Composite Pattern

CS342 Fall 2016

Problem

- I have given you the task of baking me some fresh, homemade cookies . But I want to know, not only how long each task in the process of baking cookies takes, but also how long the total process takes (this way in the future, I know how long I have to wait, and I know what task is taking so long.)
- I need to time each task, getDryIngredients(), mixIngredients(), bakeCookies(), etc..., but each one of those tasks is also made up of subtasks.
    - getDryIngredients() is made of getFlour(), getSugar(), etc...
- How can I ensure that I can keep track of the time each task takes?

Exercise

(Discussion only)

Come up with a basic design that will allow you to keep track of the time each step and substep in the process of baking cookies takes.

Possible Solutions

- Forget the details
  - Just time the whole thing and don’t worry about the individual steps
    - But what if you want to know how much time a subset of tasks takes?
- Create a chain of tasks
  - Wrap each call to a subtask in a timer. Don’t join subtasks into larger tasks.
    - As the length of the task grows, so does the complexity of this solution

Trees solve the problem

- Create a ‘tree’ of tasks that all share the same interface
  - Tree traversal, insertion, and deletion allows us to create subtasks that are joined to larger tasks
- Each task is a ‘node’ in the tree, that can either be a leaf task or a composite task of child nodes

Examples

- A corporation is made up of Divisions, Departments, Teams, and People. Each on of these shares deadlines, budgets, and work schedules.
- Files and Folders on your computer all share permissions, creation date, and size but have a hierarchical structure. Files are leaves and folders are composites.

Structure of the Composite Pattern

- A component base class that acts as an interface for the nodes
- A leaf class that defines functionality of childless classes
- A composite class that defines functionality of classes with children

Component Class

- The component class should be an interface
  - which means you do not need to implement it in Ruby, only document it

```
class Component
    def initialize()
        raise NoMethodError
    end
    def commonFunctionality()
        raise NoMethodError
    end
end
```

Leaf Class

- The leaf class implements the component interface
- It is the simplest block containing the basic functionality required by the component

```
class Leaf < Component
        def initialize()
            #initlaization code
        end
        def commonFunctionality()
            ...
        end
end
```

Composite Class

- The composite class also implements the component interface
- It is a higher level component that uses the results from child components to produce the results for its required functionality

```
class Composite < Component
          ...
          def commonFunctionality()
               @children.each{ |child|
                    accum += child.commonFunctionality()
               }
               accum
          end
     end
```

Composite Pattern

“When the sum acts like all of the parts”

- The composite pattern is useful when you are building a hierarchy and you do not want the design to be concerned with whether it is dealing with a node or leaf of the tree

Classwork

Shapes

```
def renderShapeToScreen()
        @children.each{|child|
                child.renderShapeToScreen()
        }
```

Using Operators

- Ruby allows operator overloading, which is useful for composite
  - Operator overloading allows us to refine the action for operators such as +, -, <<, etc.
- Overload the ‘<<’ operator to allow the user to easily add a task
  - def <<(task)

```
    @subtasks << task  # @subtasks is an array
end
```

Adding Arraylike syntax

We can override [] and []= to allow our Composite to act even more like an array

- ```
  def []=(index, new_value)
      @subtasks[index] = new_value
  end
  ```
- ```
  def [](index)
      @subtasks[index]
  end
  ```

Using Array-like syntax

- Now we use our composite class just like an array
  - `node[index] = new_task #if a leaf`
  - `node[index] << new_task #if a composite`
  - `node[index].getTime()`

When to use the Composite Pattern

- You have a hierarchy of classes
- All parts of the hierarchy have the same functionality
- You have an arbitrary number of subcomponents that can be added to the hierarchy

The Composite produces a Final Result

- The composite pattern should be used to produce a result. Each subtask performs an operation, not just stores a value.
  - This is not a Data Structure, though it uses a Data Structure
  - You are not using the composite to ‘store tasks’
- The node functionality should only return the accumulated result of the child functionality, nothing else.

To Leaf node or Not to Leaf Node...

- What's the difference between the leaf node class and composite node class?
  - add/remove children, access child methods
- Do you need a separate leaf and composite node class?
  - Why?
    - No: simplifies code, but has unnecessary functionality that is difficult to keep the user from calling
    - Yes: This is not a Data structure, which means a node will always be a leaf or a composite, so separating them makes sense
      - We don't need to arbitrarily add new nodes

Limitations of Composite

- It can be difficult to limit the tree to specific types in a Duck typed language
- The tree requirements can force you to overly generalize the component class