The Composite Pattern ¶

A "Structural Pattern" from the Gang of Four book

Verdict

> The Composite Pattern can bring symmetry not only to object hierarchies in Python, but even to hierarchies exposed by low-level system calls and high-level network applications. In Python, the Composite Pattern can often be implemented with less fuss than in tightly constrained object oriented languages. You won't be forced to inherit your container objects and the objects inside of them from a common parent class. Instead, you can build classes that share only a common interface rather than any implementation — or that are simply duck typed to offer common behavior.

The Composite Pattern suggests that whenever you design "container" objects that collect and organize what we'll call "content" objects, you will simplify many operations if you give container objects and content objects a shared set of methods and thereby support as many operations as possible without the caller having to care whether they have been passed an individual content object or an entire container.

This is such a general idea that we can begin by stepping back from Python and even from object-based programming, and looking at how the Composite Pattern works at the level of an operating system.

Example: the UNIX file system ¶

The admirably pithy two-letter UNIX command `ls`, according to its manual page, stands for "list directory" and when given the path to a directory will list the files and subdirectories inside.

```
$ ls /usr
3bnet/        bin/          lbin/         lost+found/   options/      spool/
adm/          gnu/          lib/          mail/         preserve/     src/
admin/        include/      local/        news/         pub/          tmp/
```

But the name "list directory" is misleadingly narrow, because `ls` is also happy to be given a single file's path as its argument!

```
$ ls /usr/bin/banner
/usr/bin/banner
```

Of course, users rarely invoke `ls` on a single file like this, without specifying any option; the only thing you learn is that the file exists (if it doesn't, `ls` will complain "No such file or directory"). Its ability to accept single file names makes more sense when combined with its "long listing" option.

```
$ ls -l /usr/bin/banner
-r-xr-xr-x   1 bin      bin        12620 Mar  3  1988 /usr/bin/banner
```

This symmetry — that `ls` operates without complaint on both individual files and also on the directories that contain them — seems so natural to most users that you might not notice the powerful design decision behind it. It would have been very easy for an operating system author to provide one command for listing a directory, and an entirely different command to show the attributes of one particular file. But the lack of symmetry would have carried a cost. While exploring the filesystem, you would then have had to constantly remember to switch from one command to the other depending on whether the object of your attention was a directory or file.

Worse yet, two separate commands would have provided no way to support wildcard operations that might match both files and directories. But, happily, `ls` has no problem with a heterogeneous argument list that has both individual files and also a directory inside:

```
$ ls -C /etc/l*
/etc/labelit    /etc/ldsysdump  /etc/led        /etc/link

/etc/log:
filesave.log
```

Here, `ls` has transparently detected that four of the names matched by the wildcard `/etc/l*` are files but that `/etc/log` is a directory with more files inside. (Well, with one file inside.) Had UNIX supplied separate commands for listing a file and listing a directory, this wildcard could not safely have been an argument to either command.

This, at the level of a command line interface, is the Composite Pattern. Operations like `ls` and `du` and `chmod` that make sense on both files and directories are implemented so that they run transparently on both. This not only lowers cognitive overhead, but makes shell scripts easier to write — there are many operations that a script can simply run blindly without needing to stop and check first whether a path names a file or a directory.

The art of using the Composite Pattern is determining where to break the symmetry. For example, the UNIX filesystem provides completely different commands `touch` for creating a new file and `mkdir` for creating a new directory. This could instead have been a single command with an option that flipped it from file-creating mode to directory-creating mode. But the designers thought that the two operations were conceptually different enough to deserve separate commands. The decision of how much symmetry to create will weigh upon each operation that a designer implements.

Forcing symmetry where there is really a difference can create awkward special cases. For example, two of the three permission bits in UNIX apply equally to directories and files: `r` gives read permission and `w` gives write permission. But the symmetry breaks at `x`, the third bit, which gives permission to "execute" a file but to "search" a directory by using it in a path. Should there have been two different `chmod` commands, one for files and one for directories? Or should the single `chmod` binary at least have used a different letter for that third bit, maybe `x` when applied to a file but `s` for "search" when applied to a directory? I myself think the designers of UNIX made the right decision here, because I find it easier to remember that `x` means something a little different for a directory than to remember a separate letter for directories, or a separate command. But the decision could have gone either way, and making decisions like these are where the designer applying the Composite Pattern needs finesse.

It should be noted that the symmetries that exist between files and directories on the command line are not exactly the same symmetries that exist down beneath `ls` and `chmod` at the level of system calls. At each level, the Composite Pattern was applied a bit differently. For system calls, some symmetry does exist: `stat()` and `chmod()` and `chown()` operate happily on both files and directories.

But `ls` is hiding the fact that if `stat()` reveals that a path names a directory, then `ls` needs to switch to a directory-specific system call to list the files inside. There is no symmetry between the UNIX system call for reading a normal file's content and the call for reading the list of files in a directory, and for a crucial reason: the two operations return different types of data. A file contains an unstructured stream of bytes; a directory, a series of distinct filenames. The question of return type will serve as a very important guardrail when you are designing in Python: if your desire to create symmetry between container and content leads you to engineer calls that require an `if` statement or `isinstance()` to safely handle their return value, then the desire for symmetry has led you astray.

On hierarchies ¶

As we now turn our attention to how the Composite Pattern looks in a programming language like Python, we should ponder a question that hangs above so much of the object-oriented and design-pattern literature from the 1990s:

Where have all the hierarchies gone?

The construction and manipulation of extensive hierarchies was both a frequent exercise for new programmers and a tedious labor for more experienced programmers. Hours were spent deciding how hierarchies would be constructed, what operations they would support, and how their destructors could be safely invoked. Hierarchies were everywhere.

And then they began to recede, like a tide that having run far up the sand begins to finally sweep out again.

- Popular languages in the late 1990s went wild for deeply nested package namespaces. To take a modern example from Go, the package `google.golang.org/appengine`, one must admit, comes with a hard guarantee that it won't conflict with the name of a package from another firm. The Zope 3 project, in its heyday, happily festooned the Python Package Index with multi-level package names like `zope.app.form` and `zope.app.i18n`. But today most Python packages opt for a simple non-compound name that jostles alongside the names of all other Python packages. And it almost never causes problems.
- The programming curricula of yore were rife with binary search trees, B+ trees, and tree balancing algorithms. But in real code, trees are very scarce. For every programmer who works on trees to, say, to write a persistent storage engine like BoltDB or Redis, a thousand programmers get to skip the exercise. Python programmers don't tend to use binary search trees; we use the even faster hash table (the Python "dictionary") whose structure, as it happens, is entirely flat — not a hierarchy.
- There was an era when hierarchy was inherent in the structure of databases. An employee record might hold a salary history right inside of it. But while hierarchy continues to exist around the edges of data storage, most recently in the form of NoSQL and document databases, our workhorse data stores tend to be flat ones: the relational database, the CSV file, the Pandas dataframe.

Again and again our discipline seems to revert back, where we can, to tables and lists and arrays where hierarchies might have reigned instead. The principle has even been enshrined in the famous *Zen of Python:*

> "Flat is better than nested."

The big exception, the realm in which hierarchy does reign supreme today, is the document. Documents are almost universally processed and represented as a hierarchy of sections and paragraphs beneath which are spans of bold and italics and hyperlinks. But the whole point of their being documents is that we aren't always forced to build them in code using object and method calls. Instead, when we can, we parse them from a native representation that makes the hierarchy explicit and natural. The great monument to the Composite Pattern on today's web is not document construction — documents are usually delivered as HTML — but document manipulation, through the Document Object Model exposed for the use of JavaScript code.

I will leave for another time my discussion of how the Document Object Model delivered a hierarchy, so programmers invented jQuery because they preferred an array instead.

Let's now turn to what the Composite pattern looks like in code.

Example: GUI programming with Tkinter ¶

Let's imagine that we want to print to the screen the hierarchy of frames and buttons out of which we have built a graphical user interface (GUI) using Tkinter, which comes built in to Python.

It would have been easy enough for Tkinter's designers to have decided that only `Frame` containers needed `winfo_children()` methods to list their children — after all, simpler widgets like `Label` and `Button` aren't supposed to contain children, and could have omitted the method entirely. But that asymmetry would have forced an `if` statement into any routine that wanted to visit both frames and their children:

```
# If Frame objects alone had offered winfo_children()

if isinstance(widget, Frame):
    children = widget.winfo_children()
    ...
else:
    # carefully avoid calling winfo_children()
    ...
```

This pattern, when it can't be avoided, can at least be improved by dodging the `isinstance()` call and instead using `getattr()` with three arguments to safely examine whether the object has the necessary method. This decouples the code from the vexed question of whether any other Tkinter widgets besides `Frame`, either today or in the future, can also include children widgets inside:

```
# Improvement: check for methods, not classes
...
winfo_children = getattr(Frame, 'winfo_children', None)
if winfo_children is not None:
    children = winfo_children()
    ...
```

In either case, the difference between container widgets and normal widgets would have haunted every piece of code that wanted to perform general processing.

But the authors of Tk chose, happily, to implement the Composite Pattern. Instead of making `winfo_children()` a special method that only `Frame` widgets offer, they made it a general method that is available on *every single widget object!* You never need to check whether it is present. For containers, it returns their list of child widgets. For other widgets? It simply returns an empty list.

Your code can therefore fly forward and always assume the presence of the method. Here, so that you can see a working example, is a complete program that builds a simple Tkinter GUI that can print out the widget hierarchy to the terminal:

```
from tkinter import Tk, Frame, Button

# Our routine, that gets to treat all widgets the same.

def print_tree(widget, indent=0):
    """Print a hierarchy of Tk widgets in the terminal."""
    print('{:<{}} * {!r}'.format('', indent * 4, widget))
    for child in widget.winfo_children():
        print_tree(child, indent + 1)
```

```
# A small sample GUI application with several widgets.

root = Tk()
f = Frame(master=root)
f.pack()

tree_button = Button(f)
tree_button['text'] = 'Print widget tree'
tree_button['command'] = lambda: print_tree(f)
tree_button.pack({'side': 'left'})

quit_button = Button(f)
quit_button['text'] = 'Quit Tk application'
quit_button['command'] =  f.quit
quit_button.pack({'side': 'left'})

f.mainloop()
root.destroy()
```

The resulting printout looks like:

```
* <tkinter.Frame object .!frame>
    * <tkinter.Button object .!frame.!button>
    * <tkinter.Button object .!frame.!button2>
```

Thanks to the Composite Pattern symmetry between widgets, no `if` statement is necessary to handle whatever kind of widget is passed to `print_tree()`.

Note that there is controversy among Composite Pattern enthusiasts over whether all widgets should really act like containers — isn't it fraudulent, they ask, for a widget to implement `winfo_children()` if it's not going to let you add child widgets? What sense does it make for it to act like a halfway container that supports read operations ("list children") without the corresponding write operations ("add child")? The more restrictive option would be to avoid putting `winfo_children()` on all widgets and instead only making truly general operations like `winfo_rootx()` universal (general, because all widgets have an *x*-coordinate). I myself tend to enjoy interfaces more when there is as much symmetry as possible.

If you study the Tkinter library — which is perhaps the most classic object oriented module in the entire Python Standard Library — you will find several more instances where a method that could have been limited to a few widgets was instead made a common operation on them all for the sake of simplicity and for the convenience of all the code that uses them. This is the Composite Pattern.

Implementation: to inherit, or not? ¶

The benefits of the symmetry that the Composite Pattern creates between containers and their contents only accrue if the symmetry makes the objects interchangeable. But here, some statically typed languages impose an obstacle.

One problem is posed by the most limited of the static languages. In those languages, objects of two different classes are only interchangeable if they are subclasses of a single parent class that implements the methods they have in common — or, if one of the two classes inherits directly from the other.

In static languages that are a bit more powerful, the restriction is gentler. There is no strict need for a container and its contents to share an implementation. As long as both of them conform to an “interface” that declares exactly which methods they implement in common, the objects can be called symmetrically.

In Python, both of these restrictions evaporate! You are free to position your code anywhere along the spectrum of safety versus brevity that you prefer. You can go the classic route and have a common superclass:

```
class Widget(object):
    def children(self):
        return []

class Frame(Widget):
    def __init__(self, child_widgets):
        self.child_widgets = child_widgets

    def children(self):
        return self.child_widgets

class Label(Widget):
    def __init__(self, text):
        self.text = text
```

Or your objects can simply duck type the same interface, and you can rely on tests to help you maintain the symmetries between containers and contents. (Where, for very simple scripts, your “test” might simply be the fact that the code runs.)

```
class Frame(object):
    def __init__(self, child_widgets):
        self.child_widgets = child_widgets

    def children(self):
        return self.child_widgets

class Label(object):
    def __init__(self, text):
        self.text = text

    def children(self):
        return []
```

Or you can choose another point on the design spectrum between these two extremes. Python supports many approaches:

- The classic common superclass architecture, shown in the first example above.
- Making the superclass an abstract base class with the tools inside Standard Library’s `abc` module.
- Having the two classes share an interface, like those supported by the old `zope.interface` package.

- You could spin up a type checking library like MyPy and use annotations to ask for hard guarantees that all of the objects processed by your code — both container and contents — implement the runtime behaviors that your code requires.
- You could duck type, and ask for neither permission or forgiveness!

Because Python offers this whole range of approaches, I choose not to define the Composite Pattern classically, where it's defined as one particular mechanism (a superclass) for creating or enforcing symmetry. Instead, I define it simply as the creation of symmetry — by whatever means — among the objects involved in concentric object hierarchies.