AndyPatterns

Extract Composite - Refactoring to a Pattern

Refactoring existing code so that it improves and conforms to a design pattern can be a good thing. Your code will be easier to understand since it follows a known 'pattern'. Of course you have to need the change - your code will probably be uncomfortable to read and be crying out for clarification.

This refactoring is about implementing a one to many relationship in your code more clearly by refactoring to the Composite design pattern.

Extract Compos

Refactoring to patterns...

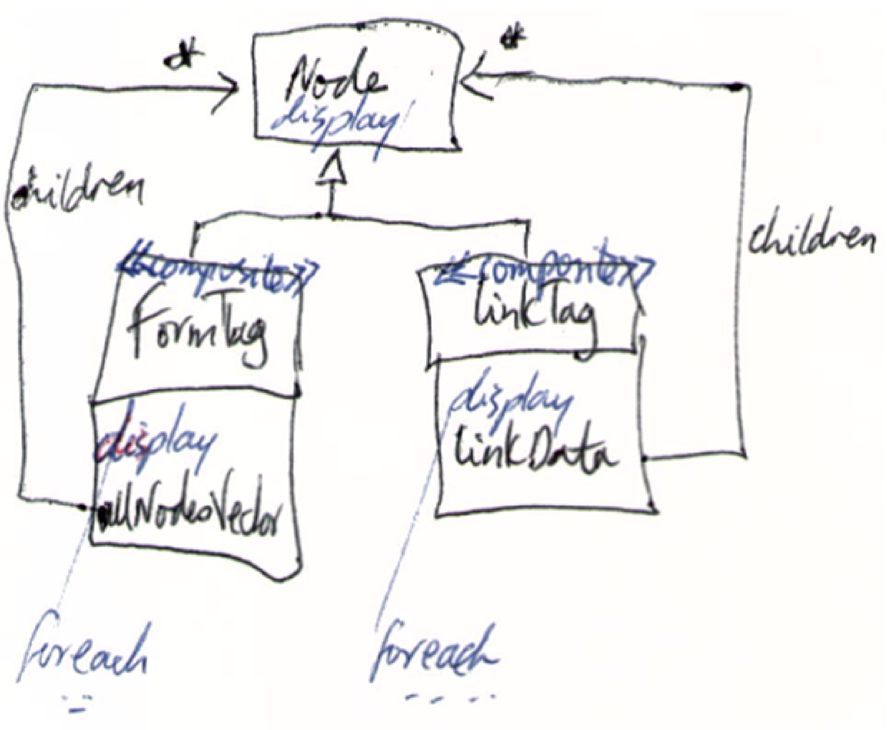
Andy Bulka
Technical Director
Austhink Software
www.austhink.com
March 2008

Extract Composite Talk Andy from **melbournepatterns**

UML - Before and After

Here is the situation before and after the refactoring, in hand drawn UML.

UML Before

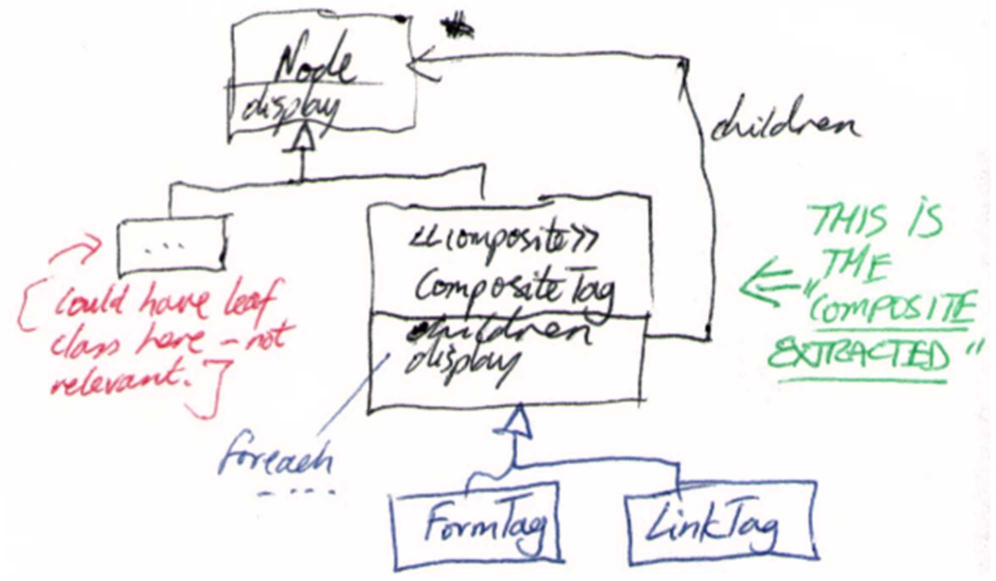


The **problem** with the "before" situation is:

- The child looping logic is duplicated twice - once in the FormTag class and again in the LinkTag class. And we all know that code duplication is evil and hard to maintain.

By refactoring, we remove the code duplication to a common base class.

UML After



The solution:

- We use the "extract composite" refactoring to put the common looping behaviour in the display() method of a common composite class.
- We make the existing classes subclasses of this new composite class.

Code - Before and After

Here is a python example of the above refactoring. Note that the display() method is actually called toPlainTextString() in this example.

Code Before

```
# Before

class Node(object):
 def __init__(self, name):
 self.name = name
 def toPlainTextString(self):
 return self.name

class FormTag(Node):
 def __init__(self):
 self.allNodesVector = []
 def toPlainTextString(self):
 result = ""
 for node in self.allNodesVector:
 result += node.toPlainTextString()
 return result

class LinkTag(Node):
 def __init__(self):
 self.linkData = []
 def toPlainTextString(self):
 result = ""
 for node in self.linkData:
 result += node.toPlainTextString()
 return result

f = FormTag()
f.allNodesVector.append(Node("a"))
f.allNodesVector.append(Node("b"))
f.allNodesVector.append(Node("c"))

l = LinkTag()
l.linkData += [Node("x"), Node("y"), Node("z")]

print f.toPlainTextString()
print l.toPlainTextString()
```

Then we apply the following steps...

- Step 1: Create a Composite Class - compile
- Step 2: Make each child container (a class in the hierarchy that contains duplicate child-handling code) a subclass of your composite - compile
- Step 3: For each method with duplicated looping code
  1. move & rename the child reference field UP to the composite using "Pull Up Field"
  2. Move the method UP to the composite using "Pull Up Method"
  3. Pull up any relevant constructor code too.
  4. Check interfaces so that client code using the old composites still works.

Code After

```
# After

class Node(object):
 def __init__(self, name):
 self.name = name
 def toPlainTextString(self):
 return self.name

class Composite(Node):
 def __init__(self):
 self.children = []
 def toPlainTextString(self):
 result = ""
 for node in self.children:
 result += node.toPlainTextString()
 return result

class FormTag(Composite):
 pass

class LinkTag(Composite):
 pass

f = FormTag()
f.children.append(Node("a"))
f.children.append(Node("b"))
f.children.append(Node("c"))

l = LinkTag()
l.children += [Node("x"), Node("y"), Node("z")]

print f.toPlainTextString()
print l.toPlainTextString()
```

Does the refactoring break anything?

A proper refactoring means you don't break anything and everything behaves as before.

```
>python -u "before.py"
abc
xyz
```

```
>python -u "after.py"
abc
xyz
```

Confirmed - we get the same output in both cases. :-)