Chapter 04 - The Adapter Pattern

September 18, 2021

0.1 Overview

- this category of patterns is called *structual design patterns*
- a structural design pattern proposes a way of composing objects for creating new functionality
- the first of these patterns we will cover is the *adapter* pattern

- the *adapter* pattern is a structural design pattern that helps us make two incompatiable interfacer compatiable
- if we have an old component and we want to use it in a new system, or a new component that we want to use in an old system, the two can rarely communicate, without requiring any code changes
- we might not be able to make code changes because we may not have access to it (might be library)
- what we need to do is write an extra layer that makes all required modifications for enabling the communication between the two interfaces
- this layer is called an `adapter`

- in general, if you want to use an interface that expects `function_a()` but you only have `function_b()`, you can use an adapter to convert (adapt) `function_b()` to `function_a()`

0.2 Use Cases

- usually. one of the two incompatible interfaces is either foreign or old/legacy
- if the interface is foreign, it means that we have no access to the source code
- if it is old, it is usually impractical to refactor it
- using an adapter for making things work after that have been implemented is a good approach because it does not require access to the source code of the foreign interface

0.3 Implementation

- we will implement a club's activities, mainly, the need to organize performance and events for the entertainment of its clients, by hiring talented artists
- at the core, we have a `Club` class that represents the club where hired artists perform some evenings
- the `organize_performance()` method is the main action that the club can perform

[1]:
```
class Club:
    def __init__(self, name):
        self.name = name
```

```
    def __str__(self):
        return f'the club {self.name}'

    def organize_event(self):
        return 'hires an artist to perform for the people'
```

- most of the time our club hires a DJ to perform, but our application addresses the need to organize a diversity of performances, by a musician or music band, by a dancer, a one-man or one-woman show

- we find an open source contributed library that brings us two intresting classes: `Musician` and `Dancer`
- in the `Musican` class, the main action is performed by the `play()` method
- in the `Dancer` class, it is performed by the `dance()` method
- in our example, to indicate that these two classes are external, we place them in a seprate module

[3]:
```
class Musician:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return f'the musician {self.name}'

    def play(self):
        return 'play music'


class Dancer:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return f'the dancer: {self.name}'

    def dance(self):
        return 'does a dance performanc '
```

- the client code, using these classes, only knows how to call `organize_performance()` method on the `Club` class
- it has no idea about the `play()` or `dance()`

- we create a generic `Adapter` class that allows us to adapt a number of objects with different interfaces, into one unified interface
- the `obj` argument of the `__init__()` method is the object we want to adapt and `adapted_methods` is a dictionary containing key/value pairs matching the method the client calls and the method that should be called

[ ]:
```
class Adapter:
    def __init-_(self, obj, adapter_methods):
        self.obj = obj
        self.__dict__.update(adapter_methods)

    def __str__(self):
        return str(self.obj)
```

- when dealing with the different instances of the classes, we have two choices:
  - the compatible object that belongs to the `Club` class needs no adaptation. We can treat it as it is
  - the incompatible objects need to be adapted first, using the `Adapter` class
- the result is that the client code can continue using the known `organize_performance()` method on all objects without the need to be aware of any interface differences between the used classes

[4]:
```
def main():
    objects = [club('Jazz Cafe'), Musician('Roy Ayers'), Dancer('Shane Sparks')]
    for obj in objects:
        if hasattr(obj, 'play') or hasattr(obj, 'dance'):
            if hasattr(obj, 'play'):
                adapted_methods = dict(organize_event=obj.play)
            elif hasattr(obj, 'dance'):
                adapted_methods = dict(organize_event=obj.dance)
            # refrencing the adapted object here
            obj = Adapter(obj, adapted_methods)
        print(f'{obj} {obj.organize_event()}')
```

- the adapter takes an instance of `Musican` or `Dancer` and makes a relationship between the old `organize event` and the new `play()` or `dance()` methods
- `organize event` becomes part of the object dictionary and we can do `Musician.organize_event` and it will work