Other Considerations of the Composite Pattern
Consequences

+ Uniformity

• Treat components the same regardless of complexity & behavior

```cpp
Expression_Tree expr_tree = ...;
Visitor visitor = ...;

for (auto iter =
    expr_tree.begin(order);
    iter != expr_tree.end(order);
    ++iter)
(*iter).accept(visitor);
```

No syntactic distinction between leaf nodes or composite nodes (iterator variant)

Eliminate type tags & switch statements when combined with other patterns.
Consequences

- **Uniformity**

- **Extensibility**
  - New component subclasses work wherever existing ones do
Consequences

+ Uniformity
+ Extensibility
+ Parsimony

- Classes & interfaces only include fields & methods that they need

```cpp
class Component_Node {

    virtual int item() const {
        throw Invalid_Function_Call("method not implemented");
        return 0;
    }

    virtual Component_Node *right() {
        return nullptr;
    }

    virtual Component_Node *left() const {
        return nullptr;
    }

    ...

};
```
Consequences

+ Uniformity
+ Extensibility
+ Parsimony

- Classes & interfaces only include fields & methods that they need

```cpp
class Component_Node {

    virtual int item() const {
        throw Invalid_Function_Call
            ("method not implemented");
        return 0;
    }

    virtual Component_Node *right(){
        return nullptr;
    }

    virtual Component_Node *left() const {
        return nullptr;
    }

    ...
}
```
Consequences
+ Uniformity
+ Extensibility
+ Parsimony

- Classes & interfaces only include fields & methods that they need

```cpp
class Leaf_Node : public Component_Node {
    ...
    int mItem;

    int item() { return mItem; }
}
```

Stores the Leaf Node’s value

```cpp
class Composite_Unary_Node : public Component_Node {
    ...
    Component_Node *mRight;

    Component_Node *right() {
        return mRight;
    }
}
```
Consequences

+ Uniformity
+ Extensibility
+ Parsimony

- Classes & interfaces only include fields & methods that they need

```cpp
class Leaf_Node : public Component_Node {
    ... 

    int mItem;

    int item() { return mItem; }
}
```

```cpp
class Composite_Unary_Node : public Component_Node {
    ... 

    Component_Node *mRight;

    Component_Node *right() {
        return mRight;
    }
}
```
Reference to the right child.
Consequences
– *Perceived complexity*
  • May need what seems like a prohibitively large number of classes and/or objects

Algorithmic Decomposition

Pattern- & OO-Decomposition
Consequences

- **Perceived complexity**
  - May need what seems like a prohibitively large number of classes and/or objects.

**Algorithmic Decomposition**

Knowledge of patterns is essential to alleviate perceived complexity.
Consequences

- *Awkward designs*
  - May yield “bloated” interfaces for composites & leaves

```c
int item()
Component_Node *left()
Component_Node *right()
void accept(Visitor &visitor)
```

*item()* is unused in composite nodes.

*left()* & *right()* are unused in leaf nodes.

See [en.wikipedia.org/wiki/Interface_bloat](en.wikipedia.org/wiki/Interface_bloat)
Implementation considerations

- Do components know their parents?
- e.g., is there an explicit “parent” pointer/reference?
Implementation considerations

- Uniform interface for both leaves & composites?
- Trade-off between uniformity & parsimony

Leaf inherits methods that it doesn’t need.
Implementation considerations

- Don’t allocate child storage in component super class.

```c
typedef struct Tree_Node {
    enum { NUM, UNARY, BINARY } tag_;  
    short use_;  
    union {
        char op_[3]; int num_;  
    } o_;  
    union {
        struct Tree_Node *unary_;  
        struct { struct Tree_Node *l_,  
        *r_; } binary_;  
    } c_;  
} Tree_Node;
```

This was a big problem with the algorithmic decomposition.
**Implementation considerations**

- Who is responsible for deleting children?
- e.g., the parent or the child itself?

See rmdir vs. /bin/rm -rf at [www.linfo.org/rmdir.html](http://www.linfo.org/rmdir.html)
Known uses
- ET++ Vobjects
- InterViews Glyphs, Styles
- Unidraw Components, Macro_Commands
- Internal representations of MIME types
- Directory structures on UNIX & Windows
- java.awt.Container #add(Component)
- Naming Contexts in CORBA
Summary of the Composite Pattern

- The expression tree processing app uses the *Composite* pattern to enhance the uniformity & extensibility of its key internal data structure.

Adding new types of nodes (& new operations on nodes) is greatly simplified.