Chapter 13 - Other Behavioral Patterns

September 18, 2021

0.1 Overview

the rest of the `Behavioral Patterns` are: - `Interpreter` - `Memento` - `Iterator` - `Template`

`Interpreter Pattern`: - the pattern is intersting for the advanced user of an application - the main idea behind this pattern is to give the ability to non-begginer `users` and domain experts to use a simple language, to get more productive in doing what they need to with the application

`Stragety Pattern` - the pattern promotes using multiple algorithms to solve a problem - for example, if you have two algorithms to solve a problem with some difference in performance depending on the input data, you can use stragety to decide which alogirhtm to use based on the input data at runtime

`Memento Pattern` - the memento pattern helps add support for `Undo` and/or `History` in an application - when implemented, for a given object, the user is able to restore a previous state that was created and kept for later possible use

`Iterator Pattern` - the pattern offers an efficient way to handle a container of objects and traverse to these memebers once at a time, using the famous `next` semantic - it really useful since, in programming, we use sequences and collections of objects alot, particularly in algorithms

`Template Pattern` - the pattern focus on eliminating code redundancy - the idea is that we should be able to redefine certain parts of an algorithm without changing its structure

0.2 Interpreter Pattern

- usually what we want to do is create a `domain-specific language` (`DSL`)
- a `DSL` is a computer language of limited expressiveness targeting a particular domain
- `DSLs` are used for different things, such as combact simulation, billing, visualization, configuration, communication protocols, and so on
- `DSLs` are devided into internal `DSLs` and external `DSLs`

`Internal DSLs` are built on top of a host programming language - an example is a language that solves linear equations using Python - the benefits is that we dont have to wory about creating, compiling and parsing grammer because the host language does that - the disadvantage is taht we are limited to the features of the host language

`External DSLs` do not depend on any host language - the creater of the DSL can decide al aspects of the language (grammer, sytnax, etc) but are also responsible for creating a parser and compiler

- the `interpreter` pattern is related only to internal `DSLs`
- the interpreter does not address parsing at all
- it assumes that we already have the parsed data in some convenient form

- this can be an `abstract syntax tree (AST)` or any other handy data structure

0.2.1 Real World Example

- a musician is an example of the `Interpreter` pattern
- musical notation represents the pitch and duration of a sound graphically
- the musician is able to reproduce a sound precisely based on its notation

0.3 Use Cases

- used when we want to offer a simple language to domain experts and advanced users to solve their problems
- the interpreter should only be used to implement simple languages, other wise use an `external DSL`
- the goal is to offer the right programming abstractions to the specialist (who are often not a programmers) so that they are more productive
- the focus is on offering a language that hides the peculiarities of the host language and offers a more human-readable syntax

0.4 Implementation

- we will create a `DSL` to control a smart home
- this fits into the `Internet of Things` era
- the user is able to control their home using a very simple event notation
- an event notation has the form of:
  - `command -> receiver -> argument`
- the `argument` part is optional
- not all events require arguments:
  - `open -> game`
- some events require arguments
  - `increase -> boiler temperature -> 3 degrees`
- the symbol `->` is used to mark the end of one part of an event and state the beginning of the next
- we will use regular expressions to parse this stuff
- the parsing part will be created using a tool called `Pyparsing`
- `pip install pyparsing`
- we will define simple grammer for our language
- we can define the grammer using the `Backus-Naur Form` (`BNF`) notation

```
event ::= command token receiver token arguments
command ::= word+
word ::= a collection of one or more alphanumeric characters
token ::= ->
receiver ::= word+
arguments ::= word+
```

- what the grammer tells us is that an event has the form of `command -> receiver -> argument` and that commands, receivers and arguments have the same form

  - the form is one or more alphanumeric characters
  - if you are woundering about the necessity of the numeric part, it is included to allow us to pass the arughments, such as degrees at the `increase -> boiler temperature -> 3 degrees` command

- the basic difference between the code and grammer definition is the code needs to be written in the bottom-up approach
- we cannot use a word without first assigning it a value
- `Suppress` is used to state that we want the `->` symbol to be skipped from the parsed results

- we will focus on just one placeholder class called the `Boiler` class
- a boiler has a default temperature of `83 Celsius`
- there are also two methods to increase and decrease the current tempature

[4]:
```
class Boiler:
    def __init__(self):
        self.temperature = 83 # in celsius
    def __str__(self):
        return f'boiler temperature: {self.temperature}'
    def increase_temperature(self, amount):
        print(f"increasing the boiler's temperature by {amount} degrees")
    def decrease_temperature(self, amount):
        print(f"decreasing the boilers tempature by {amount} degrees")
        self.temperature -= amount
```

- the next step is to add the grammar, which we already covered
- we will create a `boiler` instance and print the default behavior

[9]:
```
'''
from pyparsing import Word, OneOrMore, Optional, Group, Suppress,
alphanums'''


def grammer_to_code():
    word = Word(alphanums)
    command = Group(OneOrMore(word))
    token = Suppress("->")
    device = Group(OneOrMore(word))
    argument = Group(OneOrMore(word))
    event = command + token + device + Optional(token + argument)

    boiler = Boiler()
    print(boiler)

    # simplest way to retrieve the parsed output is using pyparsing
    print(event.parseString('increase -> boiler temperature -> 3 degrees'))

    cmd, vev, arg = event.parseString('increase -> boiler temperature -> 3␣
↪degrees')
```

```python
    cmd_str = ' '.join(cmd)
    dev_str = ' '.join(dev)

    if 'increase' in cmd_str and 'boiler' in dev_str:
        boiler.increase_temperature(int(arg[0]))
    print(boiler)
```

0.5 Stragety Pattern

- most problems can be solved in more than one way
- take, for example, the sorting proble, which is related to putting elements of a list in a specific order
- we can use the stragety pattern to pick out the best sorting algorithm

- the `Stragety` pattern promotes using multiple algorithms to solve a problem
- it killer feature is that it makes it possible to switch algorithms at runtime transparently (the client code is unaware of the change)
- so if you have two algorithms and you know that one works better with small input sizes, while the other works better with large input sizes, you can use `Strategy` to decide which algorithm to use based on the input data at runtime

0.5.1 Real-world Example

- reaching an airport to catch a flight
- if you want to save money, leave early by bus/train and not pay for parking or taxi

- in python you have `sorted()` and `list.sort()`
- one does it in plase while the other returns a new sorted array

0.5.2 Use Cases

- `Strategy` is a very generic design pattern with many use cases
- in general, whenever we want to be able to apply different algorithms dynamically and transparently, `Stragety` is the way to go
- by different algorithm, we mean different implementation of the same algorithm
- the **result** should be exactly the same, but each implementation has a different performance and code complexity

- this is not limited to sorting, it can be used to create all kinds of different resource filters (authentication, logging, data compression, encryption, etc)
- another use of the pattern is to create different formatting representations, either to achieve portability or dynamically change the representation of data

0.5.3 Implementation

- in languages where functions are not first-class citizens, each `Stragety` should be implemented in a different function
- Python treats functions as normal variables and this simplifies the implementation of the `Stragety` pattern

- assume that we are also asked to implement an algorithm to chewck if all characters in a string are unique
- for example, the algorithm should return `true` if we enter the `dream` string because none of the characters are repeated
- if we enter the `pizza` string, it should return `false` because the letter `z` exists two times
- the letters do not need to be consecutive

[10]:
```
def pairs(seq):
    n = len(seq)
    for i in range(n):
        yield seq[i], seq[(i + 1) % n]
```

- next we implement the `allUniqueSort()` function
- this accepts a string `s` and return `True` if all characters in the string are unique
- otherwise it returns `False`
- to demonstrate the `Stragety` pattern, we will make a simplification by assuming that this algorithm fails to scale
- we assume that it works fine for strings that are up to five characters
- for longer strings, we simulate a slowdown by inserting `sleep` statement

[ ]:
```
import time

SLOW = 3
LIMIT = 5
WARNING = 'too bad, you picked the slow algorithm :('

def allUniqueSort(s):
    if len(s) > LIMIT:
        print(WARNING)
        time.sleep(SLOW)
    strStr = sorted(s)
    for (c1, c2) in pairs(strStr):
        if c1 == c2:
            return False
    return True


def allUniqueSet(s):
    if len(s) < LIMIT:
        print(WARNING)
        time.sleep(SLOW)
    return True if len(set(s) == len(s)) else False
```

- unfortunately while `AllUniqueSet()` has no scaling problems, for some strange reason, this has worse performance than `allUniqueSort()` when checking slow strings
- we need to have our algorithm select one of the sorts

[13]:
```
def main():
    while True:
        word = None
        while not word:
            word = input('Insert word (type quit to exit)> ')
            if word == 'quit':
                print('bye')
                return
            strategy_picked = None
            strategies = { '1': allUniqueSet, '2': allUniqueSort }
            while strategy_picked not in strategies.keys():
                strategy_picked = input('Choose strategy: [1] Use a set [2]␣
 ↪Sort and pair ')
                try:
                    strategy = strategies[strategy_picked]
                    print(f'allUnique({word}): {allUnique(word, strategy)}')
                except KeyError as err:
                    print(f'Incorrect option: {strategy_picked}')
```

- normally the user should not be incharge of picking the stragety
- the point of the pattern is to make it possible to use different algorithms transparently

- if the person using this is not a user but a `developer` you want to encapsulate the two functions in a common class
- in this case, the other developer will just need to create an instance of `allUnique` and execute a single method, for instance, `test()`

0.6 Memento Pattern

- in many situations, we need a way to easily take a snapshot of the internal state of an object, so that we can restore the object with it
- `Memento` is a design pattern that can help us implement a solution for such situations

The `Memento` design pattern has three key components: - `Memento` a simple object that contains basic state storage and retrieval capabilities - `Originator` an object that gets and sets values of `Memento` instances - `Caretaker` an object that can store and retrieve all previously created `Memento` instances

- `Memento` shares many similarities with the `Command Pattern`

0.6.1 Real-world Examples

- an example can be found in the dictionary we use for a language, such as `English` or `French`
- the dictionary is often updated
- so sometimes, we might want to use an old version of a dictionary for research

- `Zope` with its integrated object database called `Zope Object Database` (`ZOBD`) offers a good examle of the `Memento` patterns
- it is famous for its object, Undo support, exposed *Through The Web* for context managers

- ZODB is an object database for Python and is in heavy use in the `Pyramid` and `Plone` communities

0.6.2 Use cases

- `Memento` is usually used when you want to provide some sort of `undo` and `redo` capabilities for your users
- another usage is the implementation of a `UI` dialog with `OK/Cancel` buttons, where we would store the state of the object on load, and let user decide if we restore the initial state or not

0.6.3 Implementation

- we will approach the `Memento` in a simplified way, and by doing things in a natural way for the Python language
- this means we do not necessarily need several classes
- one thing we will use is Pythons `pickle` module

- `pickle` module can transform a complex object into a byte stream and it can transform the byte stream into an object with the same internal structure

- lets thake a `Quote` class with the attributes `text` and `author`
- to create the memento, we will use a method on that class, `save_state()`, which as the name suggets will dump the state of the object, using the `pickle.dumps()` function
- `picke.dumps()` creates the `memento`

- that state can be restored later
- for that, we add the `restore_state()` method, making use of the `pickle.loads()` function

[15]:
```
class Quote:
    def __init__(self, text, author):
        self.text = text
        self.author = author

    def save_state(self):
        current_state = pickle.dumps(self.__dict__)
        return current_state

    def restore_state(self, memento):
        previous_state = pickle.loads(memento)
        self.__dict__.clear()
        self.__dict__.update(previous_state)

    def __str__(self):
        return f'{self.text} - By {self.author}'
```

[21]:
```
import pickle

def main():
```

```
    print('Quote 1')
    q1 = Quote("A room without books is like a body without a soul." 'Unknown␣
↪author')
    print(f'\nOriginal version:\n{q1}')
    q1_mem = q1.save_state()

    q1.author = 'Marcus Tullius Cicero'
    print(f'\nWe found the author, and did an updated:\n{q1}')

    q1.restore_state(q1_mem)
    print(f'\nWe had to restore the previous version:\n{q1}')
    print()
    print('Quote 2')
    q2 = Quote("To be you in a world that is constantly trying to make you be␣
↪something else is the greatest accomplishment.", 'Ralph Waldo Emerson')
    print(f'\nOriginal version:\n{q2}')
    q2_mem1 = q2.save_state()

    q2.text = "To be yourself in a world that is constantly trying to mak you␣
↪something else is the greatest accomplishment."
    print(f'\nWe fixed the text:\n{q2}')
    q2_mem2 = q2.save_state()
    q2.text = "To be yourself when the world is constantly trying to makeyou␣
↪something else is the greatest accomplishment."
    print(f'\nWe fixed the text again:\n{q2}')
    q2.restore_state(q2_mem2)
    print(f'\nWe had to restore the 2nd version, the correct one:\n{q2}')
```

0.7 Iterator Pattern

- in programming we use sequences or collections of objects alot, particularly in algorithms and when writing programs that manipulate date
- one can think of automation scripts, APIs, data-driven apps, and other domains
- this pattern is useful when we have to handle collection of objects

  Iterator is a design pattern in which an iterator is used to traverse a container and access the container's elements. The iterator patter decouples algoriths from containers; in some cases, algortihms are necessarily container-specific and thus cannot be decoupled

- `Iterator` pattern is extensively used in the Python context
- it is actually turned into a language feature

0.8 Real-World Examples

- when ever we have to go through things, such as a waiter in a restaurant asking peoples orders
- in Python we have iterable objects and iterators
- `Containers` or `sequences` types (*list*, *string*, *tuple*, *dictionary set*, etc.) are `iterable`, meaning we can iterate through them

- iteration is done automatically for you whenever you use the `for` or `while` loop to traverse through objects and access their members
- the magic `iter()` function helps us transform any object into an iterator

0.8.1 Use Cases

- it is tood to use this whenever you want one or several of the following behaviors
- make it easy to navigate through a collection
- get the next object in the collection at any point
- stop when you are done traversing through the collection

0.8.2 Implementation

- `iterator` is implemented in Python for us, within the `for` loops, list comprehensions, etc
- we can do our own implementation for special cases, using the Iterator protocol, meaning that our iterator object must implement two speical methods: `__iter__()` and `__next__()`
- an object is called `iterable` if we can get an iterator from it
- the `iter()` function (which calls `__iter__()` returns an iterator from them
- lets consider a footbal team we want to implement with the help of `FootbalTeam` class
- if we want to make an iterator out of it, we have to implement the iterator protocol, since it is not a built-in container type such as the *list* type
- basically, built-in `iter()` and `next()` functions would not work on it unless they are added to the implementation
- first we define the class of the iterator (`FootballTeamIterator`) that will be used to iterate through the football team object
- the `memebers` attribute allows us to initialize the iterator object with out container object (which will be a `FootballTeam` instance)
- we add a `__iter__()` method to it, which would return the object itself, and a `__next__()` method to return the next person from the team at each call untill we reach the last person
- these will allow looping over the memebers of the footbal team via the iterator

[22]:
```
class FootbalTeamIterator:
    def __init__(self, memebers):
        self.memebers = memebers
        self.index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.index < len(self.members):
            val = self.members[self.index]
            self.index += 1
            return val
        else:
            raise StopIteration()
```

```

```

- so for now the FootballTeam class itself
- the new thing is adding a __iter__() method to it that will initialize the iterator object that it needs (thus using FootballTeamIterator(self.members)) and return it

[23]:
```
class FootballTeam:
    def __init__(self, memebers):
        self.members = members
    def __iter__(self):
        return FootballTeamIterator(self.memebers)
```

- once we have a FootballTeam instance, we call the iter() function on it to create the iterator, and we loop though it using while loop

[24]:
```
def main():
    members = [f'player{str(x)}' for x in range(1, 23)]
    members = members + ['coach1', 'coach2', 'coach3']
    team = FootballTeam(members)
    team_it = iter(team)

    while True:
        print(next(team_it))
```

0.9 Template Pattern

- good code is one that avoids redundancy
- in OOP, methods and functions are important tools that we can use to avoid writing redundant code
- remember the sorted() examples we saw when discussing the Strategy pattern
- the sorted() function is generic enough that it can be used to sort more than one data structure (*lists*, *tuple*, etc.) using arbitrary keys

- the Template Pattern solves the problem of code redundancy
- the idea is that we should be able to redefine certain parts of an algorithm without changing its structure

0.9.1 Real-World Example

- daily routine for workers at Amazon warehouses is very close to the Template design pattern
- all workers follow more or less the same routine, but specifc parts of the routine are very different

- in python the cmd module uses the Template pattern
- cmd builds line-oriented command interpreters
- another example is asyncore, which is used to implement asynchronous socket serive client/servers

0.10 Use cases

- the idea is that if we see code reuse, we keep the `invarient` or common parts of the algorithm and abstract them out

- `Pagination` is a good use of the `Template` pattern
- pagination algorithms can be split into an abstract `invarient` part and a concrete `variant` part
- the invariant part takes care of things such as the maximum number of lines/pages
- the variant part contains functionality to show the header and footer of a specific page thjat is paginated

- all application frameworks make use of some form of the `Template` patrern

0.10.1 Implementation

- in this example we will implement a banner generator
- we want to send some text to a function and the function should generate a banner containing the text
- banners have some sort of style (dots or dashes)
- the banner generator has a default style, but we should be able to provide our own style

- the `generate_banner()` function is our `Template` function
- it accepts, as an input, the text (`msg`) that we want our banner to contain, and the style (`style`) that we want to use
- the `generate_banner()` function wraps the styled text with a simple header and footer

[25]:
```
def generate_banner(msg, style):
    print('-- start of banner --')
    print(style(msg))
    print('-- end of banner --nn')
```

- the `dots_style()` function simply capitalizes `msg` and prints `10` dots before and after it

[26]:
```
def dots_style(msg):
    msg = msg.capitalize()
    msg = '.' * 10 + msg + '.' * 10
    return msg
```

- another style that is supported by the generator is `admire_style()`
- this style shows the text in uppercase and puts an exclamation mark between each character of the text

[28]:
```
def admire_style(msg):
    msg = msg.upper()
    return '!'.join(msg)
```

- the next style is the `cow_style()`
- this style executes the `milk_random_cow()` method of `cowpy` which is used to generate a random `ASCII` art every time `cow_style()` is executed
- `pip install cowpy`

[31]:

```python
'''from cowpy import cow'''

def cow_style(msg):
    msg = cow.milk_random_cow(msg)
    return msg
```

[32]:

```python
def main():
    msg = 'happy coding'
    [generate_banner(msg, style) for style in (dots_style, admire_style,␣
 ↪cow_style)]
```