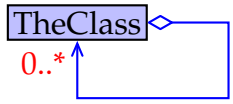
Worksheet 4: Object Relationships

Updated: 26th March, 2018

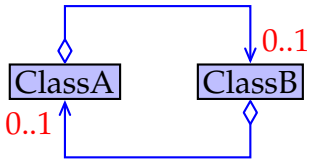
1. Discussion: Recursive Aggregation

Consider each of the following class structures, and explain what kind of *object* structures could result from it:

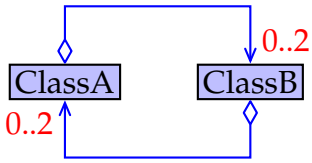
(a)



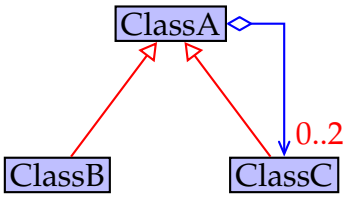
(b)



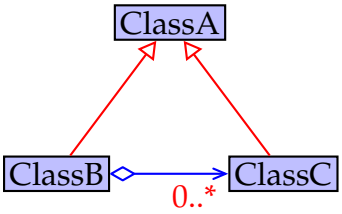
(c)



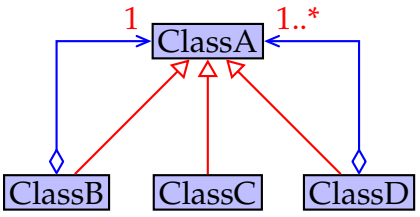
(d)



(e)



(f)



2. Discussion: Composite Operations

One of the lecture examples uses the Composite Pattern to represent files and directories, and shows how to find a given named file within the structure. However, it is somewhat limited.

(a) The recursive `find()` operation still takes $O(n)$ time (for n files and directories), because it has to individually check every entry.

  (i) What is a faster and easier way to arrange this?

  (ii) What is a slightly more sophisticated version of the `find()` method that could not be replaced so easily like this?

(b) What changes would you make so that the `find()` method returns *all* matching files (and not just the first one)?

(c) What changes would you make so that the `find()` method also finds matching *directories* (as well as files)?

3. Decorator Pattern for "Image Viewer"

Warning: The point of this worksheet is *not* just to "get the code working", but to adapt the Decorator and Composite patterns to the problems at hand.

The crucial part is the design. If you're just hacking at the code until it produces the right result, you're wasting your time!

Make a copy of your Image Viewer application.

Currently, images have a filename and a caption. In reality there is quite a lot of information ("metadata") one might wish to attach to an image, though only some is relevant for any given image. For our purposes, we'll consider the following metadata:

- Date (free-form text, to make things simple).
- GPS coordinates (for photos) – latitude, longitude and elevation.
- Rating (0–5 stars).

We'll modify the file format slightly:

```
filename1.jpg:Caption text:label=data:label=data
filename2.jpg:Caption text:label=data:label=data
filename3.jpg:Caption text:label=data:label=data
```

There is an updated "album.txt" file on Blackboard that conforms to this new specification.

Originally we just had the filename and caption text. Now we also have zero or more label=data sections, where the label is one of "date", "gps" or "rating". For dates, the data is simply free-form text (allowing for any date format). For gps, the data consists of three real numbers separated by spaces. For rating, the data is just one integer in the range 0–5.

First, sketch out a modified design in UML, based on the Decorator Pattern, for representing this information. Consider the existing `ImageRecord` class, and consider how its `getCaption()` method could fit in with the Decorator Pattern to allow all the other information to be displayed in the GUI.

Your design should *not* require that `ImageData`, `Album`, or `MainWindow` know anything about the extra information (though you may need some class name changes here and there). If you implement the Decorator Pattern properly, this should not be a problem.

Note: Decorator promotes extensibility (like many other patterns). You can add new decorations – new functionality – with minimal modifications to existing code.

When it comes to code, you will need to modify the `readAlbumFile()` function to parse the extra fields and create the Decorator objects.

4. Composite Pattern for "Product Viewer"

Obtain a copy of ProductViewer.zip from Blackboard.

This is a simple GUI application for showing a table of products, arranged in groups. At any given time, the UI shows:

- All products in the "current" product group;
- The total value of these products (i.e. price $\times$ number-in-stock, for each product, summed).

The user can change the current group using a menu (or strictly a "combo box" in GUI parlance). The user can also add new product groups, and also new products to the current group.

However, the intention is to have *hierarchical* product groups. What we would *like* to happen is as follows:

(a) In the model:

  (i) Product groups ought to be able to contain other product groups as well as products. There ought to be a single global product group at the top, which indirectly contains everything.

  (ii) The file format already caters for it, but the file reader needs to make use of the "parent" field for product groups.

  (iii) The product groups specified in "catalogue.txt" needs to be adjusted into a hierarchical form. It should be clear what the tree structure is supposed to look like given the existing group names.

  Currently each group's parent is given as "–". This will need to be changed.

(b) In the user interface:

  (i) The menu (combo box) should *only* show the current group and its sub-groups, whether direct or indirect. (Currently it shows all groups.)

  (ii) The main table should show all products within the current group *and* its sub-groups. In particular, if we're at the root of the tree, the table should display *every* product.

  (iii) The "Up" button should navigate to the group's parent, up one level in the tree. (Currently it does nothing.)

  (iv) The "Add Product Group" button should place the newly-created group *within* the current group, as a sub-group.

Using the Composite Pattern, design and implement a set of changes to the product viewer accordingly. It will probably help to draw the UML for the modified system before implementing it!

Note: As with most patterns in most situations, there are "wrinkles" that you need to address; e.g. you need a way to obtain the parent group for a given group.

5. Extra: Composite AddressBook

Note: This is not a core part of the worksheet, but is provided as extra practice if desired. Possibly you've had enough of the AddressBook application by now!

Apply the Composite Pattern to the AddressBook application, to allow for nested groups of entries. Rather than a single list of entries, we'll have a tree, containing `Entry` objects alongside `Group` objects. Groups can contain other other Entries and Groups.

The new `Group` class will resemble and essentially replace your existing `AddressBook` class. (Think about it – what purpose does the `AddressBook` class serve?)

It may help to draw the UML for the modified system before implementing it.

We'll also need a slightly more complicated file format. Originally we had this:

```
Peter Gibbons:pgibbons@initech.com:pete@gmail.com
Michael Bolton:mbolton@initech.com
Samir Nagheenanajar:snagheenanajar@initech.com:samir@hotmail.com
Milton Waddams:mwaddams@initech.com
Bill Lumbergh:blumbergh@initech.com:B.Lumbergh@curtin.edu.au.notreally
```

Now we'll have something a bit like this:

```
#group:Initech
  Peter Gibbons:pgibbons@initech.com:pete@gmail.com
  #group:Plebs
    Michael Bolton:mbolton@initech.com
    Samir Nagheenanajar:snagheenanajar@initech.com:samir@hotmail.com
  #endgroup
  #group:Bosses
    Bill Lumbergh:blumbergh@initech.com:B.Lumbergh@curtin.edu.au.notreally
  #endgroup
#endgroup
Milton Waddams:mwaddams@initech.com
```

The indentation is just for show (to illustrate the hierarchy) – you can pretend it's not really there. Importantly, there are now special "#group" and "#endgroup" indicators, which collect together groups of entries (and sub-groups within other groups).

To deal with this, we'll need to refactor the `readAddressBook` method. It's probably best to make it recursive, so that it calls itself when it encounters "#group", and returns when it encounters "#endgroup".

The other algorithmic change lies in the searching for names and email addresses. Currently, you (should) have a map (or two) to store Entry objects, and so retrieving an Entry just involves looking up the map.

Now you'll need to think about how to get the new `Group` class to perform a recursive search of its child nodes.

End of Worksheet