Chapter 06 - The Bridge Pattern

September 18, 2021

0.1 Overview

- another structural pattern is the `bridge` pattern
- the bridge pattern and the adapter pattern are very simmilar
- while the `adapter` is used later to make unrelated classes work together, the `bridge` pattern is designed up-front to decouple implementation from its abstractions

0.2 Real-World Examples

- an example of the bridge pattern can be the digital economy (information products)
- nowadays, the information product or `infoproduct` is part of the resources one can find online for training, self-improvement, or one's ideas and business development
- purpose of an information product that you find on certain marketplaces, or the website of the provider, is to deliver information on a given topic in a way such that it is easy to access and consume
- the provided material can be a PDF document or ebook, an ebook series, a video, a video series, etc.
- in SWE, device drivers are often cited as an example of the bridge pattern, when the developers of an OS define the interface for device vendors to implement it

0.3 Use Cases

- the bridge pattern is a good idea when you want to share an implementation among multiple objects
- basically instead of implementing several specialized classes, defining all that is required withing each class, you can define the following components
  - an abstration that applies to the classes
  - a seprate interface for the different objects involved

0.3.1 Bridge pattern vs Abstract Factory pattern

- the bridge pattern is more for when both the class and what it does varies often
- the class itself can be considered as the implementation and the behavior of the class as the abstraction
- the abstract factory provides an interface for creating groups of related or dependent objects, without specifying their concrete classes or their implementation concerns

0.4 Implementation

- we will be building an application where the user is going to manage and deliver content after fetching it from diverse sources:
  - a web page (based on its URL)
  - a resource accessed on an FTP server
  - a file on local file system
  - a database server
- instead of implementing several content classes, each holding the methods responsible for getting the content pieces, assembling them, and showing them inside an application, we can define an abstraction for the `Resource Content` and a separate interface for the objects that are responsible for fetching the content
- we begin with the class for our `Resource Content` abstraction, called `ResourceContent`
- then we will need to define the interface for implementation classes that help fetch content, that is, the `ResourceContentFetcher` class
- the concept is called `the Implementor`
- the first trick we use here is that, via an attribute `_imp` on the `ResourceContent` class, we maintain a reference to the object which represents the `implementor`

[1]:
```
class ResourceContent:
    '''
    Define the abstraction's interface
    Maintain a reference to an object which represents the Implementor
    '''

    def __init__(self, imp):
        self._imp = imp

    def show_content(self, path):
        self._imp.fetch(patch)
```

- we define the equivalent of an interface in python using two features of the language, the `metaclass` feature (which helps define the `type of a type`, and **abstrace base classes (ABC)**

[3]:
```
import abc

class ResourceContentFetcher(metaclass=abc.ABCMeta):
    '''
    Define the interface for implementation classes that fetch
    '''

    @abc.abstractmethod
    def fetch(path):
        pass
```

- now we can add an `implementation` class to fetch content from a web page or resource

[4]:
```
class URLFetcher(ResourceContentFetcher):
    '''
    Implement the Implementor interface and define its concrete
    implementation
    '''
    def fetch(self, path):
        # path is an URL
        req = urllib.request.Request(path)
        with urllib.request.urlopen(req) as response:
            if response.code == 200:
                the_page = response.read()
                print(the_page)
```

- we can also add an `implementation` class to fetch content from afile on the local filesystem

[6]:
```
class LocalFileFetcher(ResourceContentFetcher):
    '''
    Implement the Implementor interface and define its concrete
    implementation
    '''
    def fetch(self, path):
        # path is the filepath to a text file
        with open(path) as f:
            print(f.read())
```

- based on our `main` function to show content using both `content fetchers` could look like the following

[7]:
```
def main():
    url_fetcher = URLFetcher()
    iface = ResourceContent(url_fetcher)
    iface.show_content('http://python.org')

    print('========================')
    localfs_fetcher = LocalFileFetcher()
    iface = ResourceContent(localfs_fetcher)
    iface.show_content('file.txt')
```

Summary: 1. define `ResourceContent` class for the interface of the `abstraction` 2. define the `ResourceContentFetcher` class for the `Implementator` 3. define two `implementation` classes - `URLFetcher` for fetching content from an URL - `LocalFileFetcher` for fetching content from the local filesystem - finally, we add the `man()` function as shown earlier, and the ususal trick to call it