Chapter 10 - The Command Pattern

September 18, 2021

0.1 Overview

- we will try to implement an `undo` feature with the `Command` pattern
- the `Command` pattern helps us encapsulate an operation (undo, redo, copy, paste, etc) as an object
- what this simply means is that we can create a class that contains all the logic and the methods required to implement the operation

the advantage of doing this are as follows - we dont have to execute a command directly; it can be executed at will - the object that invokes the command is decoupled from the object that knows how to perform it; the invoker does not need to know any implementation details about the command - if it makes sense, multiple commands can be grouped to allow the invoker to execute them in order; this is useful, for instance, when implementing a multilevel undo command

0.2 Real-world examples

- we go to a restaurant for dinner, we give the order to the waiter
- the check that they use to write the order is an example of a command
- after writting the order, the waiter places it in the check queue that is executed by the cook
- each check is independent and can be used to execute many different commands, for example, one command for each item that will be cooked

0.3 Use cases

many developers use the undo example as the only case of the `Command` pattern; with the `Command` pattern however, you can do so much more

- **GUI buttons and menu items**: the `PyQt` example that was already mentioned uses the `Command` pattern to implement actions on buttons and menu items
- **other operations**: Apart form undo, commands can be used to implement any operation such as cut, copy, pste, redo and capitalize text
- **Transactional behavior and logging**: Transcational behavior and logging are important to keep a persistent log of changes
- **Macros**: By macros, we mean a sequence of actions that can be recorded and executed on demand at any point in time

0.4 Implementation

In this section, we will use the `Command` pattern to implement the most basic file utilities: - creating a file and optionally writing text (a string) to it - reading the contents of a file - renaming a file -

deleting a file

we are not going to implement these utilies from scratch, but just add an extra abstraction level on top of them so that they can be treaded as commands

- from the operations shown, renaming and creating a file support `undo`
- deleting a file and reading the contents of a file do not support undo
- `Undo` can actually be implemented on delete file operations
- one technique is to use a special trash/wastebasket directory that sotres all the deleted files, so that they can be restored when the user requests it
- this is the default behavior used on all modern desktop enviorments

Each command has two parts: - **The initialization part**: it is taken care of by the `__init__()` method and contains all the information required by the command to be able to do something useful (the path of a file, the contents that will be written to teh file, etc) - **The execution part**: it is taken care of by the `execute()` method; we call the `execute()` method when we want to actually run a command; this is not necessarily right after initializing it

[1]:
```
import os
verbose = True

class RenameFile:
    def __init__(self, src, dest):
        self.src = src
        self.dest = dest

    def execute(self):
        if verbose:
            print(f"[renaming 'self.src' to '{self.dest}']")
        os.rename(self.src, self.dest)

    def undo(self):
        if verbose:
            print(f"[renaming '{self.dest}' back to '{self.src}']")
        os.rename(self.dest, self.src)
```

Lets start with the rename utility - it is implemented using the `RenameFile` class - the `__init__()` method accepts the source (`src`) and destination (`dest`) file paths as parameters (strings) - if no path separators are used, the current directory is used to create the file - an example of using a path separator is passing the `/tmp/file1` string as `src` and the `/home/user/file2` string as `dest` - another example, where we would not use a path, is passing `file1` and `src` and `file2` an

we will add the `execute()` method to the class - the method does the actual renaming using `os.rename()` - the `verbose` veriable corresponds to a global **flag**, which, when activated gives feedback to the user about the operation that is performed - you can deactivate it if you perfer silent commands - note that although `print()` is good enough for an example, normally something more mature and powerful can be use used, for example the `logging` module

- our `rename` utility (`RenameFile`) supports the undo operation through its `undo()` method
- in this case, we use `os.rename()` again to revert the anme of the file to its original value

In this example, deleting a file is implemented in a function, instead of a class - this is to show it is not mandatory to create a new class for every command that you want to add - the `delete_file()` function accepts a file path as a string and uses `os.remove()` to delete it

[2]:
```
def delete_file(path):
    if verbose:
        print(f"deleting file {path}")
    os.remove(path)
```

The `CreateFile` class is used to create a file - the `__init__()` method for that class accepts the familiar `path` parameter and a `txt` parameter for the content (a string) that will be written to the file - if nothing is passed as `txt`, the default `hello world` text is written to the file - normally the sane default behavior is to create an empty file, but for the needs of this example, I decided to write a default string in it

[9]:
```
class CreateFile:
    def __init__(self, path, text='hello world\n'):
        self.path = path
        self.txt = txt

    def execute(self):
        if verbose:
            print(f"[creating file '{self.path}']")
        with open(self.path, mode='w', encoding='utf-8') as out_file:
            out_file.write(self.txt)

    def undo(self):
        delete_file(self.path)
```

- we then add an `execute()` method, in which we use the `with` statement and pythons `open()` built in function to open the file (`mode='w'`) and the `write()` function to write the `txt` string to it

- the undo for this operation of creating a file is delete that file
- so the `undo()` method, which we add to the class, simply uses the `delete_file()` function to achieve that, as follows

- the last utility gives us the ability to read the contents fo a file
- the `execute()` method of the `ReadFile` class uses `open()` again, this time in read mode and just prints the content of the file using `print()`

[8]:
```
class ReadFile:
    def __init__(self, path):
        self.path = path

    def execute(self):
        if verbose:
            print(f"[reading file '{self.path}']")
        with open(self.path, mode='r', encoding='utf-8') as in_file:
```

```
            print(in_file.read(), end='')
```

- the `main()` function makes uses of the utilities we have defined
- the `orgi_name` and `new_name` parameters are the orginal and new name of the file that is created and renamed
- a command list is used to add (and configure) all the commands that we want to execute at a later point
- note that the commands are not executed unless we explicitly call `execute()` for each command

- the next step is to ask the users if they want to undo the executed commands or not
- the user selects whether the commands will be undone or not
- if they choose to undo them, `undo()` is executed for all commands in the commands list
- howerver, since not all commands support undo, exception handling is used to catch (and ignore) the `AttributeError` exception generated when the `undo()` method is missing

[14]:
```
def main():
    orig_name, new_name = 'file1', 'file2'
    commands = (
        CreateFile(orig_name),
        ReadFile(orig_name),
        RenameFile(orig_name, new_name)
    )
    [c.execute() for c in commands]

    answer = input('reverse the executed command [y/n] ')
    if answer not in 'yY':
        print(f"the result is {new_name}")
        exit()
    for c in reversed(commands):
        try:
            c.undo()
        except AttributeError as e:
            print("Error", str(e))
```