFalse: [container compositionBoundsFor: self]
```

The instance diagram shows that the `VisualPart`’s container is not nil, so `bounds` sends `compositionBoundsFor:` to the container. The container is a `Wrapper`, and `Wrapper` inherits `compositionBoundsFor:` from `VisualPart`.

```
VisualPart>>compositionBoundsFor: aVisualPart
   "The receiver is a container for aVisualPart.
   An actual bounding rectangle is being searched for by
   aVisualPart. Forward to the receiver's container."
   ^container compositionBoundsFor: self
```

This implementor shows the recursion of `compositionBoundsFor:`, which travels up the visual tree via the container relationships. Eventually the recursion reaches a terminating case, a bounding visual such as the window itself. The bounding visual just computes the bounds.

```
ScheduledWindow>>compositionBoundsFor: aVisualComponent
   "Answer the receiver's bounds."
   ^self bounds
```

This finally returns the bounds for the window and the recursion unwinds.

This simple explanation would seem to imply that any visual's bounds are the same as its window's, but this is not the case. Many visual classes, especially `Wrapper`s (Decorator visuals), affect their components' bounds. For example, a `Translating-Wrapper` adjusts a visual's offset from its container by translating its coordinates. It does this is by specializing the implementation of `compositionBoundsFor:`.

```
TranslatingWrapper>>compositionBoundsFor: aVisualPart
    ^(container compositionBoundsFor: self)
        translatedBy: self translation negated
```

Invalidation and redisplay of a window's graphics also uses Chain of Responsibility. When a client tells a visual to `invalidate`, the message defined by `Visual-Part>>invalidateRectangle:repairNow:forComponent:` recurses up the tree until the window handles it. When a client tells the window to redisplay via `ScheduledWindow>>displayDamageEvent:`, the message defined by `Visual-Component>>displayOn:` recurses down the tree depth-first as each visual redisplays itself.

Finally, VisualWorks also uses Chain of Responsibility to determine which `Controller` in a visual tree wants control. In polling windows, `Scheduled-Window>>subViewWantingControl` initiates a recursion where `Visual-Part>>objectWantingControl` and `CompositePart>>component-WantingControl` alternate down the tree. In event-driven windows, `Visual-Part>>handlerForMouseEvent:` recurses down the visual tree as the controller searches for the receiver of a mouse event.

Field Description Trees

The File Reader (Woolf, 1997) reads record-oriented files using a tree of field format objects that describes a record's format. The framework contains a special stream class, `FormattedStream`, that combines a regular stream and a field format tree to read the file and map the records to domain objects. (See Decorator for more details on `FormattedStream`.)

As a `Stream`, `FormattedStream` implements `next` to read the next record from the file. `FormattedStream>>next` initiates the recursive message defined by `FieldFormatDescription>>readObjectFrom:into:`. The message recurses down the field format tree and reads the record into a domain object. The way `next` initiates the read request without knowing which field formats will handle the request is an example of Chain of Responsibility. The nodes in the field format tree form the chain. Each node fulfills its responsibility for reading its part of the record.

`FormattedStream` also implements `nextPut:` to write a domain object to the file as a record. `nextPut:` initiates the recursive message `FieldFormat-Description>>writeObjectIn:to:`. The message recurses down the field format tree and writes the domain object into the record. This recursive implementation of `nextPut:` is another example of Chain of Responsibility. The field format tree nodes are the tree and each fulfills its responsibility for writing the object into the record.

For more details about how the field format tree works, see the Sample Code section.

Interpreter Trees

An Interpreter traverses an expression tree to evaluate the regular expression that the tree represents (DP 244). To traverse the tree, a client sends an interpret message to the root node in the tree. The node interprets itself by interpreting its children, which in turn interpret their children. The way the interpret message recurses through the tree is an example of Chain of Responsibility.

Class Hierarchy

A less obvious example of Chain of Responsibility is messages implemented across classes in a hierarchy. The beauty of Chain of Responsibility is that the chain is composed of instances so that the chain can be adjusted dynamically (such as when a subview is added into a window that is already open). The class tree is not dynamic, it is static because it is difficult (if not impossible) to change at runtime. However, it is still a tree, a set of chains, and so it can still implement Chain of Responsibility.

The implementation of `initialize` is a good example of Chain of Responsibility in a class hierarchy. Imagine class `D` is a subclass of `C`, which is a subclass of `B`, which is a subclass of `A`, and that all four classes implement `initialize`. Each implementor would send “`super initialize`” except for the one in A.

```
A class>>new
   ^self basicNew initialize

A>>initialize
   ...

B>>initialize
   super initialize.
   ...

C>>initialize
   super initialize.
   ...

D>>initialize
   super initialize.
   ...
```

When a client creates an instance of D and new runs initialize, the implementor of initialize in D will run the one in C, which in turn runs the implementor in B, which runs the one in A. This way, each class takes responsibility for implementing its own part of the object. This is Chain of Responsibility.

Equal, Hash, and Copy

Many common messages in Smalltalk are often implemented using Chain of Responsibility. The messages equal (=), hash, and copy are three examples.

The default implementation of equal in Object is not very interesting. By default, equal is implemented as double-equal (==). However, structurally complex classes like those for domain objects often implement equal in a much more interesting way. For example, let's consider how a stereotypical Person object would implement equal. For the sake of the example, let's say that two Persons are equal if their names are equal.

```
Object subclass: #Person
   instanceVariables: 'name address phoneNumber'
   classVariables: ''
   poolVariables: ''

Person>>= aPerson
   "Two Persons are equal if their names are equal."
   (aPerson isKindOf: Person) ifFalse: [^false].
   ^self name = aPerson name

Object subclass: #PersonName
   instanceVariables: 'firstName lastName'
   classVariables: ''
   poolVariables: ''

PersonName>>= aPersonName
   "Two PersonNames are equal if their
   first names and last names are equal."
   (aPersonName isKindOf: PersonName) ifFalse: [^false].
   ^(self firstName = aPerson firstName)
      and: [self lastName = aPerson lastName]

String>>=
   "Two Strings are equal if their characters are equal."
   "This example is from VisualWorks."
   | size |
   aString isString ifFalse: [^false].
   aString isSymbol ifTrue: [^false].
   (size := self size) = aString size ifFalse: [^false].
   1 to: size do:
         [:index | (self at: index) = (aString at: index)
            ifFalse: [^false]].
   ^true
```

As this code shows, two Persons are equal if their PersonNames are equal, which are equal if their first and last name Strings are equal, which are equal if their Characters are equal. This delegation of equal from Person to PersonName to String to Character is Chain of Responsibility. Each object takes responsibility for its part of the computation and then delegates the rest of the responsibility to its appropriate attributes.

Any class that implements equal should also implement hash. (Beck, 1997; Woolf; 1996) A class' implementor of hash works by delegating to the same attributes it delegates equal to. For example, since our Person class delegates equal to its name attribute, it should do the same for hash:

```
Person>>hash
    "A Person's hash value is the same as its
    name's hash value."
    ^self name hash
```

Since PersonName uses two attributes for equality, both of them must be hashed and combined:

```
PersonName>>hash
    "A Person's hash value is a combination of its
    parts' hash value."
    ^self firstName hash bitXor: self lastName hash
```

So, like equal, Person implements hash using Chain of Responsibility, delegating to its PersonName, which delegates to its first and last names.

The basic algorithm for an object to create a deep (i.e., fully independent) copy of itself is this: the object copies itself and all of its parts, which copy themselves and all of their parts, and so on. This is Chain of Responsibility as well. For a fuller discussion of implementing copy, see the Prototype pattern.

Object-Oriented Trees

One of the most common and powerful techniques you can learn in object-oriented programming is "How to implement object-oriented trees." These trees combine three design patterns: Composite, Decorator, and Chain of Responsibility. Composite is the structural pattern that creates trees out of polymorphic nodes. Decorator refines the structure to enable additional behavior to be added to either a branch or leaf node. And Chain of Responsibility provides behavior to request that the tree perform a function without having to know the tree's structure.

Most Composite structures also contain Decorators. Because Composite breaks a Component's implementation into two subclasses, Composite and Leaf, it is difficult to add behavior to the Component through subclassing. Subclassing Component won't work because the subclass won't affect Composite or Leaf. Duplicate subclasses of Composite and Leaf are undesirable. Instead, Decorator is a flexible alternative to subclassing that allows additional behavior to be added to nodes in a structure dynamically. Thus by implementing the extra behavior in a Decorator subclass of Component, you can add the behavior as a Decorator to a Composite or a Leaf.

Not only are Composite and Decorator convenient to use together, their designs fit together well. This book's chapters on Composite and Decorator discuss how the Component superclass declares the subclasses' core interface. The discussions also caution against extending the core interface in subclasses because that can ruin the polymorphism that the client expects. Because of this, classes in either a Composite or Decorator class tend to have a fairly limited interface that nevertheless provides a full set of functionality. Narrowing the Component's interface to apply the first pattern (either Composite or Decorator) is difficult, but once it's been done, applying the second pattern is much easier. Usually Composite is applied first to define the tree structure. After that, applying Decorator is relatively easy because Composite has already defined Component's limited core interface.

It is difficult to request a tree to perform a function. As discussed earlier, a tree is not one object, it is a collection of objects arranged in a hierarchical fashion. Thus for the tree as a whole to perform a function, it must tell each of its nodes to perform that function. The procedure to perform the function on all of the tree's nodes cannot make many assumptions about the tree's overall structure since that structure can easily change. Thus the procedure should let the tree structure itself perform the function on each node. A procedure that defers to the tree structure is an example of the Chain of Responsibility pattern.

Thus the Composite, Decorator, and Chain of Responsibility patterns are often used together. Composite defines object-oriented trees. Having done so, Decorator and Chain of Responsibility enhance the tree's functionality and flexibility.

Applicability

Here is when you can and cannot use the Chain of Responsibility pattern:

- *Chain already exists.* The pattern does not create a chain where there was none. *Design Patterns* says it can (page 226), but it's more accurate to say that the chain comes from applying a Structural pattern like Composite or Decorator or by simply implementing a linked structure. The chain might even be a series of Adapters, although it wouldn't be polymorphic. The chain is either a linked-list or a tree. In a tree, each path between the root and a leaf—in either direction—is a linked-list. The pattern uses this existing chain to introduce the new Chain of Responsibility behavior.
- *Collections are not chains.* As discussed earlier, a `Collection` (e.g., an `OrderedCollection` or an `Array`) is not a chain in the sense of this pattern.

As the diagrams showed, in a chain, each node points to its successor. In a `Collection`, the elements are not aware of each other and so no element knows what the next one is. Chain of Responsibility cannot traverse a Collection structure; the client must use Iterator instead.

Implementation

There are several issues to consider when implementing the Chain of Responsibility pattern:

1. *Automatic forwarding in Smalltalk.* This technique, described in *Design Patterns* (page 229), seems simple but should be avoided. Rather than forwarding each message individually to its successor, a Handler can simply implement `doesNotUnderstand:` to forward any message that the Handler does not understand to its successor. This technique is extremely flexible, but also quite haphazard. For more details about this implementation technique and its dangers, see Proxy.

2. *Do work, pass the buck, or both? Design Patterns* implies that each handler in the chain either handles the request completely and stops the chaining, or passes the request to its successor without doing anything. Smalltalk commonly uses a third option where a handler will handle the request partially but also pass the request to its successor. This way the work of handling the request is spread through the handlers. For example, a `TranslatingWrapper` changes the bounds that are being computed but still passes the request to its container. Each implementor in a chain of `initialize` methods does some of the initialization.

3. *Placement of recursion methods.* The recursion in Chain of Responsibility consists of three methods: an initiator, a recurser, and a terminator. Since the terminator method is the base case of the recurser method, they are polymorphic. The initiator method usually cannot be polymorphic with the recurser and terminator. As shown in the following table, where you implement these messages depends on which direction the recursion is traveling in:

	method's class		
direction	*initiator*	*recurser*	*terminator*
up the tree	Leaf, Component, or Client	Component	Component
down the tree	Component or Client	Composite	Leaf

- *Recursing up a tree.* A path from a leaf node to the root node in a tree is a linked-list. The initiator node is a leaf, so the initiator method is implemented in the Leaf class. If the recursion can start with any node, the initiator method is implemented in the Component class. The initiator method can also be implemented in a Client class. Since the recursion can travel through any node, the recurser method is implemented in the Component class. Since any node can act as the root, the terminator method is also implemented in the Component class.

- *Recursing down a tree.* Recursing down a tree from the root node to the leaves traverses the entire tree, usually depth-first. Thus recursion branches when it hits a Composite node; i.e., it recurses the path for one child, unwinds, recurses the path for the next child, unwinds, and so forth for the rest of the children. If the recursion can start with any node, the initiator method is implemented in the Component. It can also be implemented in a Client class. The recurser method is implemented in the Composite class; it continues the recursion by running the recursion on each of its branches. The terminator method is implemented in the Leaf class; there is usually no reason to move it up into the Component class.

Sample Code

The File Reader (Woolf, 1996), discussed above, implements a couple of examples of Chain of Responsibility. It creates a tree to describe the mappings of a record's fields to a domain object's structure and uses a special stream to read the records in as domain objects.

As a `Stream`, `FormattedStream` implements `next` to read the next record from the file. `next` initiates the recursive message `readObjectFrom:into:` which recurses down the field format tree and reads the record into a domain object.

```
Stream subclass: #FormattedStream
   instanceVariables: 'dataStream streamFormat ...'
   classVariables: ''
   poolVariables: ''

FormattedStream>>next
   "See superimplementor."
   ...
   ^streamFormat readObjectFrom: dataStream

Object subclass: #StreamFormatDescription
   instanceVariables: 'dataFieldFormat resultChannel ...'
   classVariables: ''
   poolVariables: ''

StreamFormatDescription>>readObjectFrom: dataStream
   ...
   dataFieldFormat
      readObjectFrom: dataStream
      into: resultChannel.
   ^self result
```

```
Object subclass: #FieldFormatDescription
   instanceVariables: ''
   classVariables: '...'
   poolVariables: ''

FieldFormatDescription>>readObjectFrom: dataStream into:
aValueModel
   "Reads the field that the receiver describes from
   dataStream and stores the field's value in aValueModel."
   self subclassResponsibility
```

Each kind (subclass) of field format has a different procedure for reading its data out of the data stream. In the simplest case, a leaf field format reads the data from the next field, converts it, and stores it in the return object.

```
FieldFormatDescription subclass: #LeafFieldFormat
   instanceVariables: 'readSelector writeSelector'
   classVariables: ''
   poolVariables: ''

LeafFieldFormat>>readObjectFrom: dataStream into: aValueModel
   "See superimplementor."
   | bytes fieldValue |
   ...
   bytes := self readFieldFrom: dataStream.
   ...
   fieldValue := bytes perform: readSelector.
   aValueModel value: fieldValue
```

A composite field format reads itself out of the record by recursively reading each of its child fields out of the record.

```
FieldFormatDescription subclass: #CompositeFieldFormat
   instanceVariables: 'fieldFormats resultChannel
fieldAdaptors'
   classVariables: ''
   poolVariables: ''

CompositeFieldFormat>>readObjectFrom: dataStream into:
aValueModel
   "See superimplementor."
   ...
   self readFieldsFrom: dataStream
```

```
CompositeFieldFormat>>readFieldsFrom: dataStream
   "Read the fields out of dataStream into the result
object."
   1 to: self numberOfFields
      do:
         [:i |
         | field adaptor |
         field := self fieldAt: i.
         adaptor := fieldAdaptors at: i.
         field readObjectFrom: dataStream into: adaptor]
```

A decorator field format simply forwards the read request to its component.

```
FieldFormatDescription subclass: #FieldFormatDecorator
   instanceVariables: 'fieldFormat'
   classVariables: ''
   poolVariables: ''

FieldFormatDecorator>>readObjectFrom: dataStream into:
aValueModel
   "See superimplementor."
   fieldFormat readObjectFrom: dataStream into: aValueModel
```

`FormattedStream` also implements `nextPut:` to write a domain object as a record using the recursive message `writeObjectIn:to:`.

```
FormattedStream>>nextPut: anObject
   "See superimplementor."
   streamFormat writeObject: anObject to: dataStream

StreamFormatDescription>>writeObject: anObject to: dataStream
   ...
   resultChannel value: anObject.
   dataFieldFormat writeObjectIn: resultChannel to:
dataStream

FieldFormatDescription>>writeObjectIn: aValueModel to:
dataStream
   "Writes the value in aValueModel to soruceStream
   using the field format described by the receiver."
   self subclassResponsibility
```

The recursion of `writeObjectIn:to:` is implemented just like `readObject-From:into:`. Since it also recurses down the tree, its key implementors are in `CompositeFieldFormat` and `LeafFieldFormat`. `FieldFormatDecorator` just forwards the message.

The Sample Code section in Composite also demonstrates Chain of Responsibility.

Known Smalltalk Uses

Linked-list structures are rare in Smalltalk, but tree structures are common. Virtually every tree structure in Smalltalk uses Chain of Responsibility to distribute behavior.

Related Patterns

Chain of Responsibility and Composite, Decorator, and Adapter

When a Composite delegates a message to its children or a Decorator delegates to its component, this is Chain of Responsibility. Similarly, an Adapter delegating to its adaptee is also Chain of Responsibility, although the chain is not polymorphic. All of these examples are rather degenerate because one object delegating to its collaborator is not a very extensive chain.

The better example of Chain of Responsibility is when a series of Composites, Decorators, and Adapters form a chain. Then a message that starts at the beginning of the chain travels through the chain looking for handlers to handle it. The client doesn't know which nodes will handle the request. The Structural patterns form the chain, but the behavior of the message traversing down through the nodes is Chain of Responsibility.

Chain of Responsibility vs. Iterator

Iterator allows each node in a structure to be considered so that functions may be performed on it in isolation from the other nodes in the structure. The Iterator knows which node will handle each request; it is the node that is currently being iterated on. The Iterator ensures that each node is considered in turn.

Chain of Responsibility does not consider each of the structure's nodes in isolation. Instead, a client makes a request of one of the nodes and that node passes the request to other nodes until one of them handles it. All of the nodes in the structure may be involved, or only the first one may participate. Either way, the client is unaware of which nodes are ultimately involved.

Chain of Responsibility and Interpreter

Interpreter uses the interpret message to traverse an abstract syntax tree. This traversal is an example of Chain of Responsibility.