Chapter 03 - Other Creational Patterns

September 18, 2021

0.1 Overview

- other creational patterns are the `prototype` pattern and the `singleton` pattern

prototype: - the `prototype` pattern is useful when one needs to create objects based on an existing object using a `cloning` technique - the idea is to use a copy of that objects complete structure to produce the new object - this is almost natural in python because we have a `copy feature` that helps

singleton: - the `singleton` pattern offers a way to implement a class from which you can only create one object - some consider this an `anti-pattern` - its useful when you need to create one and only one object, for example, to store and maintain a global state for our program

0.2 Prototype Pattern

- assume that you want to create an application for storing, sharing and editing presentation and marketing content for products promoted by a group of salespeople
- this is simmilar to `network marketing`, where the individuals partner with a company to distribute products withing their social networks, using promotional tools
- `Bob` has a bunch of presentation material he show to customers
- `Alice` joins his team and uses his material
- one day, `Alice` realizes that they could get more customers if the video was ubtitled in french
- to help everyone, the system could allow the distributors with certian rank or trust levels, such as `Bob` to create independent copies of the orginal presentation video, as long as the new verison is validated by the `compliance team` of the `backing company`
- `Bob` makes the copy and gives it to `Alice` who can make changes

- the alternative is that each person had a refrence to a single object and one person making changes to that object change it for everyone

- the prototype design pattern helps us with creating object clones
- in the simplest version, this pattern is just a `clone()` function that accepts an object as an input parameter and returns a clone of it
- in python this can be done using `copy.deepcopy()`

0.2.1 Use Cases

- the prototype is useful when we have an existing object that needs to stay untouched, and we want to create an exact copy of it, allowing changes in some parts of the copy
- there is also the frequent need for duplicating an object that is populated from a database and has refrences to other database-based object

- it is costly (multiple queries to a database) to clone such a complex object, so prototype is a conventient way to solve the problem

0.2.2 Implementation

- when you have to manage multiple websites, there is a point where it becomes difficult to follow
- you need to access information quickly, such as `IP addresses` that are involved, domain names and their expriation dates, and maybe details about the DNS parameters
- so we need a kind of inventory tool
- lets imagine how those teams deal with this type of data daily activites, and touch on the implementation of a piece of software that helps consolidate and maintain the data
- at the heart of this system, we will have a `Website` class for holding all the useful information such as the name, the domain name, the description, the author of a website we are managing and so on
- in the `__init__()` method of the class, only some parameters are fixed: `name`, `domain`, `description` and `author`
- but we also want flexibility and client code can pass more parameters in th form of keywords using `kwargs` variable lenght collection
- note that there is a Python idiom to set an arbitrary attribute named `attr` with a value `val` on an object `obj` using hte `setattr()` built-in function: `setattr(obj, attr, val)`
- so we are using this technique for the optimal attributes of our class, at the end of the initialization method, this way:

```
for key in kwargs:
    setattr(self, key, kwargs[key])
```

[16]:

```
import copy

class Website:
    def __init__(self, name, domain, description, author, **kwargs):
        self.name = name
        self.domain = domain
        self.description = description
        self.author = author
        for key in kwargs:
            setattr(self, key, kwargs[key])

        def __str__(self):
            summary = [f'Website "{self.name}"\n',]
            infos = vars(self).items()
            ordered_infos = sorted(infos)

            for attr, val in ordered_infos:
                if attr == 'name':
                    continue
                summary.append(f'{attr}: {val}\n')
            return ''.join(summary)
```

- next the Prototype class implements the prototype design pattern
- the heart of the Prototype class is the clone() method which is incharge of cloning the objects using copy.deepcopy() function
- since cloning means we allow setting values for optional attributes, notice how we use the setattr() technique here with the attrs dictionary

- for more convenience, the Prototype class contains the register() and unregister() methods, which can be used to keep track of the cloned objects in a dictionary

[17]:

```
class Prototype:
    def __init__(self):
        self.objects = dict()

    def register(self, identifier, obj):
        self.objects[identifier] = obj

    def unregister(self, identifier):
        del self.objects[identifier]

    def clone(self, identifier, **attrs):
        found = self.objects.get(identifier)
        if not found:
            raise ValueError(f'Incorrect object identifier: {identifier}')

        obj = copy.deepcopy(found)
        for key in attrs:
            setattr(obj, key, attrs[key])
        return obj
```

- in the main() function, we can clone a first Website instance (site1), to get a second objects site2
- basically we instantiate the prototype class and we use its .clone() method

[27]:

```
def main():
    keywords = ('python', 'data', 'apis', 'automation')
    site1 = Website('ContentGardening',
                    domain='contentgardening.com',
                    description='Automation and data-driven apps',
                    author='Kamon Ayeva',
                    category='Blog',
                    keywords=keywords)


    prototype = Prototype()
    identifier = 'ka-cg-1'
    prototype.register(identifier, site1)
```

```
   site2 = prototype.clone(identifier,
                           name='ContentGardeningPlayground',
                           domain='play.contentgardening.com',
                           description='Experimentation for techniques␣
↪featured on the blog',
                           category='Membership site',
                           creation_date='2018-08-01')
```

- to end that function, we can use the `id()` function which returns the memeory address of an object, for comparing both objects addresses
- when we clone an object using a deep copy, the memory address of the clone must be different from the memeory addresses of the orginal object

```
for site in (site1, site2):
    print(site)

print(f'ID site1 : {id(site1)} != ID site2 : {id(site2)}')
```

Summary: 1. define the `Website` class, with its initialization method (`__init__()`) and its string representation method (`__str__()`) 2. define the `Prototype` class 3. have a main function that: - defines the `keywords` list we need - create the instance of the `Website` class, called `site` - create the `Prototype` object and we use its `register()` method to register `site` with its identifiers - we clone the `site1` object to get `site2`

> use the `id()` function to see if they have different memeory addresses

0.3 Singleton

- singleton pattern restricts the instantiation of a class to `one` object, which is useful when you need one object to coordinate actions for the system
- the basic idea is that only one instance of a particular class, doing a job, is created for the needs of the program
- to enusre this works, we need mechanism that prevent the instantiation of the class more than once and also prevent cloning

0.3.1 Real-world examples

- thing of a captain of a boat or a ship
- on the ship he is in charge and responsible for important decisions and a number of requests are directed to him because of this responsibility

- in software, the Plone CMS has, as its core, an implementation of the singleton
- there are several singleton objects available at the root of a Plone site, called `tools`, each in charge of providing a specific set of features for the site
- for example, the `Catalog tools` deals with content indexation and search features (built in search)
- the `Membership tool` which deals with things releated to user profiles
- each tool is global to the site, created from a specific `singleton` class and you cant create another instance of that `singleton` class in the context of the site

0.3.2 Use Cases

- is useful when you need to create only one object or need some sort of object capable of maintaining a global state for your program
- other possible use cases:
  - controlling concurrent access to a shared resource. For example, the class managing the connection to a database
  - a service or resource that is transversal in the sense that it can be accessed from different parts of the application or by different users and do its work; for example, the class at the core of the logging system or utility

0.3.3 Implementation

- lets implement a program to fetch content from web pages inspired by the tutorial from `Micheal Ford`
- we want to be able to track the list of web pages that were tracked, hence use of the singleton pattern: we need a signle object to maintain that global state

[31]:

```python
import urllib.parse
import urllib.request

class URLFetcher:
    def __init__(self):
        self.urls = []
    def fetch(self, url):
        req = urllib.request.Request(url)
        with urllib.request.urlopen(req) as response:
            if response.code == 200:
                this_page = response.read()
                print(the_page)
                urls = self.urls
                self.urls = urls
```

- to create a singleton, we need to make sure one can only create one instance of it
- to see if our class implements a singleton or not, we could use a trick which consistgs of comparing two instances, using the `is` oeprator

[32]:

```python
if __name__ == '__main__':
    f1 = URLFetcher()
    f2 = URLFetcher()
    print(f1 is f2)
    #print(URLFetcher() is URLFetcher())
```

```
False
```

- the reccomended technique is to use `metaclass`
- we first implement a metaclass for the singleton, meaning the class (or type) of the class that implements the singleton pattern, as follows

[33]:
```
class SingletonType(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(SingletonType, cls).__call__(*args,␣
 ↪**kwargs)
        return cls._instances[cls]
```

- now we can rewrite the URLFetcher class to use that metaclass
- we also add a dump_url_registry() method, which is useful to get the current list of URLs tracked

[34]:
```
class URLFetcher(metaclass=SingletonType):

    def fetch(self, url):
        req = urllib.request.Request(url)
        with urllib.request.urlopen(req) as response:
            if response.code == 200:
                the_page = response.read()
                print(the_page)
                urls = self.urls
                urls.append(url)
                self.urls = urls

    def dump_url_registry(self):
        return ', '.join(self.urls)

if __name__ == '__main__':
    print(URLFetcher() is URLFetcher())
```

```
True
```

[35]:
```
def main():
    MY_URLS = ['http://www.voidspace.org.uk',
            'http://google.com',
            'http://python.org',
            'https://www.python.org/error',
            ]

    print(URLFetcher() is URLFetcher())

    fetcher = URLFetcher()
    for url in MY_URLS:
        try:
            fetcher.fetch(url)
        except Exception as e:
            print(e)
    print('-------')
    done_urls = fetcher.dump_url_registry()
    print(f'Done URLs: {done_urls}')
```

Summary: 1. we define the `SingletonType` class, with its special `__call__()` method 2. we define `URLFetcher`, the class implementing the fetcher for the web page, initalizing it with the `urls` attribute; we add its `fetch()` and `dump_url_registry()` methods