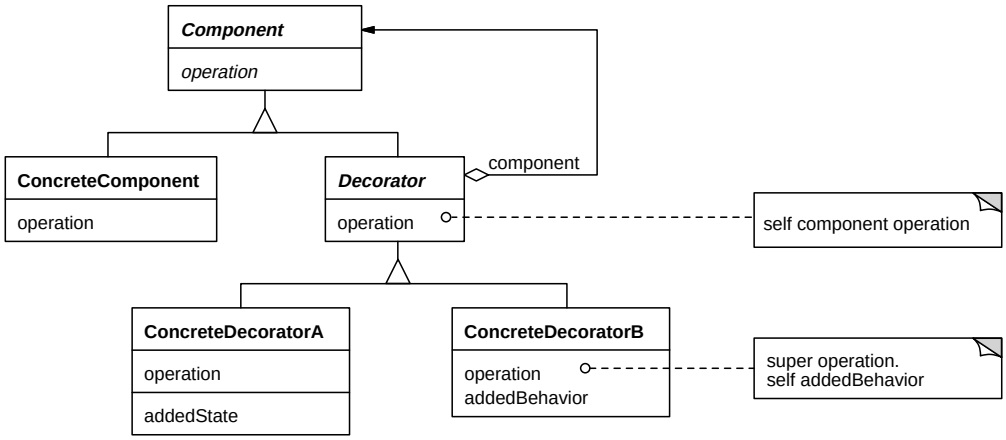
DECORATOR (DP 175) Object Structural

Intent

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Structure



Discussion

A class implemented as a Decorator does not provide core functionality itself. Instead, Decorator instances enhance the core functionality of the objects they decorate. A Decorator and the object it decorates (its Component) are of the same broad supertype, which is to say that they have the same core interface. Because the classes are polymorphic, the difference between a component and a decorated component is transparent to the client.

The way to recognize a Decorator class is that it has a single instance variable whose type is the same as the Decorator's. The Decorator delegates its implementation to this instance variable, but in the process it implements some of the messages to add extra behavior in addition to (or instead of) the straight delegation.

The name "decorator" seems to imply that this pattern can only be used with visuals, such as wrapping scrollbars around a text view, but the pattern is actually more versatile. It can be used to add functionality to any kind of object, not just a visual.

Systems Pattern

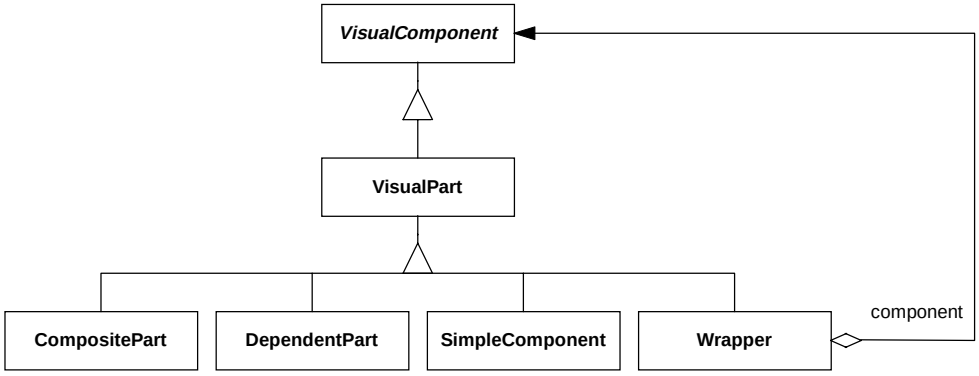
Decorator is a common pattern, but a fairly systems-oriented one, meaning that it is commonly used for implementing basic systems frameworks like windowing systems,

streams, and fonts. However, it is fairly uncommon for modeling domains, so most Smalltalk applications developers do not use it very often. Furthermore, VisualWorks uses Decorator to implement its base frameworks far more heavily than the other dialects do, so most Smalltalk examples are from VisualWorks. We will discuss one domain example, insurance caps, but first let's look at how Decorator works in general.

Wrapper Hierarchy

The `Wrapper` hierarchy in VisualWorks is a classic example of the Decorator pattern. VisualWorks implements graphical user interfaces using the Model-View-Controller (MVC) framework discussed in *Design Patterns* (pages 4-6) and in Factory Method. In Model-View-Controller, the Model contains the state that can be displayed, the View displays the state, and the Controller handles input that manipulates the state. The Model-View-Controller framework embodies several design patterns, including Observer, Composite, and Strategy, as well as Decorator.

The View in Model-View-Controller triad is implemented as the `VisualComponent` hierarchy; its main classes are shown below. `VisualComponent`'s main subhierarchy is `VisualPart`. `VisualPart` has leaf subclasses such as `DependentPart` and `SimpleComponent`, composition subclasses like `CompositePart` (an example of the Composite pattern), and Decorator subclasses like `Wrapper`.

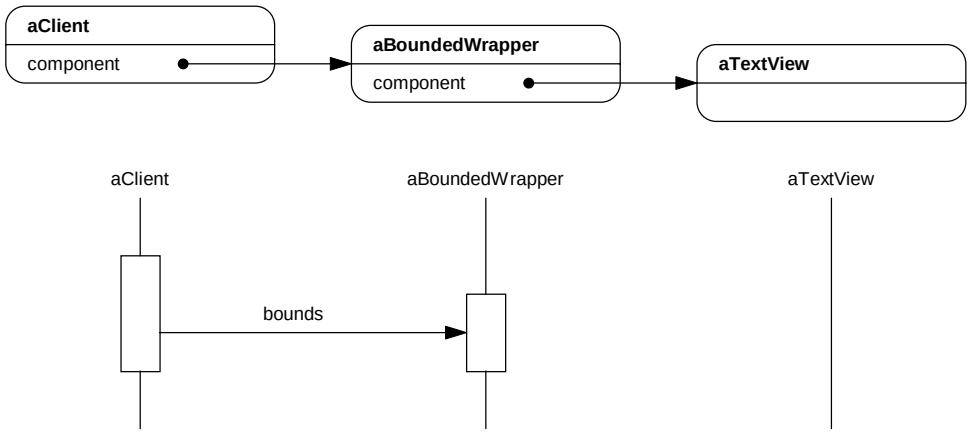


`Wrapper` itself does nothing except define the Decorator pattern. It has an instance variable called `component` that is a `VisualComponent`. Right away you know that something unusual is going on because `Wrapper` is a subclass of `VisualComponent` and yet its instance variable is also a `VisualComponent`. This forms a recursive structure of `VisualComponent`s (`Wrapper`s) whose components are in turn `VisualComponent`s. `Wrapper` subimplements various key messages from `VisualPart` and `VisualComponent`, but its implementation of those messages does little more than forward the same message to its component. That's all `Wrapper`

does. Thus a visual wrapped with a `Wrapper` behaves no differently than it would without the wrapper.

This lack of real behavior is typical for the top class in a Decorator hierarchy. The class will do little more than define an instance variable and override key messages to forward them to the instance variable. Decorator subclasses are implemented as typical subclasses—they subimplement their superclass to change its behavior. In the case of a Decorator subclass, it takes the top class' generic decoration behavior that does nothing and changes it to add specific decoration behavior. You see this in all Decorator hierarchies, not just `Wrapper`.

Once `Wrapper` has defined the Decorator pattern, its subclasses have a tremendous amount of freedom to affect the behavior of both the view and the controller. All the subclass has to do is change the implementation of the key messages it wishes to affect. It can choose from any of the messages that `Wrapper` forwards to its component. For example, a `BoundedWrapper` sets the bounds of its visual—the screen space allotted for it. By wrapping a visual with a `BoundedWrapper`, the visual's bounds become irrelevant because the wrapper sets the bounds. When a client asks the component what it's bounds are, the wrapper intercepts the message and returns its own bounds instead of its visual's. This message interaction is shown in the diagram below. In this way, you can control any `VisualComponent`'s bounds just by wrapping a `BoundedWrapper` around it.



There are other common uses for `Wrapper` subclasses. `ReversingWrapper` switches the foreground and background colors when drawing a visual to highlight it. `GraphicsAttributeWrapper` sets the graphics attributes of its visual such as its color and line width. `PassivityWrapper` can be toggled between enabled and disabled mode. When it's enabled, it forwards messages unaffected. But when it's disabled,

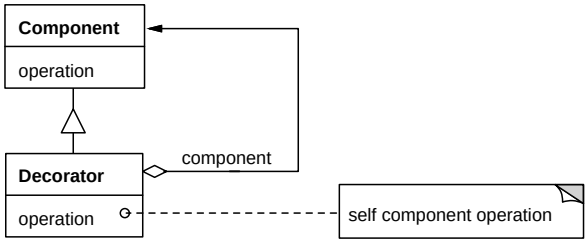
it makes its visual look disabled by drawing it grayed-out, and makes the visual act disabled by blocking all controller messages. Thus simply by introducing `Wrapper` as a Decorator class, it is now possible to control all kinds of properties of a visual simply by subclassing `Wrapper`.

Nested Decorators

A single Component often has multiple Decorators nested on it. For example, to add scrolling to a visual, VisualWorks wraps a `ScrollWrapper` around it. However, it usually also wraps a `BorderedWrapper` around the `ScrollWrapper`. `BorderedWrapper` is a subclass of `BoundedWrapper` that sets a visual's bounds and draws a border to outline those bounds. Wrapping a `BorderedWrapper` around a `ScrollWrapper` sets the bounds of the scrollable area and displays those bounds visually. The order of the Decorators is important. If the `ScrollWrapper` wrapped the `BorderedWrapper`, it would try to scroll the bounded rectangle; that would not work very well.

Combination Component/ConcreteComponent Class

Smalltalk implementations of the Decorator pattern often combine the Component and ConcreteComponent classes into a single class called Component as shown below. For example, in the `VisualComponent` hierarchy in VisualWorks, `VisualPart` plays the role of both the Component class and the ConcreteComponent class.

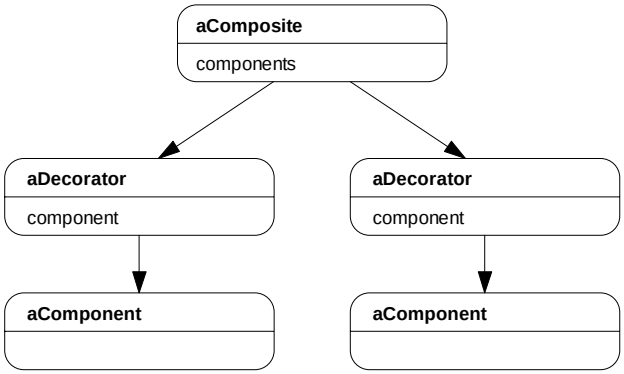


Combining Component and ConcreteComponent together into one class can cause problems. You can no longer introduce new behavior just for the ConcreteComponent classes without also introducing it into the Decorator classes as well. For example, in the `VisualComponent` hierarchy, to introduce behavior into all of the ConcreteComponent classes like `DependentPart` and `SimpleComponent`, you would want to add the behavior to `VisualPart`. But by adding the new behavior there, it will be inherited by non-ConcreteComponent classes like `CompositePart` and `Wrapper`.

However, often the ConcreteComponent class is empty because it doesn't have any separate behavior from the Component class. When this happens, ConcreteComponent is usually merged into Component to avoid implementing an empty subclass.

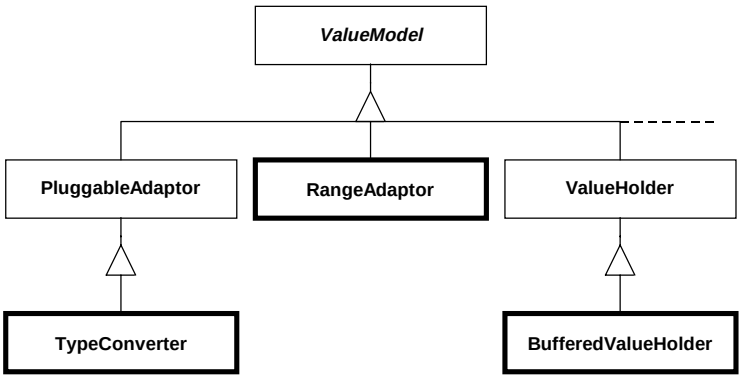
Decorator is a Subclass

One confusing aspect of the Decorator pattern is that the Decorator class is a subclass of the abstract Component class that it decorates. This is counterintuitive because a Decorator instance is a parent of the component it decorates. For example, the following diagram shows a typical tree structure with Decorators on the Components. Each Decorator is a parent of its Component; the component is the decorator's child. It then seems obvious to make the Component class a subclass of the Decorator class. Yet the Component class defines the interface that the Decorator class must fulfill, so the Component class is the superclass of the Decorator class.



No Decorator Subclass

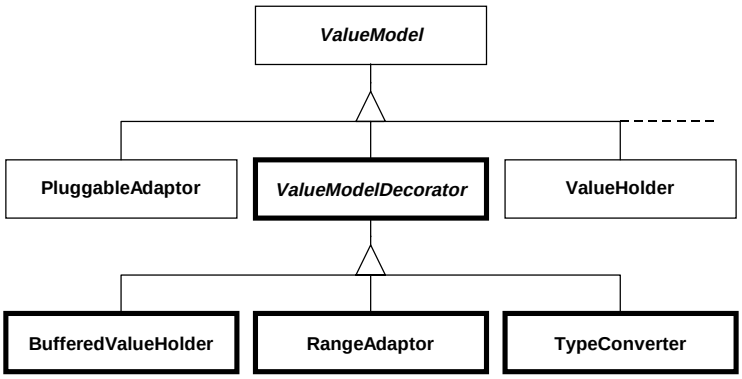
A strongly typed language like C++ virtually requires that the Decorator pattern be implemented as a separate Decorator subhierarchy in the Component hierarchy. Smalltalk does not force this constraint. For example, the `ValueModel` hierarchy in VisualWorks contains three Decorator classes: `BufferedValueHolder`, `RangeAdaptor`, and `TypeConverter`. Most `ValueModel` classes are Adapters on a `subject` where the subject can usually be any kind of `Object`, but these three `ValueModel` classes are Decorators because they expect their subjects to be other `ValueModel`s. As this diagram shows, the `ValueModel` Decorator classes are completely unrelated to each other in the hierarchy, yet they still work as Decorators.



However, even though Smalltalk does not require it, multiple Decorator classes in the same hierarchy should have a common Decorator superclass. Otherwise, each Concrete-Decorator class has to define its own Decorator behavior. For example, none of the subclasses in the `Wrapper` hierarchy have to define their Decorator behavior because they inherit that from their superclass, `Wrapper`. All a subclass has to do is define how it wants to decorate the visual. To the contrary, the three `ValueModel` Decorator classes duplicate behavior that would only have to be implemented once if they were subclasses of a common `ValueModelDecorator` superclass.

Another shortcoming of not having a common Decorator class is the difficulty in recognizing that the classes follow the Decorator pattern. For example, `BufferedValueHolder` looks like a `ValueHolder` with a buffer. This actually looks like an anti-example of the Decorator pattern, where subclassing was used instead of a Decorator. However, even though `BufferedValueHolder` is implemented as a subclass of `ValueHolder`, it does not have a `ValueHolder` built into it. Instead, it expects its subject to be a `ValueModel`—any kind of `ValueModel`—including a `ValueHolder`. So a `BufferedValueHolder` can be used to add a buffer to any `ValueModel`. That would be a lot more obvious if it were called "BufferedValue**Model**" and were a subclass in a Decorator branch instead of a subclass of a ConcreteComponent.

The following diagram shows a hypothetical `ValueModel` hierarchy that implements the Decorator pattern better. It introduces a `ValueModelDecorator` class to implement the Decorator pattern and makes `BufferedValueHolder`, `RangeAdaptor`, and `TypeConverter` its subclasses. Notice that it combines the Component and ConcreteComponent classes together as `ValueModel`.



`ValueModel` Decorators provide another example of the power of nesting `ValueModel`s. If a `ValueModel` needs a buffer, you wrap it with a `BufferedValueModel`. If it needs its type converted, you wrap it with a `TypeConverter`. Better still, to both buffer a value and change its type, you would use a `BufferedValueHolder` and a `TypeConverter`. Order is not as important in this case, but it is still significant. If the `BufferedValueHolder` decorates the `TypeConverter`, the converted value is buffered. The other way around, the unconverted value is buffered.

Decorator's Core Interface

The Component class must define the core interface that the Decorator and ConcreteComponent subclasses implement. For more information about what a core interface is and how to implement one, see Composite.

As with Composite, subclasses in a Decorator structure should avoid extending their Component's interface. The extended interface will contain more messages that the Component's interface such that a client will have to first determine the type of the receiver before it can safely use the extended interface.

Insurance Caps

Having said that Decorator is rarely used in modeling domains, here is a domain example. Decorator can be used in the insurance domain to cap the payments made to a policyholder for a claim. A health insurance policy would reimburse various medical procedures at various rates. However, the policy as a whole may have a maximum amount it will pay for a single claim. If all holders of this policy have the same cap, it can be built into the policy. But if various policyholders have the same policy but different caps, or if the cap is very difficult to build into the policy, the cap can be implemented as a Decorator on the policy.

`Policy` will have a method like `reimbursementForClaim:` that processes the claim and computes the reimbursement amount. Because different policyholders have different caps, this method will ignore all caps based on policyholder. Thus the reimbursement amount may be above the cap for some policyholders. To prevent this, each `Policyholder` does not hold a `Policy` directly, it holds a `PolicyCap` that holds the `Policy`. The cap also implements `reimbursementForClaim:`; it gets the reimbursement amount from the Policy, then tests it against the cap and returns whichever is less.

Here's some very brief example code for this hypothetical example. First you need to declare the Component class, the abstract superclass that declares the interface for both the ConcreteComponent and the Decorator. We'll call this Component `AbstractPolicy` and declare the main message we're interested in, `reimbursementForClaim:`.

```
Object subclass: #AbstractPolicy
   instanceVariables: ''
   classVariables: ''
   poolVariables: ''

AbstractPolicy>>reimbursementForClaim: aClaim
   "Calculate how much money the policy will pay
   for aClaim and return that amount."
   ^self subclassResponsibility
```

Next we'll implement the ConcreteComponent class, `Policy`. It's the easy one to implement because it just acts the way a policy does. We won't show the code for `reimbursementForClaim:` because it can get rather complex and really has nothing to do with the Decorator pattern.

```
AbstractPolicy subclass: #Policy
   instanceVariables: '...'
   classVariables: ''
   poolVariables: ''

Policy>>reimbursementForClaim: aClaim
   "... code to calculate the reimbursement ..."
```

Finally, we'll implement the Decorator class, `PolicyCap`. Because it is a Decorator, it needs an instance variable to point to its Component. We'll call it `policy`. It also needs an instance variable to hold the amount of the reimbursement cap.

```
AbstractPolicy subclass: #PolicyCap
   instanceVariables: 'policy capAmount'
   classVariables: ''
   poolVariables: ''
```

The PolicyCap calculates the reimbursement amount in two steps. First, it asks its policy how much the reimbursement should be. Second, it returns either that amount or the cap, whichever is less.

```
PolicyCap>>reimbursementForClaim: aClaim
    | uncappedAmount cappedAmount |
    uncappedAmount := self policy reimbursementForClaim:
aClaim.
    cappedAmount := uncappedAmount min: self capAmount.
    ^cappedAmount
```

Other insurance functions could also be implemented as Decorators. For example, PolicyDeductable could be another Decorator that subtracts the policyholder's deductibles and copayments out of the reimbursement amount.

Stream Decorators

The Known Uses section of the Decorator pattern in *Design Patterns* discusses a StreamDecorator class (page 183). StreamDecorator has subclasses for compressing an ASCII stream and for converting 8-bit ASCII into 7-bit ASCII.

Smalltalk developers have discovered another use for Stream decorators. Streams are typically used to store characters and bytes, because that's what flat files hold. Yet Smalltalk stores complex objects, not just characters and bytes. How can those objects be stored in files?

VisualWorks provides a framework called BOSS (the Binary Object Streaming Service) for storing objects in files. The BOSSTransporter hierarchy is a set of stream-like classes for reading and writing objects. One subclass, BOSSReader, implements the next protocol; the other, BOSSWriter, implements the nextPut: protocol. Both use an instance variable called stream, either a ReadStream or a WriteStream, to implement their behavior. For example, BOSSWriter converts each object into a BOSSBytes—a special ByteArray—and then uses the WriteStream to write those bytes into a file.

BOSSTransporter is another example that a Decorator class does not have to be implemented in the same hierarchy as its Component classes. Both Stream and BOSSTransporter are subclasses of Object; that's all they have in common. This would not work very well in C++ because of it's strong typing: the client would have to know whether it was using a Stream or a BOSSTransporter. It could not switch between the two interchangeably because they're not in a single Component hierarchy. Java might be able to handle this if it declared a "streaming" interface and declared both Stream and BOSSTransporter to fulfill it. (Bridge discusses Java interfaces.)

Just because Smalltalk can handle Decorators and Components in completely separate hierarchies does not mean implementing Decorators this way is a good idea. By imple-

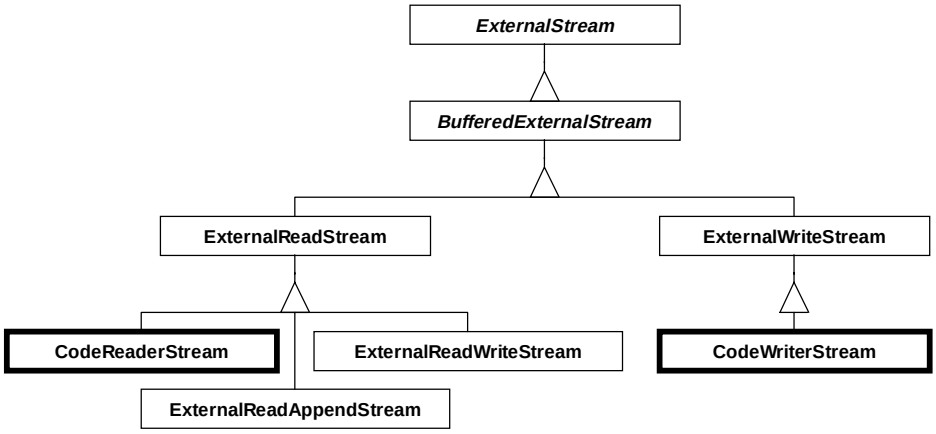
menting them in the same hierarchy, you're assuring that they will support the same interface so that they can be used interchangeably. This allows the client to ignore the details of whether it's using a Component or a Decorator on a Component. When the hierarchy is split in two like this, both hierarchies must be maintained to keep their interfaces polymorphic. It is always better to avoid duel maintenance when possible.

The File Reader framework (Woolf, 1996) contains a better example of a stream decorator. It implements `FormattedStream` as a subclass of `Stream`. As a `Stream`, `FormattedStream` implements the usual messages like `next`, `nextPut:`, and `atEnd`. Its main instance variable is `dataStream`, an instance of `Stream`. `FormattedStream` implements `nextPut:` to accept an object and uses a `StreamFormatDescription` to convert the object into a record. It then uses the `dataStream` to write that record into a file. `FormattedStream>>next` reverses the process to read a record out of a file and convert it back into an object. Because the `dataStream` can be any type of `Stream`, the "file" can be any object that can be streamed across.

Missed Opportunities

For every good opportunity where the Decorator pattern is used, there seems to be another where it might have been used but wasn't. The key to the Decorator pattern is that it is a flexible alternative to subclassing. Rather than adding behavior to a class by creating a subclass of it, we can add the behavior by creating a Decorator for it. Then that Decorator can be used to decorate any other class of the same type. Plus, Decorators can be nested, so rather than having to choose between a subclass or its peer, you can decorate with both.

There are numerous hierarchies that seem like they could benefit from a more flexible alternative than subclassing. For example, in the VisualWorks `Stream` hierarchy, `ExternalReadStream` has a subclass called `CodeReaderStream` and `ExternalWriteStream` has a subclass called `CodeWriterStream`. The following diagram shows these classes. Although their protocols are counterparts of each other, such reciprocal behavior is often encapsulated into a single class. But if there were a single `CodeStream` class, how could it be used with both `ExternalReadStream`s and `ExternalWriteStream`s? It should be implemented as a `Stream` Decorator, much like those in ET++ (DP 183).



For that matter, the orthogonal differentiation in the `Stream` hierarchy between read/write behavior and internal/external implementation is a subclassing nightmare. First the hierarchy splits into internal and external subhierarchies, then the subhierarchies duplicate the implementation for behaviors like read, write, append, and combinations thereof. Perhaps if read, write, and append were implemented as Decorators, they could be wrapped around internal and external streams in any combination desired.

Here's another example. `ApplicationModel` in VisualWorks has several abstract subclasses that add helpful behavior: `SimpleDialog` creates modal windows; `LensApplicationModel` interfaces with the Object Lens framework; third-party vendors have added their own abstract subclasses like `ValueInterface` (Abell, 1995). The problem is, what if you want a modal window that is Lens enabled? Do you subclass `SimpleDialog` or `LensApplicationModel`? If these classes were implemented as `ApplicationModel` Decorators, you could easily use both. A regular window would be declared as a subclass of `ApplicationModel`. If an instance of the window needed to be modal, wrap it with a `SimpleDialog` Decorator. If another instance needs to gather its data using the Lens, wrap it with a `LensApplicationModel` Decorator. If you need both, nest the two Decorators.

The `Collection` hierarchy is another example of inflexible subclassing that might benefit from the Decorator pattern. There are four basic ways to store a collection: as an array (`Array`, `OrderedCollection`), a linked list (`LinkedList` and `Link` in VisualWorks), a hash table (`Set`), or a (balanced binary) tree. Various `Collection` classes enhance these basic storage structures: duplicate elimination (`Set`), sorting (`SortedCollection`), and dependency notification (`List` in VisualWorks). The problem comes when you need a collection to preserve order but eliminate duplicates. You'll probably implement `OrderedSet` as a subclass of `OrderedCollection`.

But then if you also need a collection that sorts into order while removing duplicates, you'll also need to implement `SortedSet` as a subclass of `SortedCollection`. The code in `OrderedSet` and `SortedSet` will be very similar; the main difference will be their superclasses.

A more flexible solution would be to divide the `Collection` hierarchy into Concrete-Component and Decorator branches. ConcreteComponent classes would implement array, linked list, hash table, and perhaps tree. Decorator would implement duplicate elimination, sorting, and dependency notification. Then an "ordered set" would be an array with a duplicate elimination decorator. A "sorted set" would be an array with a duplicate elimination decorator on a sorting decorator. (In fact, an `OrderedCollection` might become an `Array` that is wrapped with a "growable collection" decorator.) This would be much more flexible than the subclassing the current `Collection` hierarchy uses, although perhaps less efficient. (See Bridge for IBM's approach to implementing collections.)

Implementation

There are several issues you should consider when implementing a Decorator:

1. *Use a Decorator superclass.* As noted earlier, Decorator classes do not have to be implemented in the same hierarchy as their Component classes. Even when they are, they do not have to be unified with a common Decorator superclass. However, you should always implement the Decorator pattern hierarchy as shown in the Structure diagram: a decoration-defining Decorator class at the top that is a subclass of the Component class.

   This obviously makes the decorator classes easier to implement. But it also makes your use of the Decoration pattern more obvious and easier to understand by anyone reviewing or maintaining your code. Again, look at the `ValueModel` Decorator classes scattered throughout the `ValueModel` hierarchy; do those look like Decorators? The `BOSSTransporter` hierarchy is completely separate from the `Stream` hierarchy; does it look like a `Stream` Decorator? The Decorator subclass of Component not only makes the pattern easier to implement, it makes the pattern easier to recognize and maintain.

2. *Consider a ConcreteComponent subclass.* Consider implementing a separate ConcreteComponent subclass of the Component class. This will divide the Component hierarchy into two distinct subhierarchies: Decorator—classes that can decorate; and ConcreteComponent—classes that can be decorated.

3. *Only Decorator delegates.* The only decorator class that should delegate to its component is the top class in the Decorator subhierarchy. All Decorator subclasses should defer their default behavior to their Decorator superclass and let it handle the delegation.

4. *Don't assume component is concrete.* Do not assume that a Decorator subclass will delegate directly to a ConcreteComponent. Since decorators can be nested, it may be delegating to another Decorator.

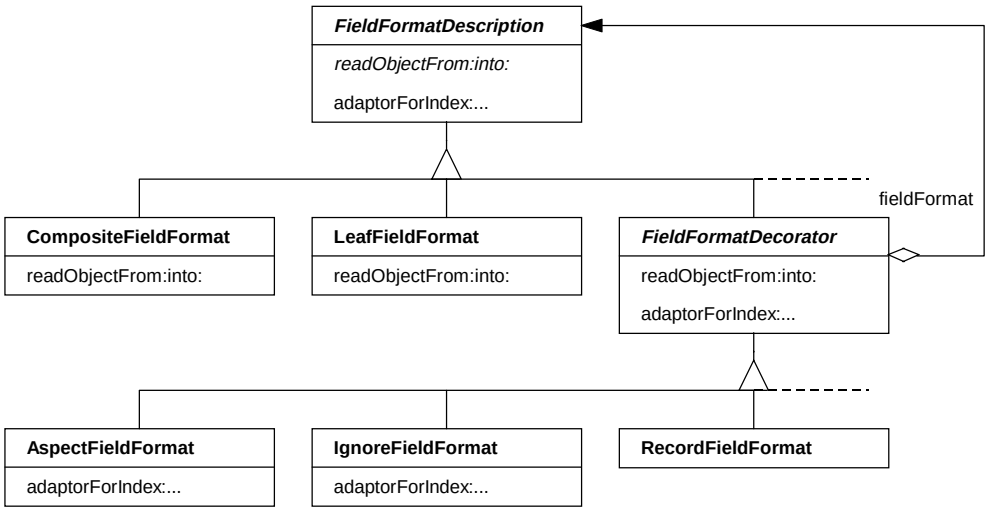
5. *Three ways to forward.* There are three ways for the Decorator to forward its operation messages to its component:

   - Simple forward — Send the message to the component without performing any other behavior.
   - Extended forward — Perform extra behavior before and/or after forwarding the message to the component.
   - Override — Perform behavior instead of forwarding the message to the component; this behavior may be to do nothing.

Sample Code

As discussed earlier, The File Reader (Woolf 1997) contains a `Stream` Decorator class called `FormattedStream`. The File Reader's main hierarchy, `FieldFormatDescription`, also contains a textbook example of the Decorator pattern.

A `FieldFormatDescription` codifies how a field should be read from a file. It knows whether the field is delimited or fixed-length, what the delimiter or length is, etc. The main things a description knows how to do is read and write the field. Its three subclasses are `LeafFieldFormat`, `CompositeFieldFormat` (an example of the Composite pattern), and `FieldFormatDecorator`. This diagram shows the hierarchy.



Let's look at how it's implemented. The superclass, `FieldFormatDescription`, declares the core interface for the entire hierarchy. It fulfills the role of the Component class. As an abstract class, it introduces basic messages like `readObjectFrom:into:` and `adaptorForIndex:andSubjectChannel:`.

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

FieldFormatDescription>>readObjectFrom: dataStream into:
aValueModel
   self subclassResponsibility

FieldFormatDescription>>adaptorForIndex: anInteger
andSubjectChannel: aValueModel
   ^(IndexedAdaptor subjectChannel: aValueModel)
      forIndex: anInteger
```

`FieldFormatDecorator` is a subclass of `FieldFormatDescription` that implements the Decorator class. It has an instance variable called "fieldFormat" that acts as the pointer to the Decorator's component. It implements the core `FieldFormatDescription` messages to delegate them to its `fieldFormat`.

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

```
FieldFormatDecorator>>readObjectFrom: dataStream into:
aValueModel
   fieldFormat
      readObjectFrom: dataStream
      into: aValueModel

FieldFormatDecorator>>adaptorForIndex: anInteger
andSubjectChannel: aValueModel
   ^fieldFormat
      adaptorForIndex: anInteger
      andSubjectChannel: aValueModel
```

There are many helpful behaviors that can be added to a field description in the process of reading and writing a field, and the hierarchy implements those behaviors as decorators. `RecordFieldFormat` enhances a description (usually a `CompositeFieldFormat`) to expect a record delimiter at the end. It does this by overriding `readObjectFrom:into:` to read the record into a separate stream and read the field(s) from it. Notice that it does not delegate to its `fieldFormat` directly; it lets its superclass do that via `super>>readObjectFrom:into:`.

```
RecordFieldFormat>>readObjectFrom: dataStream into:
aValueModel
   | recordStream |
   recordStream :=
      (dataStream upTo: recordDelimiter) readStream.
   super readObjectFrom: recordStream into: aValueModel
```

`AspectFieldFormat` maps the field to a domain object's aspect. `IgnoreFieldFormat` maps a field's value to the bit bucket. They both do this by overriding `adaptorForIndex:andSubjectChannel:`. By default, the framework reads a record into an `Array` and uses `IndexedAdaptor`s (a kind of `ValueModel`) to map field values into array slots. This default is implemented in `FieldFormatDescription>>adaptorForIndex:andSubjectChannel:` (shown above). `AspectFieldFormat` overrides this to use an `AspectAdaptor`; `IgnoreFieldFormat` uses a `ValueHolder`.

```
AspectFieldFormat>>adaptorForIndex: anInteger
andSubjectChannel: aValueModel
   ^(AspectAdaptor subjectChannel: aValueModel)
      forAspect: aspect

IgnoreFieldFormat>>adaptorForIndex: anInteger
andSubjectChannel: aValueModel
   ^ValueHolder new
```

Thus any field or group of fields can be treated as a record. It can also be mapped to an aspect or ignored. The decorators can be nested to map a record to an aspect or ignore it.

Since it doesn't make sense to both map a field to an aspect and ignore it, code in `AspectFieldFormat` and `IgnoreFieldFormat` prevents both of those decorators from being applied to a single field.

Known Smalltalk Uses

Wrapper

The `Wrapper` hierarchy in VisualWorks is discussed above. `Wrapper` subclasses are Decorators that can be added to `VisualComponent`s.

ValueModels

Several of the `ValueModel` classes in VisualWorks are decorators, as discussed above.

Stream Decorators

`BOSSTransporter` in VisualWorks is a `Stream` decorator. `FormattedStream` in The File Reader framework is also a `Stream` decorator. Both examples are discussed above.

FieldFormatDecorator

`FieldFormatDecorator` is a `FieldFormatDescription` decorator in The File Reader framework. It is discussed in the Sample Code section.

SyntheticFont

`SyntheticFont` is a decorator in the `ImplementationFont` hierarchy in VisualWorks. An `ImplementationFont` maps a Smalltalk font to one of the fonts built into the operating system. (This is an Adaptor.) A `SyntheticFont` adds properties to the font that the platform font may not support. For example, a `SyntheticFont` has a "strikethrough" setting. When strikethrough is on, the `SyntheticFont` draws its characters by drawing them first with the `ImplementationFont` and then drawing a line through them to look like this: ~~example of strikethrough~~.

Related Patterns

Decorator and Adapter

Decorator and Adaptor are often confused. Both are called "Wrapper" because they wrap another object to change it. However, a Decorator preserves the interface of its Component. An Adaptor specifically converts the interface of its Adaptee into the interface that its Client expects. A Decorator changes or adds behavior to its Component; otherwise, it has no value. An Adaptor can change or add behavior to its Adaptee, but its primary purpose it to convert its interface. If an Adaptor has the same interface as its Adaptee, it has no value. Decorators can easily be nested because they have the same interface.

Adaptors cannot be nested easily because the interfaces would have to alternate from one to another to another.

Decorator and Proxy

Decorator and Proxy are often confused. A Decorator changes or extends the behavior of its Component while preserving its Component's interface. A Proxy controls access to its Subject while preserving its Subject's interface. A Decorator always allows access to its Component. A Proxy does not change its Subject's behavior except to make it available or unavailable. Various Decorator subclasses represent different behaviors that can be added. Various Proxy subclasses represent different ways of controlling access.

For example, `CachedImage` is a class in the `PixelArray`/`Image` hierarchy in VisualWorks that looks like a Decorator but is really a Proxy. It "decorates" an image by caching the `Image` as a `Pixmap`, a form that is more efficient but less flexible. The reason `CachedImage` is not implemented as a subclass of `Image` is that `Image` is an abstract class with numerous concrete subclasses. By implementing `CachedImage` as a "Decorator," it can be wrapped around any `Image` subclass.

Yet unlike a Decorator, `CachedImage` does not add behavior, it simply adds efficiency. The initialization of the caching is only invoked the first time the `CachedImage` is used, not every time the way Decorator behavior would be. Most importantly, `CachedImage` fails the can-be-nested test: wrapping a `CachedImage` around another `CachedImage` adds no benefit. Thus `CachedImage` is a Proxy that diverts use of an `Image` to use a more efficient `Pixmap` instead.

Decorator, Composite, and Chain of Responsibility

Decorator and Composite are often used together in the same hierarchy. This is because both of them require limiting the interface of their Component to a core interface that a client can use with any node in the structure. Thus once the type's interface has be limited for one pattern, the other pattern can easily be applied to also support that core interface. Boiling down an extensive Component interface into a simplified core interface is difficult. Once that is done, either Decorator or Composite can easily be applied.

A Decorator communicates with its Component through a Chain of Responsibility. A series of nested Decorators culminating with a ConcreteComponent form a chain. When a message is sent to the top Decorator in such a chain, each Decorator decides whether to handle the message, forward it to its Component, or both. If the message reaches the ConcreteComponent, it is ultimately handled there.

See Chain of Responsibility for more details about using these three patterns together.