Chapter 08 - Other Structural Patterns

September 18, 2021

0.1 Overview

- other common structural patterns are `flyweight`, `model-view-controller` (`MVC`), and `proxy`

flyweight: - the `flyweight` pattern teacher programmers how to minimize memory usage by sharing resources with simmilar objects as much as possible - memory issues can over the overhead of object creation - we can have issues where we need to create a very large number of objects (and possibly users) that need to coexist at the same time

MVC - the `MVC` pattern is useful mainly in application development and helps developers improve the maintainability of their applications by avoiding mixing the business logic with the user interface

proxy - in some applications, we want to execute one or more important actions before accessing an object, and this is where the `proxy` pattern comes in - an example is the accessing of sensitive information. Before allowing any user to access sensitive information, we want to make sure that the user has sufficient privilages - another case might be `Lazy initialization`; we want to delay the creation of a computationally expensive object untill the first time the user actually needs to use it - the idea of the `proxy` pattern is to help with performing such actions before accessing the actual object

0.2 The Flightweight Pattern

- the `flyweight` design pattern is a technique used to minimize memory usage and improve performance by introducing data sharing between objetcs
- a `flyweight` is a shared object that contains state-independent, immutable data
- the state-dependent, mutable dats should not be part of the `flyweight` because this is information that cannot be shared, since it differs per object
- if `flyweight` needs extrinsic data, it should be provided explicitly by the client code

- lets assume that we are creating a performance-critical game, a `first-person shooter`
- in `FPS` games, player/soldiers on the same team share some states, such as representation and behavior
- all soldiers have some common actions, such as jump, duck and so forth
- this means that a `flyweight` that will contain all of the common data
- not everything is shared such as `health`, `location`, `etc.`

0.2.1 Real-world Examples

- since this is an optimization design pattern, there is not an easy non-computing example of it
- we can think of `flyweight` as caching in real life

- many bookstores have dedicated shelves with the newest and most popular publications; this is similar to caching
- first, you can take a look at the dedicated shelves for the book you are looking for, and if you cannot find it, you can ask the bookseller to assist you

0.2.2 Use Cases

- flyweight is all about improving performance and memory usage
- all embeded systems and performance-critical applications (games, 3-D graphics), can benefit from it

Gang of Four's Requirements: - the application needs to use a large number of objects - there are so many objects that it's too expensive to store/render them. Once the mutable state is removed (because if it is required, it should be passed explicitly to flyweight by the client code), many groups of distince objects can be replaced by relatively few shared objects - objects identity is not important for the application. We cannot rely on object identity ebcause objects sharing causes identity comparisions to fail (objects that appear different to the client code end up having the same identy

0.2.3 Implementation

- we will create a small car park to illustrate the idea, making sure that the whole output is readable in a signle terminal page
- howerver, it is important to note that no matter how large you make the car park, the memory allocation stays the same

- `memoization` and the `flyweight` pattern may seem similar, but there is an important difference
- `memoization` is an optimization technique that uses a cache to avoid recomputing results that were already computed in an earlier execution step
- `memoization` does not focus on a specific programming paradigm such as `OOP`
- in Python, memoization can be applied to both methods and simple functions
- `flyweight` is an `OOP`-specific optimization design pattern that focuses on sharing object data

- first we will need an `Enum` parameter that describes the three different types of car that are in the car park

[12]:
```
from enum import Enum

CarType = Enum('CarType', 'subcompact compact suv')
```

- then we will define the class at the core of our implementation: `Car`
- the `pool` variable is the object pool (in other words, our cache)‘
- notice that `pool` is a class attribute (a variable shared by all instances
- using the `__new__()` special method, which is called before `__init__()`, we are converting the `Car` class to a metaclass that supports self-refrences
- this means that `cls` references the `Car` class
- when the client cod3e creates an instance of `Car`, they pass the type of the car as `car_type`
- the type of the car is used to check if a car of the same type has already been created
- if that's the case, the perviously created object is returned

- otherwise, the new car type is added to the pool and returned

- the `render()` method is what will be used ot render a car on the screen
- notice how all the mutable information not known by `flyweight` needs to be explicitly passed by the client code
- in this case, a random `color` and the coordinates of a location (of form `x, y`) are used for each car
- also note that to make `render()` more useful, it is necessary to ensure that no cars are rendered on top of each other

[15]:
```
class Car:

    pool = dict()

    def __new__(cls, car_type):
        obj = cls.pool.get(car_type, None)
        if not obj:
            obj = object.__new__(cls)
            cls.pool[car_type] = obj
            obj.car_type = car_type
        return obj

    def render(self, color, x, y):
        type = self.car_type
        msg = f'render a car of type {type} and  color {color} at ({x}, {y})'
        print(msg)
```

- the `main()` function shows how we can use the flyweight pattern
- the color of a car is a random value from a predefined list of colors
- the coordinates use random values between 1 and 100
- although `18` cars are rendered, memory is allocated only for `3`
- the last line of the output proves that when using `flyweight`, we cannot rely on object identity
- the `id()` function returns the a unique `ID` (or memory address of an object as an integer) for each object
- in our case, even if two objects appear to be different, they actually have the same identity if they belong to the same `flyweight family` (in this case, the family is defined by `car_type`
- Of course, different identity comparisons can be still used by objects of different families, but that is possible only if the client knows the implementation details

[17]:
```
import random


def main():
    rnd = random.Random()
    colors = 'white black silver gray red blue brown beige yellow green'.split()
    min_point, max_point = 0, 100
    car_counter = 0
```

```
    for _ in range(10):
        c1 = Car(CarType.subcompact)
        c1.render(random.choice(colors),
                        rnd.randint(min_point, max_point),
                        rnd.randint(min_point, max_point))
        car_counter += 1

    for _ in range(3):
        c2 = Car(CarType.compact)
        c2.render(random.choice(colors),
                    rnd.randint(min_point, max_point),
                    rnd.randint(min_point, max_point))
        car_counter += 1

    for _ in range(5):
        c3 = Car(CarType.suv)
        c3.render(random.choice(colors),
                    rnd.randint(min_point, max_point),
                    rnd.randint(min_point, max_point))
        car_counter += 1

    print(f'cars rendered: {car_counter}')
    print(f'cars actually created: {len(Car.pool)}')

    c4 = Car(CarType.subcompact)
    c5 = Car(CarType.subcompact)

    c6 = Car(CarType.suv)
    print(f'{id(c4)} == {id(c5)}? {id(c4) == id(c5)}')
    print(f'{id(c5)} == {id(c6)}? {id(c5) == id(c6)}')


if __name__ == '__main__':
    main()
```

```
render a car of type CarType.subcompact and  color black at (6, 12)
render a car of type CarType.subcompact and  color beige at (80, 14)
render a car of type CarType.subcompact and  color brown at (32, 0)
render a car of type CarType.subcompact and  color black at (42, 68)
render a car of type CarType.subcompact and  color brown at (96, 74)
render a car of type CarType.subcompact and  color black at (3, 90)
render a car of type CarType.subcompact and  color white at (2, 40)
render a car of type CarType.subcompact and  color silver at (52, 79)
render a car of type CarType.subcompact and  color black at (2, 25)
render a car of type CarType.subcompact and  color green at (40, 35)
render a car of type CarType.compact and  color beige at (8, 58)
```

```
render a car of type CarType.compact and  color silver at (58, 3)
render a car of type CarType.compact and  color silver at (1, 80)
render a car of type CarType.suv and  color yellow at (35, 0)
render a car of type CarType.suv and  color silver at (28, 98)
render a car of type CarType.suv and  color white at (20, 74)
render a car of type CarType.suv and  color gray at (24, 54)
render a car of type CarType.suv and  color blue at (30, 40)
cars rendered: 18
cars actually created: 3
2578857432448 == 2578857432448? True
2578857432448 == 2578858283264? False
```

0.3 Model View Controller Pattern

- one of the design principles related to software engineering is the `sepration of concerns` (`SoC`) principle
- the idea behind the `SoC` principle is to split an application into distinct sections, where each section is addresses a separate concern
- examples of such concerns are the layers used in a layered design (`data access layer`, `business logic layer`, `presentation layer`, etc)

- the `MVC` pattern is nothing more than the `SoC` principle applied to `OOP`
- the name of the pattern comes from the three main components used to split a software application
  - the `model`, `view` and the 'controller
- `MVC` is considered an architectural pattern rather than a design pattern
- the difference between an architectural and a design pattern is that the former has a broader score than the latter

- the `model` is the core component
  - it representing knowladge
  - it contains and manages the (business logic), data, state and rules of an application
- the `view` is a visual representation of the model
  - examples of views are a computer GUI, the text output of a computer terminal, etc
  - the view only displays the data; it doesn't handle it
- the `controller` is the link/glue between the model and view
- all communication between the model and the view happens through a controller

A typical use of an application that uses `MVC`, after the initial screen is rendered to the user is as follows: 1. the user triggers a view by clicking (typing, touching, and so on) a button 2. the view informs the controller of the user's actions 3. the controller processes user input and interacts with the model 4. the model performs all the necessary validation and state changes and informs the controller about what should be done 5. the controller instructs the view to upgrade and display the output appropraitely, following the instructions that are given by the model

0.3.1 Use Cases

- the separation between the view and model allows graphical designers to focus on the UI part and programmers to focus on development, without interfering with each other

- because of the loose coupling between the view and model, each part can be modified/extended without affectiong the other; for example, adding a new view is trivial; just implement a new controller for it
- maintaining each part is easier because the responsibilities are clear
- ideally, you want to create smart models, thin controllers and dumb views

A `model` is considered smart because it does the following - contains all the validation/business rules/logic - handles the state of the application - has access to application data (database, cloud, and so on) - does not depend on the `UI`

A `controller` is considered thin because it does the following - updates the model when the user interacts with the view - updates the view when the model changes - processes the data before delivering it to the model/view if necessary - does not display the data - does not access the application data directly - does not contain validation/business rules/logic

A `view` is considered dumb because it does the following - display the data - allows the user to interact with it - does only minimal processing, ususally provided by a template language - does not store any data - does not access the application data directly

0.3.2 Implementation

- we will implement a `MVC` from scratch, using a very simple example: a quote printer
- the user enters a number and sees the quote related to the number
- the quotes are stored in a quotes tuple
- this is the data that normally exists in a database, file, and so on, and only the model has direct access to it

[3]:
```
quotes = (
 'A man is not complete until he is married. Then he is finished.',
 'As I said before, I never repeat myself.',
 'Behind a successful man is an exhausted woman.',
 'Black holes really suck...',
 'Facts are stubborn things.'
)
```

the `model` is minimalistic; it only has a `get_quote()` method that returns the quote (string) of the quotes tuple based on its index `n` - note that `n` can be less than or equal to zero, due to the way indexing works in Python - improving this behavior is given as an excerise for you at the end of this section

[4]:
```
class QuoteModel:
    def get_quote(self, n):
        try:
            value = quotes[n]
        except IndexError as err:
            value = 'Not found!'
        return value
```

- the `view` has three methods
  - `show()`, which is used to print a quote (or the message `Not found`) on the screen

  - `error()`, which is used to print an error message on the screen
  - `select_quote()`, which reads the user's selection

[5]:
```
class QuoteTerminalView:
    def show(self, quote):
        print(f'And the quote is: "{quote}"')
    def error(self, msg):
        print(f'Error: {msg}')
    def select_quote(self):
        return input('Which quote number would like to see? ')
```

- the controller does the coordination
- the `__init__()` method initializes the model and view
- the `run()` method validates the quoted index given by the user, gets the quote from the model, and passes it back to the view to be displayed as shown

[7]:
```
class QuoteTerminalController:
    def __init__(self):
        self.model = QuoteModel()
        self.view = QuoteTerminalView()
    def run(self):
        valid_input = False
        while not valid_input:
            try:
                n = self.view.select_quote()
                n = int(n)
                valid_input = True
            except ValueError as err:
                self.view.error(f"Incorrect index '{n}'")
        quote = self.model.get_quote(n)
        self.view.show(quote)
```

[9]:
```
def main():
    controller = QuoteTerminalController()
    while True:
        controller.run()

if __name__ == '__main__':
    main()
```

```
Which quote number would like to see? 1
And the quote is: "As I said before, I never repeat myself."
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-9-24dec0390828> in <module>
      5
      6 if __name__ == '__main__':
----> 7     main()
```

```
<ipython-input-9-24dec0390828> in main()
      2     controller = QuoteTerminalController()
      3     while True:
----> 4         controller.run()
      5
      6 if __name__ == '__main__':

<ipython-input-7-22a1287faa3f> in run(self)
      7         while not valid_input:
      8             try:
----> 9                 n = self.view.select_quote()
     10                 n = int(n)
     11                 valid_input = True

<ipython-input-5-4f924dce09ed> in select_quote(self)
      5         print(f'Error: {msg}')
      6     def select_quote(self):
----> 7         return input('Which quote number would like to see? ')

c:
 ↪\users\vicktree\appdata\local\programs\python\python39\lib\site-packages\ipyk ernel\kernelba
 ↪py in raw_input(self, prompt)
    846                 "raw_input was called, but this frontend does not␣
 ↪support input requests."
    847             )
--> 848         return self._input_request(str(prompt),
    849             self._parent_ident,
    850             self._parent_header,

c:
 ↪\users\vicktree\appdata\local\programs\python\python39\lib\site-packages\ipyk ernel\kernelba
 ↪py in _input_request(self, prompt, ident, parent, password)
    890             except KeyboardInterrupt:
    891                 # re-raise KeyboardInterrupt, to truncate traceback
--> 892                 raise KeyboardInterrupt("Interrupted by user") from Non
    893             except Exception as e:
    894                 self.log.warning("Invalid Message:", exc_info=True)

KeyboardInterrupt: Interrupted by user
```

0.4 The Proxy Pattern

- the `proxy` design pattern gets its name from the proxy object used to perform an important action before accessing the actual object

Four different well-known `proxy` types: - a `remote proxy`, which acts as the local representation of

an object that really exists in a different address space (for example, a network server) - a `virtual proxy`, which uses lazy initialization to defer the creating of a computationally expensive object untill the moment it is actually needed - a `protection/protective proxy`, which controls access to a sensitive object - a `smart (reference) proxy`, which performs extra actions when an object is accessed; examples of such actions are reference counting and thread-safety checks

- we will be implementiong a virtual proxy
- first we create a `LazyProperty` class that can be used as a decorator
- when it decorates property, `LazyProperty` loads the property lazily (on the first use), instead of instantly
- the `__init__()` method creates two variables that are used as aliases to the method that initialize a property
- the `method` variable is an alias to the actual method, and the `method_name` variable is an alias to the methods name

- to get a better understanding of how the two aliases are used, print their value to the output (uncomment the two commented lines in the code)

[18]:
```
class LazyProperty:
    def __init__(self, method):
        self.method = method
        self.method_name = method.__name__
        print(f"function overriden: {self.method}")
        print(f"function's name: {self.method_name}")

    def __get__(self, obj, cls):
        if not obj:
            return None
        value = self.method(obj)
        print(f'value {value}')
        setattr(obj, self.method_name, value)
        return value
```

- the `LazyProperty` class is actually a descriptor
- descriptors are the reccomended mechanisms to use in Python to override the default behavior of its attributes access methods
  - `__get__()`, `__set__()`, `__delete__()`
- the `LazyProperty` class overrides only `__set__()` becuase that is the only accessed method it needs to override
- in other words, we don't have to override all access methods
- the `__get__()` method access the value of the property, the underlying method wants to assign, and use `setattr()` to do the assignment manually
- what `__get__()` actually does is very neat; it replaces the method with the value!
- this means that not only is the property lazily loaded, it can also be set only once

- the `Test` class shows how we can use the `LazyProperty` class
- there are three attributes: `x`, `y`, and `_resource`
- we want the `_resource` variable to be loaded lazily;
- thus we initialize it to `None`

[48]:
```
class Test:
    def __init__(self):
        self.x = 'foo'
        self.y = 'bar'
        self._resource = None

    @LazyProperty
    def resource(self):
        print(f'initializing self._resource which is: {self._resource}')
        self._resource = tuple(range(5)) # expensive
        return self._resource
```

```
function overriden: <function Test.resource at 0x0000028C891E8550>
function's name: resource
```

- the `resource()` method is decorated with the `LazyProperty` class
- for demonstration purposes, the `LazyProperty` class initializes the `_resource` attributes as a tuple
- normally this would be a slow/expensive initialization (database, graphics, and so on
- the `main()` function, as follows, shows how lazy initialization behaves

[49]:
```
def main():
    t = Test()
    print(t.x)
    print(t.y)
    # do more work
    print(t.resource)
    print(t.resource)
```

- notice how overriding the `__get__()` access method makes it possible to treat the `resource()` method as a simpel attribute
- we can use `t.resource` instead of `t.resource()`

In executation output of this example, we can see that: - the `_resource` variable is indeed initialized not by the time the `t` instane is created, but the first time that we use `t.resource - the second time`t.resource`is used, the variable is not initialized again - that's why the initialization string initializing`self._resource‘ is shown only once

[51]:
```
main()
```

```
foo
bar
initializing self._resource which is: None
value (0, 1, 2, 3, 4)
(0, 1, 2, 3, 4)
(0, 1, 2, 3, 4)
```

there are two baisc, different kinds of lazy initialization in `OOP` - at the `instance level`: this means that an object's property is initialized lazily, but the property has an object scope. Each instance (object) of the same class has its own (different) copy of the property - at the `class or module`

`level`: in this case, we do not want a different copy per instance, but all the instances share the same property, which is lazily initialized

0.4.1 Real-World Examples

- the `Chip` in your card is a good example of how protective proxy is used in real life
- the debit/card cintains a chip that first needs to be read by the `ATM` or card reaser
- after the chip is verified, a password (PIN) is required to complete the transaction
- this means that you cannot make any transactions without phyiscally presenting the card and knowing the PIN

- in software, the `weakref` module of Python contains a `proxy()` method that accepts an input object and returns a smart proxy to it
- Weak references are recommended way to add reference-counting support to an object

0.5 Use Cases

- it is used when creating a distributed system using either a private network or the cloud
- in a distributed system, some objects exist in the local memeory and some objects exist in the memory of remote computers. if we dont want the client code to be aware of such differences, we can create a remote proxy that hides/encapsulates them, making the distributed nature of the application transparent
- it is used when our application is suffering from performace issues due to the early creation of expensive objects; introducing lazy initialization using a virtual proxy to create the objects only at the moment they are actually required can give us significant performance improvements
- it is used to check if a user has sufficient privilages to access a piece of information; if our application handles sensitive information, we want to make sure that the user trying to access/modify it is allowed to do so. A protection/protective proxy can handle all security-related actions
- it is used when our application (or library, framework, etc) uses multiple threads and we want to move the burdan of thread safety from the client code to the application. In this case, we can create a smart proxy to hide the thread-safety complexities from client
- an `object-relational mapping` (`ORM`) API is also an example of how to use a remote proxy; `ORM` provides OOP-like access to a relational database. `ORM`'s act as a proxy to a relation database that can be actually located anywhere either at a local or remote server

0.5.1 Implementation

we will implement a simple protection proxy to view and add users; the service provides two options: - `viewing the list of users`: the operation does not require special privileges - `adding a new user`: this operation requires the client to provide a speical secret message

- the `SensitiverInfo` class contains the information that we want to protect
- the users variable is the list of exisiting users
- the `read()` method prints the list of the users
- the `add()` method adds a new user to the list

[52]:
```
class SensitiveInfo:
    def __init__(self):
        self.users = ['nick', 'tom', 'ben', 'mike']
    def read(self):
        nb = len(self.users)
        print(f"there are {nb} users: {' '.join(self.users)}")
    def add(self, user):
        self.users.append(user)
        print(f'Added user {user}')
```

- the `Info` class is a protection proxy of `SensitiveInfo`
- the `secret` variable is the message required to be known/provided by the client code to add a new user
- note that this is just an example; in reality, you should never store passwords like this
- the `Info` class, as we can see next, the `read()` method is a wrapper to `SensitiveInfo.read()` and `add()` method ensures that a new user can be adde

[54]:
```
class Info:
    '''protection proxy to SensitiveInfo'''
    def __init__(self):
        self.protected = SensitiveInfo()
        self.secret = '0xdeadbeef'
    def read(self):
        self.protected.read()
    def add(self, user):
        sec = input('what is the secret? ')
        self.protected.add(user) if sec == self.secret else print("That's wrong!
 ↪")
```

[55]:
```
def main():
    info = Info()
    while True:
        print('1. read list |==| 3. add user |==| 3.quit')
    if key == '1':
        info.read()
    elif key == '2':
        name = input('choose username: ')
        info.add(name)
    elif key == '3':
        exit()
    else:
        print(f'unknown option: {key}')
```