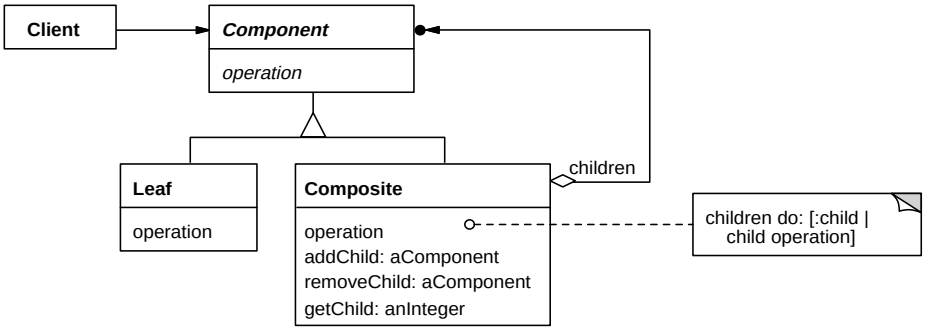
COMPOSITE (DP 163) Object Structural

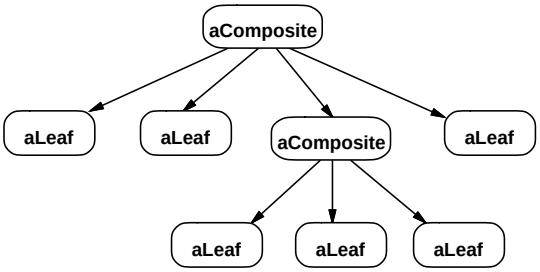
Intent

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Structure



A typical Composite object structure might look like this:



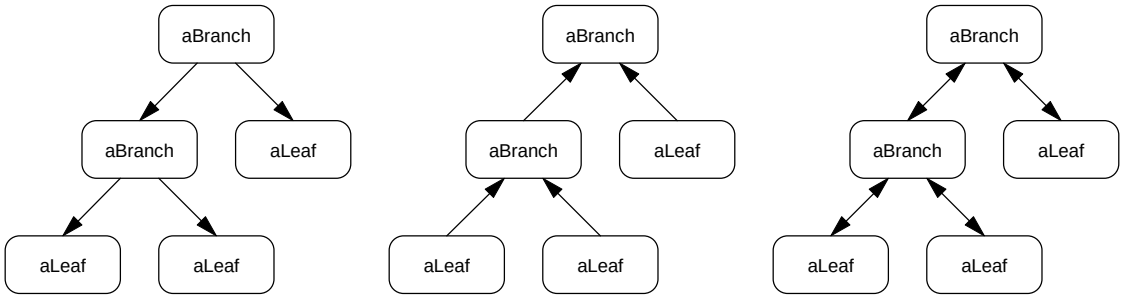
Discussion

The key to the Composite pattern is two classes—one that represents atomic objects and one that represents a group of such objects—that support the same core interface so that clients can treat both kinds of objects the same way. The group object, called a Composite, acts like an atomic object by delegating its behavior to the objects in the group. Because both kinds of objects support a single core interface, clients can collaborate with both kinds of objects interchangeably. The Composite itself is a client that takes advantage of the polymorphic core interface because each of its group members might be an atomic object, called a Leaf, or it might be another Composite. In this way, a Composite can contain other Composites and so on until the final Composites contain nothing but Leaves.

The Composite pattern implements a "tree" data structure. Tree is one of the most common and powerful data structures in procedural programming, and its importance applies to object programming as well. Any hierarchical, compositional relationship lends itself to being modeled as a tree: all of the various departments and offices within a government; the systems within a multiprocessor; the sales territories of a multinational corporation. The question is: In what ways do the levels in the hierarchy behave the same? What makes it recursive, almost fractal, so that every branch of the tree looks like the whole tree and in fact looks like its own branches? The issue is not what makes the levels and branches different, but in what makes them the same, because the properties that are the same are reusable.

Tree Structure

There are three ways to structure a tree, as shown below: top-down, bottom-up, and doubly-linked. In all three tree examples, the nodes and their relationships to each other are the same. However, their structures are different because of the direction of their pointers. In a top-down tree, each branch points to all of its children, so a client can ask a branch what its children are but a node does not know what its parent is. In a bottom-up tree, each node points to its parent, but a branch does not know what its children are. In a doubly-linked tree, each node knows both its parent and its children. The direction of the pointers determines what direction messages can travel through the tree—down, up, or both. (See Chain of Responsibility.)



top-down bottom-up doubly-linked

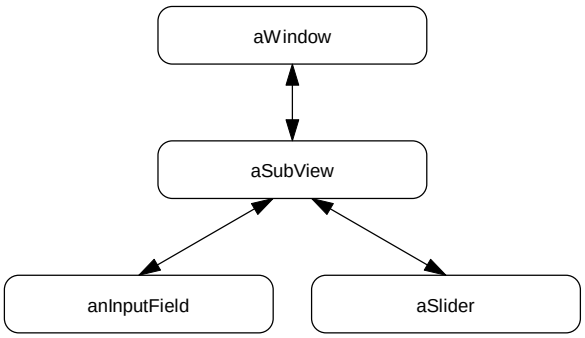
The Composite pattern defines the first type of tree, a top-down tree where the composite knows what its children are and the messages travel down the tree.

Window Trees

Perhaps the best known but seemingly unlikely example of a tree structure is a window in a windowing system. A window looks like a two-dimensional rectangle but is stored internally as a tree. The window itself is the tree's root, with the window's subviews and

their subviews comprising the branch nodes, ultimately culminating in widgets that are leaf nodes.

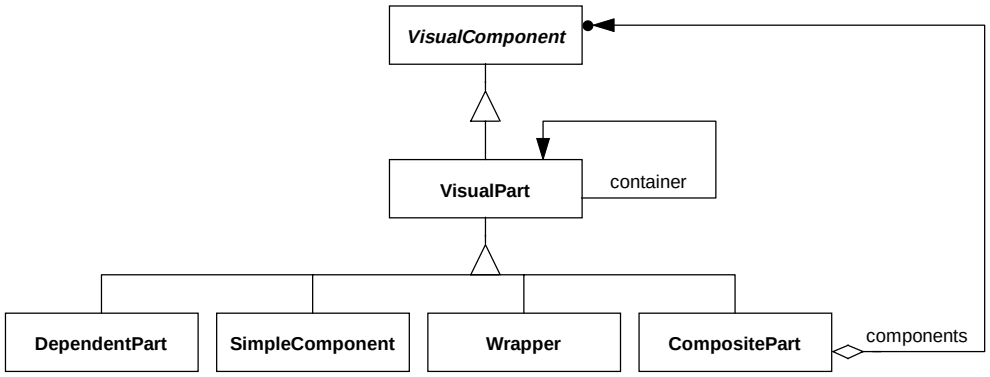
In VisualWorks, a window's tree is comprised of three distinct types of nodes: window, subview, and widget. This instance diagram shows the object structure of a simple window. In a tree of visuals, each visual refers to its parent node as its *container* and to its children nodes as its *components*. The structure is a tree (and not a graph) because each node has exactly one parent (except for the root which by definition has no parent).



The three different kinds of nodes are easy to recognize by looking at how many components each one has. The window node has just one component. The widgets are leaf nodes because they have no components. The subview node is a branch node because it has multiple components. Subviews are the composite in the Composite pattern.

In the Composite pattern, the Composite class and the Leaf class both have the same core interface, as defined by their common Component superclass. Thus a client object using a node in a tree does not have to distinguish between branch nodes and leaf nodes. They both have the same core interface and so the client can simply send the node messages and know that whatever kind of node it is, it will respond to the message appropriately.

These are the main classes in the `VisualComponent` hierarchy in VisualWorks. Here, the Component class is `VisualComponent`, `CompositePart` is the Composite class, and the Leaf classes are `DependentPart` and `SimpleComponent`.



These `VisualComponent` classes are the participants in the Composite and Decorator patterns:

Pattern	*Participant*	*VisualWorks Class*	*Visual Smalltalk Class*	*IBM Smalltalk Class*
Composite and Decorator	Component	`VisualComponent` and `VisualPart`	`SubPane`	`CwBasicWidget`
Composite and Decorator	Leaf / ConcreteComponent	`VisualComponent` and `VisualPart`	`SubPane`	`CwPrimitive`
Composite	Composite	`CompositePart`	`GroupPane`	`CwComposite`
Decorator	Decorator	`Wrapper`	none	none

A composite and a leaf have the same core interface that allow them to work polymorphically. For example, in VisualWorks, any visual knows its preferred bounds. A leaf visual simply knows how big it wants to be. A composite visual determines the preferred bounds of its components and merges these overlapping rectangles into a single container rectangle. Another example is the way all visuals know how to draw themselves. A leaf visual just draws itself. Most composite visuals are invisible, so they just draw themselves by telling their components to draw themselves. In this way, any branch of a visual tree, from a single node to the entire tree, can be treated as a single visual and told how to behave. How the behavior is produced is a function of the branch's structure, but that is hidden from the client object that makes the request.

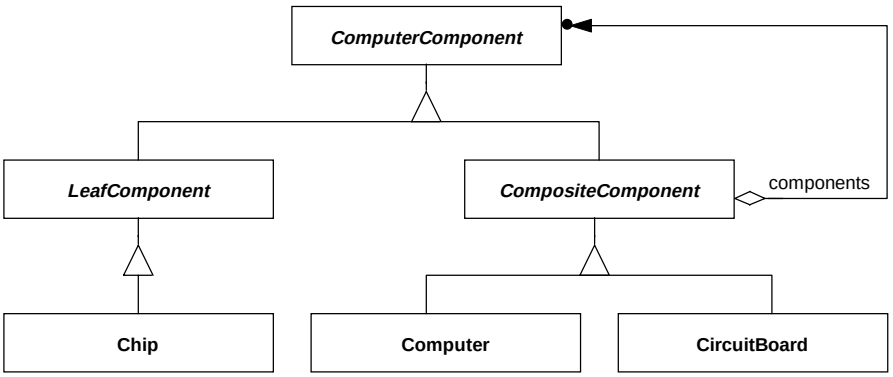
Visual Smalltalk and IBM Smalltalk implement graphical windows in a similar manner. Their classes are also listed in the table above. The composite visual classes are `GroupPane` in Visual Smalltalk and `CwComposite` in IBM Smalltalk. Both visual hierarchies are examples of the Composite pattern.

Nested Composites

A key advantage of the Composite pattern is not just that a Composite can be used to group together Leaves, but also that it can be used to group together other Composites as well. In other words, a Composite can contain other Composites, which in turn can contain other Composites, until finally all of the branches terminate with Leaves. This forms a recursive structure such that the tree can have as many levels as necessary and the difference between a tree and a branch is indistinguishable. The Structure object diagram (shown above) illustrates this well.

Limited Types of Children

In general, a Composite can contain both Composite and Leaf nodes. However, some domains restrict this. For example, a computer can be modeled to contain circuit boards that contain chips.



`Chip` is a Leaf class and `Computer` and `CircuitBoard` are Composite classes. The Composite class specifies that it can contain any Components, but its subclasses are more restrictive. A `Computer` can only contain `CircuitBoard`s and a `CircuitBoard` can only contain `Chip`s.

So `Computer` (which is a Composite) cannot directly contain a `Chip` (a Leaf class), even though the Composite pattern would normally allow it. Unrestricted application of the Composite pattern would allow computers to contain other computers and for a computer to contain both circuit boards and chips that are not part of any circuit board (which makes you wonder what those chips are mounted on). To prevent these invalid configurations that the Composite pattern would otherwise allow, the implementation of the `Computer` and `CircuitBoard` classes must contain constraint code that disallows them.

Thus the Composite pattern is so general as to allow any tree structure, even those that may not make sense from a domain perspective. If the domain contains constraints on what makes a valid tree, those constraints must be coded in the Composite class and its subclasses.

Limited Number of Children

In general, a Composite can have an unlimited number of children. Typically, a branch node doesn't care how many branches it has. Most domains allow this and even expect it. A composite visual doesn't care how many component visuals it contains. A computer usually doesn't limit how many circuit boards it can contain, nor is there a limit on the number of chips a circuit board can contain. A Composite's implementation supports this by using a `Collection` for its children. This way, the Composite can contain as many children as necessary.

However, some domains limit the number of child nodes a branch can have. A binary tree node cannot have an unlimited number of child nodes, just two at most. For example, to model a single-celled organism that reproduces by splitting into two organisms, you would track an organism, its pair of children, their children-pairs, and so on. Thus each organism has only two children, not an unlimited number of children.

The Composite pattern supports branches with a fixed number of children as well as an unlimited number. *Design Patterns* shows how the unlimited case is implemented using a `children` variable that is a `Collection`. To implement a limited number of children, implement a separate variable for each potential child. For example, if the Composite can only have two children, it should implement two variables like `leftChild` and `rightChild` or `mother` and `father`. The Composite would also implement the policy of what happens when one or more of the children are unspecified.

The Composite operations are also implemented differently in a Composite with a limited or fixed number of children. In the unlimited case, the children are stored in a `Collection`; a method iterates over them using a message like `do:` (see the Iterator pattern). With a limited number of children, the Composite operation accesses each child explicitly and performs the operation on it separately. If the operation has results, the Composite operation gathers those results and merges them.

For example, the typical code for an operation in a Composite with an unlimited number of children would be:

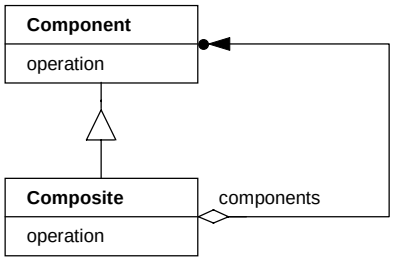
```
Composite>>operation
   self children do: [ :child | child operation]
```

The same operation in a Composite with exactly three children would be implemented as:

```
Composite>>operation
    "Assumes none of the children is nil."
    self firstChild operation.
    self secondChild operation.
    self thirdChild operation
```

Combination Component/Leaf Class

Smalltalk implementations of the Composite pattern often combine the Component and Leaf 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 Leaf class. In Visual Smalltalk, `Window` plays both roles.[1]



Combining Component and Leaf together into one class can cause problems. You can no longer introduce new behavior just for the Leaf classes without also introducing it into the Composite classes as well. For example, in the `VisualComponent` hierarchy, to introduce behavior into all of the Leaf classes like `DependentPart` and `Simple-Component`, you would want to add the behavior to `VisualPart`. But by adding the new behavior there, it will be inherited by non-Leaf classes like `CompositePart` and `Wrapper`. However, if the Leaf class is empty because it doesn't have any behavior that is different from the Component class, merge Leaf into Component to avoid implementing an empty subclass.

Composite is a Subclass

One potentially confusing aspect of the Composite pattern for novice designers is that the Composite class is a subclass of the Component class. This may seem counterintuitive because a Composite instance is a parent of the Components it contains. For example, the Structure object diagram at the beginning of this pattern shows a typical tree structure with Composites that contain Components. Each Composite is a parent of its Components; the Components are the Composite's children. It then seems obvious to make the Component class a subclass of the Composite class. Yet the Component class

[1] In practice, only certain subclasses of `Window` are used as Composites, but in theory, any subclass can be because they all have the `children` instance variable.

defines the interface that the Composite class must fulfill, so the Component class is the superclass of the Composite class. Try not to confuse part-whole object relationships with class inheritance.

No Component Superclass

Sometimes in Smalltalk, a Composite class and its Component class aren't even in the same hierarchy. In these cases, the Composite class is often part of the Collection hierarchy. For example, a `FontDescriptionBundle` in VisualWorks is a collection of `FontDescription`s. A `FontDescription` represents a platform-independent description of a font that will later be bound to a font on the runtime platform to display text in that font. `FontDescription` is a Proxy. `FontDescriptionBundle` is used to search a platform for multiple `FontDescription`s at once.

`FontDescriptionBundle` is clearly a composite `FontDescription`. Not only does the bundle contain multiple descriptions, but also the two classes share a core interface that allow them to be used interchangeably and polymorphically. For example, `FontDescription>>findMatchOn:allowance:` searches the platform's fonts for a suitable match. `FontDescriptionBundle` implements the same message to look for a suitable match for any of its fonts. The operation is less likely to fail with a `FontDescriptionBundle` because of the multiple search criteria.

Because the two classes share the same interface and serve the same purpose, we'd expect them to be parts of the same hierarchy. Yet they are not. `FontDescription` is a subclass of `Object` but `FontDescriptionBundle` is a subclass of `OrderedCollection`. The classes' lack of a common superclass makes it rather difficult to add new behavior to both classes polymorphically. When a developer implements new messages in `FontDescription`, he must remember to also implement them in `FontDescriptionBundle`—in a totally separate hierarchy—to preserve the classes' polymorphism. This would be simpler and less error-prone if the two classes had a common superclass that the developer could extend.

Component's Core Interface

Design Patterns suggests that the Component class should have both generic operations and ones that are Composite specific. This is reflected in the Structure diagram which shows that Component (not Composite) should declare messages like `Add()`, `Remove()`, and `GetChild()`. Implementation points 3-5 (DP 167-69) discuss this as a tradeoff between safety and transparency.

Design Patterns, being heavily weighted toward C++, favors transparency so that the type checking in C++ isn't too burdensome. If Component did not declare the Composite specific messages, code could not both treat a Composite as a more general Component and send it the Composite-specific messages. It would first have to cast the Component as specifically being a Composite before it could send those messages. Smalltalk

does not have these typing constraints, so it favors safety over transparency. Since only Composite can implement these messages in a useful way, they are not added to Component and thus not implemented in Leaf.

The transparency that *Design Patterns* favors is problematic. The question is: If Component is going to declare these composite messages, how should Leaf implement them? It could implement them to do nothing and return `nil`, or to fail and return errors. Safety says that a class should not understand messages that it cannot implement meaningfully. Transparency says that all Component subclasses should implement the same interface so that they can be used interchangeably.

Smalltalk solves this dilemma by employing a concept called a core interface. A *core interface* is one that does not declare enough behavior to implement all of an object's responsibilities, but one that is sufficient to support key collaborations. For example, the `Collection` classes `Set`, `OrderedCollection`, and `SortedCollection` all share two protocols that the proposed ANSI Smalltalk standard calls "extensible" and "contractible." (X3J20, 1996) This means that they understand messages like `add:` and `remove:`. This core interface is important because it allows these classes to be used interchangeably whereas `Array` could not be used in their place. However, the classes' full interfaces are not the same. `OrderedCollection` understands `at:put:` but `Set` and `SortedCollection` do not. `SortedCollection` implements `sortBlock:` to set that attribute, one that `OrderedCollection` and `Set` do not have. But these `Collection` classes do work enough alike that a client using their core add/remove interface does not need to know the collection's class or its full interface.

Using a core interface is a way of playing fast and lose with an object's protocol that is much more difficult in strongly-typed languages. In C++ or Java, a class' interface must be declared up front and the compiler ensures that all messages sent to an object are ones that its class implements. Smalltalk is dynamically-typed, so the compiler does not verify that an object understands the messages it is sent. However, at run time, if the object does not understand the message, a message-not-understood error occurs.

The Composite pattern wants to treat Composite and Leaf objects interchangeably. It wants to treat them both as Components. To accomplish this, in a strongly-typed language, the Component must declare any message that any subclass (Composite or Leaf) will implement. Thus it declares messages like `addChild:`, `removeChild:`, and `getChild:`, and then forces Leaf to try to implement them. In a dynamically-typed language like Smalltalk, the superclass (Component) need only declare the messages that all subclasses will implement. The tradeoff is that clients who wish to treat Composites and Leaves interchangeably must use only the core interface they share.

Thus the Structure diagram of the Composite pattern in Smalltalk is somewhat different than the one shown in *Design Patterns*. The Component in *Design Patterns* implements

composite messages like `Add()`, `Remove()`, and `GetChild()`. When implementing the pattern in Smalltalk, the Component usually does not define these messages because they are not appropriate for all subclasses. Component in Smalltalk only defines messages that are appropriate for all subclasses, such as `operation`. Then these are the only messages that the Client can use transparently. To use the composite messages, the Client must first determine that it is collaborating with a Composite, not just any Component.

Implementing the Core Interface

As discussed above, the Component class defines a core interface that the Composite and Leaf subclasses implement. Clients collaborate with Components through this core interface so they can use Composites and Leaves interchangeably. For this to work successfully, care must be taken in implementing the core interface.

First, implement the interface itself in the Component class with Template Methods. This will define all of the interface's messages in terms of a small number of kernel messages. The Component should use as few kernel messages as possible because both the Composite and Leaf classes will have to implement all of these messages.

Second, defer implementation of the kernel message in the Component class. It should defer implementation of these messages to subclasses by implementing them as `subclassResponsibility` or `implementedBySubclass`.

Third, implement the kernel messages in the Composite and Leaf subclasses. In the Leaf class, the message simply performs its behavior. In the Composite class, the message forwards the message to each of its children and merges their behavior.

Fourth, avoid extending the core interface in any subclasses of Composite or Leaf. These subclasses can have extended interfaces, but because they are not part of Component's interface, client objects will not be able to use this extended protocol polymorphically. Thus to use the extended interface, the client will first have to determine the type of the receiver. This goes against the spirit of the Composite pattern.

This fourth directive is especially difficult. It means that all subclasses essentially need to have the same interface as the Component class. Thus the Component class needs to declare not only its own interface, but also an interface extensive enough to satisfy all subclasses as well. In practice, the messages that subclasses can still add are instance creation messages. When a client is creating an instance, it knows the instance's specific class and uses its extended interface instead of Component's.

For example, with the Composite class, the add-child messages are extensions, but are used as part of instance creation when the client knows that it just created a Composite. Similarly, a client checks to see if a node has a child it wants to remove, so it knows that the node must be a Composite before sending the remove-child message. If the node weren't a Composite, it couldn't contain the child in the first place.

Field Format Description Trees

Another rather unintuitive example of the Composite pattern occurs when reading record-oriented flat files. Each record is composed of fields, where each field is a simple type such as a string or a number. Each file record corresponds to a record construct defined in a procedural language. In an object-oriented language, the record construct corresponds to a class and each set of record data in the file corresponds to an instance of the class.

Thus a framework for reading record-oriented flat files requires at least two classes, `Field` and `Record`, where `Record` is a collection of `Field`s. These two classes will work much better if they're implemented using the Composite pattern. Then they can implement a message like `readFromStream:` polymorphically. `Field` implements it to read the next field from the file stream. `Record` implements it to read each of its fields from the stream, plus it reads its record delimiter if it has one. Thus a client can use the same message to read a single field or a complete record without regard to which one it's actually reading.

The Composite pattern allows the framework to start nesting file structures. It does not make much sense for one record to be nested inside of another. However, it does make sense for a common series of fields to be nested inside another common series of fields. This is essentially a sub-record nested in a record. For example, the fields for a Customer might contain the fields for his Address. This corresponds to a domain object like `Address` being nested as a single object inside another domain object like `Customer`. Thus a `Record` is really just a `CompositeField` with an optional record delimiter. The true Composite class is `CompositeField` which acts like a single field but actually contains multiple `Field`s. This allows nested object to be stored in the Record as nested fields.

Missed Opportunities

Occasionally you find a class that would be easier to use if it had been implemented using the Composite pattern. The key to the Composite pattern is two polymorphic classes where one class is a collection of the other. If a client has to distinguish between an object and a collection of them, it would benefit from the Composite pattern. If a client has to distinguish between objects with two different interfaces, it would benefit from a pattern that introduces polymorphism. If one of the objects is a collection of the other, Composite would be the best pattern.

For example, `SelectionInList` in VisualWorks is a very useful class that tracks both a list of objects and which one is currently selected. It is implemented using two `ValueModel`s, one that holds the collection and another that holds the selection's index in the collection. `SelectionInList` is often used alongside `ValueModel`s, where some aspect values are stored in `ValueModel`s and others are the current selec-

tion in a list of possible values. However, SelectionInLists and ValueModels cannot be used interchangeably because they have different interfaces.

SelectionInList and ValueModel should be interchangeable. This would require giving them the same polymorphic interface. SelectionInList should have the same interface as ValueModel, where its value is its selection. Then any widget could use a ValueModel without regard to whether it was a SelectionInList.

Thus some pattern should be applied to SelectionInList and ValueModel to make them polymorphic. Since SelectionInList is composed of two ValueModels, and because it should behave like a ValueModel, applying the Composite pattern will cause the proper transformation. It will move SelectionInList into the ValueModel hierarchy and implement its ValueModel interface by delegating to its component ValueModels and merging their behaviors together.

With SelectionInList in the ValueModel hierarchy, it would have to implement the standard ValueModel messages like value. Here's an example of how that would be implemented:

```
SelectionInList>>value
    | list selectionIndex |
    list := self listHolder value.
    selectionIndex := self selectionIndexHolder value.
    ^selectionIndex = 0
        ifTrue: [nil]
        ifFalse: [list at: selectionIndex]
```

Notice how SelectionInList, a Composite, implements value by delegating the message to each of its component ValueModels and merging their results together (in this case using at:). This is classic Composite behavior.

Implementation

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

1. *Use a Component superclass.* A Composite subclass does not have to be implemented in the same hierarchy as its Leaf class. However, the pattern will be much easier the implement, recognize, and use when both classes are implemented in a single Component hierarchy. This makes their polymorphism easier to define and maintain.

2. *Consider a Leaf subclass.* Consider implementing a separate Leaf subclass of the Component class. This will divide the Component hierarchy into two distinct subhierarchies: Composite—classes that are composed of Components; and Leaf—classes that cannot be decomposed.

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

4. *Nesting.* Composites can be nested. Not only can a Composite contain Leaves, it can also contain other Composites. Programmers often forget this and implement the Composite to assume that its children are Leaves. The first time a client inserts a Composite into another Composite as a child, the top Composite fails because of an oversight in its implementation.

5. *Kinds of children.* Can the Composite contain any type of child? This is a domain-specific issue, not an implementation issue. But if the design contains constraints on which types of objects can be contained within Composite objects, the Composites' implementations must enforce these constraints through validation code. For example, an implementor of `addChild:` may need to verify the child's type before adding it.

6. *Number of children.* Is the Composite's number of children limited? If so, use a separate variable to store each child. Otherwise, use a single `Collection` variable that can store an unlimited number of children. If the limit is large and the children do not have individual roles within the Composite, use the `Collection` variable and enforce the limit in the `addChild:` method. Implement the Composite operations accordingly.

7. *Four ways to forward.* There are four ways for the Composite to forward its operation messages to its component:

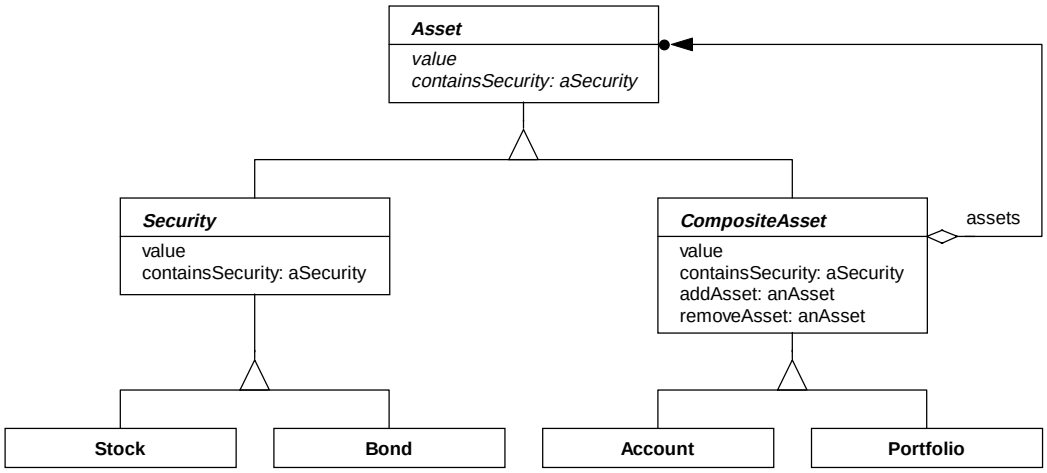
    - Simple forward — Send the message to all of the children and merge the results without performing any other behavior.
    - Selective forward — Conditionally forward the message to only some of the children and merge the results.
    - Extended forward — Perform extra behavior before and/or after forwarding the message to some or all of the children and merging the results.
    - Override — Perform behavior instead of forwarding the message to the children; this behavior may be to do nothing.

Sample Code

Let's look at a typical example from the financial domain, an Account that contains Securities. An Account is composed of its Securities. Similarly, a client's Portfolio is composed of its Accounts. A Security can be a Stock, a Bond, or some other type of asset.

A client will often want to know the value of his Portfolio. That is determined by summing the value of all of his Securities in all of his Accounts. A client might also want to know if he owns a particular Security. That is determined by searching for that Securities in all of the Accounts in his Portfolio.

The basic object model for this domain would look like this.



To implement this object model, first we'll implement `Asset` as the Component class. It does not need any variables. It declares two messages, `value` and `containsSecurity:`, that its subclasses should implement.

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

Asset>>value
   "Return the value of this Asset."
   ^self subclassResponsibility

Asset>>containsSecurity: aSecurity
   "Answer whether this Asset contains aSecurity."
   ^self subclassResponsibility
```

Next we'll implement `Security`. It will need an instance variable to store its `value`. Since it can also implement `value` and `containsSecurity:`, it does so.

```
Asset subclass: #Security
   instanceVariables: 'value'
   classVariables: ''
   poolVariables: ''

Security>>value
   "See superimplementor."
   ^value

Security>>containsSecurity: aSecurity
   "See superimplementor."
   "For a Leaf, we'll say it includes aSecurity
   if it is aSecurity."
   ^self = aSecurity
```

Now we'll implement the Composite class, `CompositeAsset`. It needs an instance variable, `assets`, to store its children and a method to access this variable. It also implements `value` and `containsSecurity:` as composite operations.

```
Asset subclass: #CompositeAsset
   instanceVariables: 'assets'
   classVariables: ''
   poolVariables: ''

CompositeAsset>>assets
   "Return the list of assets."
   ^assets

CompositeAsset>>value
   "See superimplementor."
   "Return the sum of the assets."
   ^self assets
      inject: 0
      into: [ :sum :asset | sum + asset value]

CompositeAsset>>containsSecurity: aSecurity
   "See superimplementor."
   "See if one of the assets is aSecutiry."
   ^self assets includes: aSecurity
```

Look closely at the implementation of `containsSecurity:`. It assumes that `assets` is a `Collection` of `Security`s. That will usually be true when the Composite is an `Account`, but not when it's a `Portfolio`. Thus this implementation will always return `false` for a `Portfolio`, but this failure will not be obvious or easy to find. Remember, a Composite does not just contain Leaves; it can also contain other Composites. So `containsSecurity:` must be implemented as though the assets may be other Composites.

```
CompositeAsset>>containsSecurity: aSecurity
   "See superimplementor."
   "See if one of the assets is aSecutiry."
   self assets
      detect: [ :asset | asset containsSecurity: aSecurity]
      ifNone: [^false].
   ^true
```

This improved implementation of `containsSecurity:` is an example of Chain of Responsibility, as is the implementation of `value`. Notice that implementors in Composite should use Chain of Responsibility, that the problem with the original implementation of `containsSecurity:` is that it used includes: and so did not use Chain of Responsibility.

Known Smalltalk Uses

Collection

`Collection` itself is the ultimate example of the Composite pattern in Smalltalk, and also the least useful. It implements the pattern with the Component class (`Object`), the Composite class (`Collection`), and numerous Leaf classes (all other subclasses of `Object`). An element in a `Collection` can be another `Collection`.

`Object` defines operations that `Collection` reimplements as composite operations. For example, `Object` defines `printString` to display the receiver's class. `Collection` reimplements it to display not just the `Collection`'s class, but also the `printString`s of the elements in the `Collection`. However, `Object` does not have a very customized interface, so neither does `Collection`. This is the least domain specific example of the Composite pattern.

Visuals

The major Smalltalk dialects use the Composite pattern to implement their windowing systems, as discussed earlier.

Field Format Descriptions

The File Reader (Woolf, 1997) reads record-oriented flat files by reading each record one field at a time, as discussed earlier. The ability of a group of fields to act like a single field is an example of the Composite pattern. It is implemented with three classes: `FieldFormatDescription` (Component) and two subclasses, `LeafFieldFormat` (Leaf) and `CompositeFieldFormat` (Composite).

Media Elements

The Composite pattern is used in EFX, a digital video editing and special effects environment (Alpert et al., 1995), to uniformly represent aggregate and primitive media elements. EFX's user interface incorporates a horizontal timeline for specifying the

content of a video composition. The user adds rectangular bars that represent media elements—video clips, audio segments, and special effects. Their position and width along the timeline determines when they play and for how long. The editor provides multiple tracks that enable multiple media to co-occur in time it.

At any time, multiple media elements in any number of tracks can be selected and grouped. The result is a single group element containing the other elements, and visually portrayed as a grouped media element in a single track. Such aggregate elements understand the same core protocol as individual media elements—all know how draw themselves in the timeline, all provide trimming methods to increase/decrease their time span, all implement accessor messages for their start time and duration, and so on. The group element also supports nesting so that primitive and group elements may be aggregated together into larger group elements.

Sentence Structures

ParCE, a natural language parser (Alpert & Rosson, 1992), represents sentence constituents as objects. It parses a sentence into a parse tree. In the tree, non-terminal constituent objects (Composites) represent the sentence, noun phrases, prepositional phrases, and so on. They ultimately contain the appropriate terminal constituent objects (Leaves representing nouns, verbs, determiners, etc.). All of these constituent objects—the individual words, the phrases, and the sentences—respond to the same message protocol for parsing, printing, accessing, and so on.

Brokerage Accounts

Frameworks that model the financial domain often implement a Composite class such as `Account` to represent brokerage accounts. Classes like `Asset` and `Security` define the Component and Leaf classes. `Account` defines a Composite asset that is composed of other `Asset`s. Other Composite assets might be customer `Portfolio`, a broker's `BrokerAccountsUnderManagement`, the branch's `BranchAccountsUnderManagement`, etc. See the Sample Code section for details.

SignalCollection

`SignalCollection` is a Composite class in the `Collection` hierarchy in VisualWorks. It does not *contain* a collection of `Signal`s; it *is* a collection of `Signal`s.

VisualWorks implements exception handling with two separate classes, `Signal` and `Exception`. `Signal` describes the types of errors that can occur. `Exception` describes a specific error. An `Exception` refers to its `Signal` to know what kind of error occurred. This is an example of the Type Object pattern. (Johnson & Woolf, in press)

To trap an `Exception`, the developer specifies the `Signal` whose `Exception`s should be trapped. `Signal`s form a hierarchy, so a `Signal` will trap any `Exception`s for it or any of its child `Signal`s. However, to trap multiple `Signal`s from

different parts of the hierarchy, the programmer has to specify each `Signal` individually and tell the whole list to trap `Exception`s. He does this by creating a `SignalCollection` that contains the list of `Signal` roots, then tells it to trap `Exception`s just as if it were a single `Signal`.

Because `Signal` and `SignalCollection` have the same interface for trapping `Exception`s, and because `SignalCollection` works by delegating its behavior to its `Signal`s, these two classes are an example of the Composite pattern. As with any example where the Composite class is implemented in the `Collection` hierarchy, this example would be better implemented if `Signal` and `SignalCollection` were implemented in the same hierarchy.

Composite ProgramNodes

`SequenceNode` is a Composite class in the `ProgramNode` hierarchy in VisualWorks. Smalltalk compiles source code into a tree of parse nodes that are `ProgramNode`s. There are different `ProgramNode` subclasses for various programming constructs, such as `LiteralNode`, `BlockNode`, `AssignmentNode`, and `ConditionalNode`. `SequenceNode` represents a series of statements (i.e., lines of code) within a method or a block. Each of these statements is itself a `ProgramNode`, so this is an example of the Composite pattern.

A more subtle example of Composite in the `ProgramNode` hierarchy is `ConditionalNode`. A `ConditionalNode` has three main parts: `condition`, `trueBody`, and `falseBody`. They form an `ifTrue:ifFalse:` statement like this:

```
condition ifTrue: trueBody ifFalse: falseBody
```

Each of these parts is itself a `ProgramNode`. `ConditionalNode` is an example of a Composite with a limited number of children, in this case three children variables for its three main parts.

Other examples of Composite in the `ProgramNode` hierarchy are: `ArithmeticLoopNode`, `AssignmentNode`, `CascadeNode`, `LoopNode`, `SimpleMessageNode`, and `MessageNode`. All of these classes contain multiple `ProgramNode`s and combine them together to act like a single `ProgramNode`.

CompositeFont

`CompositeFont` is a Composite class in the `ImplementationFont` hierarchy in VisualWorks. The hierarchy looks like this (using the VisualWorks convention showing each class' instance variables within parentheses):

```
Object ()
    ImplementationFont ()
        CompositeFont (currentFont fonts ...)
        DeviceFont ()
            ...
        SyntheticFont (baseFont ...)
```

`ImplementationFont` is an Adapter that adapts fonts to a standard, object interface. `DeviceFont` actually adapts one of the a native fonts on the platform. `SyntheticFont` is a Decorator that adds effects that may not be available in the platform's fonts. `CompositeFont` combines together multiple platform fonts to form characters that my not otherwise be available in a single platform font, such as international characters. It makes the combination of fonts act like one font.

This is a textbook example of both Composite and Decorator. It has the Component class (`ImplementationFont`), the Leaf/ConcreteComponent (`DeviceFont`), Composite (`CompositeFont`), and Decorator (`SyntheticFont`).

FontDescriptionBundle and SPSortedLines

`FontDescriptionBundle` is a Composite class in VisualWorks. Its Leaf class is `FontDescription`. VisualWorks uses a `FontDescription` to search for a native font the current platform that matches the description. The `FontDescription` may actually be a `FontDescriptionBundle` that describes a range of fonts.

This example is difficult to recognize as being a Composite because the two classes are not implemented in the same hierarchy as the pattern recommends. Instead, `FontDescription` is a subclass of Object and `FontDescriptionBundle` is in the `Collection` hierarchy.

`SPSortedLines` is a Composite class in VisualWorks whose Leaf class is `SPFillLine`. `SPFillLine` is a line segment that has special behavior for determining if it intersects with another line segment. `SPSortedLines` is a collection of such lines. This is another example of the Composite pattern where the Composite and Leaf classes are not implemented in the same hierarchy.

Composite Numbers

Some `Number` classes are actually implemented using other `Number`s. They are composite numbers. Any `Number` class that is not a platform data type (e.g., integer, float, or double) is probably a composite number. These composite numbers are examples of the Composite pattern.

For example, the `Fraction` class in Smalltalk is a composite number. It contains two numbers, a numerator and a denominator, and uses them to compute its value. When dividing two numbers, storing the result as a `Fraction` instead of a `Float` avoids

round-off error (although it also hurts efficiency). Since a `Fraction` acts like a single `Number` but is composed of two separate `Number`s, it is a Composite.

Another example is a fixed decimal number class, such as `FixedPoint` in VisualWorks and `Decimal` in IBM Smalltalk. A fixed decimal is like a floating point number except that the fractional portion is stored as an integer to avoid round-off error. It is a composite number because it is composed from simpler numbers, usually `Integer`s.

Beck (1996) briefly discusses a pattern called Impostor, a more domain-specific example of a composite number. He uses the pattern to implement `MoneySum`, an object that will add together two `Money` objects with different currencies without converting them to a single currency. Rather than actually adding the `Money`s, `MoneySum` just stores them both, and presumably any others that might later be added in. Later, when the sum is actually needed, such as to display the value in a certain currency, `MoneySum` performs all of the conversions at once: it converts its contents to that currency and adds them together. This one set of conversions is usually more efficient than numerous intermediate conversions to a currency that may not even be needed. Because `MoneySum` and `Money` have the same interface, they can be used interchangeably. Thus `MoneySum` is a Composite `Money`.

Related Patterns

Composite, Decorator, and Chain of Responsibility

Composite and Decorator 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 Composite or Decorator can easily be applied.

A Composite communicates with its Components through a Chain of Responsibility. A series of nested Composites culminating with a Leaf form a chain. When a message is sent to the top Composite in such a chain, that Composite forwards the message to its children. When the message reaches a Leaf, it is ultimately handled there. The ConcreteHandlers in a Chain of Responsibility are not required to have the same interface, as long as each handler knows the interface of its successor. However, Composites do have the same interface, so Handlers in a chain of Composites forward the same message repeatedly.

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

Iterator

The Composite often uses the Iterator pattern to delegate a message to its children.