Chapter 07 - The Facade Pattern

September 18, 2021

0.1 Overview

- The `facade` pattern helps us to hide internal complexity of our systems and expose only what is neccessary to the client through a simplified inteface.
- `facade` is an abstraction layer implemented over an existing complex system
- on OOP this means that you can have several classes, but only one should be exposed to the client

0.2 Real-world examples

- when you call a bank, the customer service employee acts as a `facade` between you and the actual department
- a `key` to a car can be a `facade` to the complex process of starting a car

0.3 Use Cases

- most usual reason to use the `facade` pattern is for providing a single, simple entry point to a complex system
- by introducing the `facade`. the client code can use a system by simply calling a single method/function
- when you have more than one layer in your system, you can introduce one `facade` entry point per layer, and let all layers communicate with each other through their facades
- this promotes `loose coupling` and keeps the layers as independent as possible

0.4 Implementation

- assume we want to create an operating system using a multi-server approach, similar to how it is done in `MINIX 3` or `GNU Hurd`
- a multiserver operating system has a minimal kernal, called a `microkernel`, which runs in privileged mode
- all the other services of the system are following a server architecture (driver server, process server, etc)
- each server belongs to a different memory address space and runs on top of the microkernel in user mode

- the pros of this approach are that the operating system can become more fault-tolerant, reliable ans secure
- for example, since all drivers are running in user mode on a driver server, a bug in a driver cannot crash the whole system, nor can it affect the other servers

- the cons of the approachare the performance overhead and the complexity of system programming, because the communication between a server and the `microkernel`, as well as between the independent servers, happens using message passing
- message passing is more complex than the shared memeory model used in monolithic kernels such as `Linux`

- we begin with a `Server` interface. An `Enum` parameter describes the different possible states of a server
- we use the `ABC` module to forbid direct instantiation of the `Service` interface and make the fundamental `boot()` and `kill()` methods mandatory, assuming that different actions are needed to be taken for booting, killing, and restarting each server

ABC Module: - we need to subclass `ABCMeta` using the `metaclass` keyword - we use the `@abstractmethod` decorator for starting which methods should be implemented (mandatory) by all subclasses of servers

[6]:

```python
from abc import abstractmethod, ABCMeta
from enum import Enum

State = Enum('State', 'new running sleeping restart zombie')

class Server(metaclass=ABCMeta):
    def __init__(self):
        pass
    def __str__(self):
        return self.name

    @abstractmethod
    def boot(self):
        pass

    @abstractmethod
    def kill(self, restart=True):
        pass
```

- a modular operating system can have a great number of intersting servers: a file server, processing server, an authentication server, a network server, a graphical/window server, etc.
- the following example includes two stub servers: the `FileServer` and the `ProcessServer`
- Apart form the methods required to be implemented by the `Server` interface, each server can have its own specific method
- for instance, the `FileServer` has a `create_file()` method for creating files, and the `ProcessServer` has a `create_process()` method for creating processes

[8]:

```python
class FileServer(Server):
    def __init__(self):
        '''actions required for initializing the file server'''
        self.name = 'FileServer'
        self.state = State.new
    def boot(self):
        print(f'booting the {self}')
        '''actions required for booting the file server'''
        self.state = State.running

    def kill(self, restart=True):
        print(f'Killing {self}')
        '''actions required for killing the file server'''
        self.state = State.restart if restart else State.zombie

    def create_file(self, user, name, permissions):
        '''check validity of permissions, users rights, etc.'''
        print(f"trying to create the file '{name}' for user '{user}' with␣
 ↪permissions {permissions}")

class ProcessServer(Server):
    def __init__(self):
        '''actions required for initializing the process server'''
        self.name = 'ProcessServer'
        self.state = State.new

    def boot(self):
        print(f'booting the {self}')
        '''actions required for bootin'''
        self.state = State.running

    def kill(self, restart=True):
        print(f'Killing {self}')
        '''actions required for killing the process server'''
        self.state = State.restart if restart else State.zombie
    def create_process(self, user, name):
        '''check user rights, generate PID, etc.'''
        print(f"trying to create the process '{name}' for user '{user}'")
```

- the `OperatingSystem` class is a `facade`
- in the `__init__()`, all the necessary server instances are created
- the `start()` method used by the client code, is the entry point to the system
- more wrapper methods can be added, if necessary, as access points to the services of the servers, such as the wrappers, `create_files()` and `create_process()`
- from the clients point of view, all those services are provided by the `OperatingSystem` class
- the client should not be confused by unnecessary details such as the existence of server and the responsibility of each server

[10]:

```
class OperatingSystem:
    '''The Facade'''
    def __init__(self):
        self.fs = FileServer()
```

```python
        self.ps = ProcessServer()

    def start(self):
        [i.boot() for i in (self.fs, self.ps)]

    def create_file(self, user, name, permissions):
        return self.fs.create_file(user, name, permissions)

    def create_process(self, user, name):
        return self.ps.create_process(user, name)
```

[13]:

```python
class User:
    pass

class Process:
    pass

class File:
    pass


class WindowServer:
    '''dummy class'''
    pass


class NetworkServer:
    '''dummy class'''
    pass


def main():
    os = OperatingSystem()
    os.start()
    os.create_file('foo', 'hello', '-rw-r-r')
    os.create_process('bar', 'ls', '/tmp')

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

```
//output from code above

booting the FileServer
booting the ProcessServer
trying to create the file 'hello' for user 'foo' with permissions -rw-r-r
trying to create the process 'ls /tmp' for user 'bar'
```

0.5 Summary

- the `facade` pattern is ideal for providing a simple interface to client code that wants to use a complex system but does not need to be aware of the system' complexity