Course Project EECS 3311

George Giannopoulos **ID: 213468442**

Course Director: Maleknaz Nayebi

Course Project: Doctoshotgun(Project#3)

Due Date: August 15th, 2021

Phase#: Phase 1

Student Name	**George Giannopoulos**
GitHub User ID	**Giannou3250**
Chosen Project (SUD)	**rbignon/doctoshotgun**
Link to the Forked repository	**https://github.com/Giannou3250/doctoshotgun**
Link to the First Pull Request (on your forked repository)	**https://github.com/Giannou3250/doctoshotgun/pull/1**
Link to the second Pull Request (on your forked repository)	**https://github.com/Giannou3250/doctoshotgun/pull/2**

Pattern#1: **Structural Design Pattern (Bridge Method)**

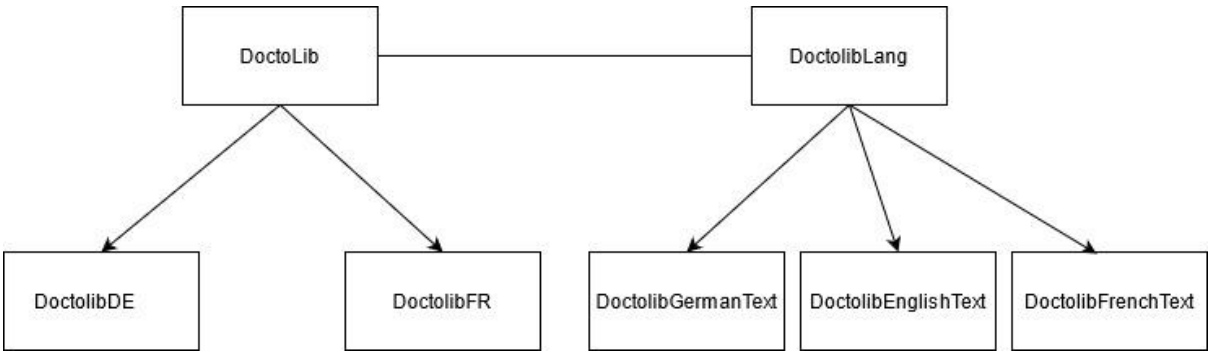
What is the bridge Pattern?

The bridge pattern is a structural design pattern used to separate related classes into different hierarchies. One Hierarchy is used for **abstraction**, the other is used for **Implementation.** These two Hierarchies will have their own tree structure. A parent (root) class as well as corresponding children (subclasses). Therefore these two class Hierarchies can be developed independently of one another even though they are related. The bridge pattern solves a problem that is known as the **cartesian product complexity explosion**. I will explain this problem through the explanation of my implementation below.

My Implementation

The implementation of my design pattern was applied to the **Doctolib** class along with its two subclasses, **DoctolibDE** and **DoctolibFR** (in the doctoshotgun.py file). These classes are responsible for accessing the doctolib application to book vaccination appointments for patients in their respective cities. **DoctolibDE** is respective to Germany, and **DoctolibFR** is respective to France. As we can see from this Hierarchy. **Doctolib** and its respective subclasses represent the **implementer** in my bridge pattern for this case. The **Abstraction** in this case is a set of classes I designed that directly relate to the implementer classes. The classes consist of: **DoctolibLang (root)**, **DoctolibFrenchText (subclass)**, **DoctolibGermanText (subclass)**, and **DoctolibEnglishText (subclass)**. This consists of a set of classes responsible for having a specific language designed for the implementer classes.

fig.1



In the figure above (fig.1) we can see a visual representation between these two hierarchies. The implementer and its subclasses act independently as a tree structure compared to the abstraction of doctolibLang and its subclasses. But they are bridged together by object instantiation. This will be explained further.

fig.2

```
rom1@money (master ●…) ~/src/doctoshotgun $ ./doctoshotgun.py fr paris romain@bignon.me -p 0
Password:
Starting to look for vaccine slots for Romain Bignon ...
This may take a few minutes/hours, be patient!

Center Centre de vaccination Covid-19 - CPAM de Paris 75:
- Centre de vaccination Amelot - CPAM de Paris 75... no availabilities
- Centre de vaccination Maroc - CPAM de Paris 75 ... no availabilities

Center Centre de vaccination Covid-19 - Paris 17e:
- Centre de Vaccination Covid-19 - Paris 17e... no availabilities

Center Centre de Vaccination Covid-19 - Ville de Paris:
- Centre de Vaccination - Mairie du 10e... found!
  ├ Best slot found: Mon May 17 16:30:00 2021
  ├ Second shot: Sat Jun 26 17:00:00 2021
  ├ Booking for Romain Bignon ...
  └ Booking status: True

🚀 Booked! Congratulations.
rom1@money (master ●…) ~/src/doctoshotgun $
```

The goal of the abstraction is to provide different languages to the user when the doctoshotgun program runs depending on their country. The diagram above (fig.2) provides a sample run of the program. As you can see there status text provided to the user showing current progress on certain events such as; “Starting to look for vaccine slots for Romain bignon”. This text is displayed in english. So in my implementation of the bridge pattern, the text could be represented in English by providing an instance of the class

DoctolibEnglishText to the implementer. But it could also provide text in French, or German, depending on whatever the implemer wants. This is where the bridge pattern becomes extremely useful.

The class structure is as follows below.

Abstraction:

class DoctolibLang:
{
Abstraction: Pass methods to subclasses
}

class DoctolibFrenchText():
{
Provide french text for implementer class
}

class DoctolibGermanText():
{
Provide german text for implementer class
}

class DoctolibEnglishText()
{
Provide english text for implementer class
}

Implementation:

Doctolib()
{
Instantiate Language object (german, english or french)
Pass methods to subclasses
}

DoctolibDE()
{
Use any language object
Implement functionality to program
}

DoctolibFR
{
Use any language object
Implement functionality to program
}

As seen from this pseudocode demonstration of the code example above, the language types are created separately from the doctolib functionality for the program. The client can instantiate any Language object type and use it for whatever doctolib subclass they want (**this is the bridging that is occurring between the two hierarchies**). Keeping the code structured this way, by separating the abstraction of language from the implementation of doctolib functionality is a lot more manageable and neat. It makes it easier to pick and choose what language a client programmer wants to use easily and effortlessly.

Why My implementation is useful

If the bridge pattern was not used here we would have to create doctolib subclasses for every country **and** language for the root class **Doctolib**. We would have a list of subclasses:

Class Doctolib()

Class DoctolibFREnglish()
Class DoctolibDEEnglish()
Class DoctolibFRFrench()
Class DoctolibDEFrench()
Class DoctolibFRGerman()

Class DoctolibDEGerman()

This is **3 x 2 = 6 subclasses**. This method is extremely slow and monotonous. Not to mention it is extremely messy and cumbersome to have this amount of subclasses and code piled in one area. Say if we wanted to add another country to our program that doctolib supports, such as **Canada**. We would then need to develop **three** more subclasses for full language support of Canada here (So that the user will have full freedom to use any language they desire if they want).

Class DoctolibCAEnglish
Class DoctolibCAFrench
Class DoctolibCAGerman

So the **cartesian product** complexity here is 3 x 3 = 9. The number of subclasses can grow at a very fast rate here depending on the amount of variations of language and countries we add. We would have to develop subclasses for them all where doctolib is the root.

With the bridge method, if we wanted to add Canada, all we simply have to do here is add **one** subclass to our **implementer** tree hierarchy under Doctolib. Such ***Class DoctolibCA.*** Then we instantiate an object from the abstraction tree of whatever language we want (French, English, German) and use it in our newly created subclass.

Clearly the bridge method is the better option here as we don't have to create a multitude of ever growing variations of classes. Whether we want to add a new language or a new country. we just create **one** subclass for this to its respective hierarchy.

References for this pattern:

#1: https://www.geeksforgeeks.org/bridge-method-python-design-patterns/

#2: https://refactoring.guru/design-patterns/bridge

#3: https://www.youtube.com/watch?v=F1YQ7YRjttI

#4 https://stackabuse.com/the-bridge-design-pattern-in-python

#5 Week11 Lecture notes

Pattern#2: **Creational Design Pattern (Singleton pattern)**

What is the Singleton Pattern?

The singleton design Pattern is a creational design pattern that allows a class/object to **instantiate one, and only one** instance of itself. This means that this object reference acts as a **global access point** to all operations in a program that will use this object reference. Therefore a singleton object instance should be responsible for creation, initialization, access and enforcement. Singleton objects are notably known to have their **constructors privatized** in order to **prevent access** to this constructor call from **outside clients**. They should also consist of a method which can be used to **reference the singleton instance**. This would typically be known as the **getInstance**() method.

The singleton pattern is a very popular creational design pattern that can be used and seen as helpful in very specific circumstances. The singleton

classes can be designed for circumstances such as logging, thread programming or caching. The singleton pattern can be a great creational pattern to use when you **only need one instance** of an object to perform a functionality. If you know that you don't need more than one object instance at a time. The singleton can be a viable option.

fig.3

```python
class SingletonMeta(type):
    """
    Singleton class implementation
    """

    _instances = {} #instances list

    def __call__(cls, *args, **kwargs): #this call function will be responsible for setti
        """
        Possible changes to the value of the `__init__` argument do not affect
        the returned instance.
        """
        if cls not in cls._instances:
            instance = super().__call__(*args, **kwargs)
            cls._instances[cls] = instance
        return cls._instances[cls]
```

fig.4

```python
class AvailabilitiesPage(JsonPage, metaclass=SingletonMeta):

    __shared_instance = 'Singleton'

    @staticmethod
    def getInstance():

        """Static Access Method"""
        if AvailabilitiesPage.__shared_instance == 'Singleton':
            AvailabilitiesPage()
        return AvailabilitiesPage.__shared_instance

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        """virtual private constructor"""
        if AvailabilitiesPage.__shared_instance != 'Singleton':
            raise Exception ("This class is a singleton class !") #exception raised if an objec
        else:
            AvailabilitiesPage.__shared_instance = self

    def find_best_slot(self, start_date=None, end_date=None):
        for a in self.doc['availabilities']:
            date = parse_date(a['date']).date()
            if start_date and date < start_date or end_date and date > end_date:
                continue
            if len(a['slots']) == 0:
                continue
            return a['slots'][-1]
```

My Implementation

The figures above (fig.3 and fig.4) depict my full implementation of the singleton class for the **AvailabilitiesPage** class object, This class is responsible for creating a **JSON type** object of the vaccine booking availability page on doctolib. The JSON objects in the doctoshotgun class are responsible for fetching relevant text data from different sections/parts of the doctolib application. Say for example a user is being booked in Paris for a vaccine appointment. Through the booking process of the doctoshotgun program. At some point information would need to be fetched from the availability page on Doctolib for vaccine bookings in paris. So in this case the availability page object would be instantiated in order to fetch that data to progress with the booking process. This is what the main functionality of **AvailabilitesPage** does.

Fig.5 AvailabilitiesPage with no singleton implementation

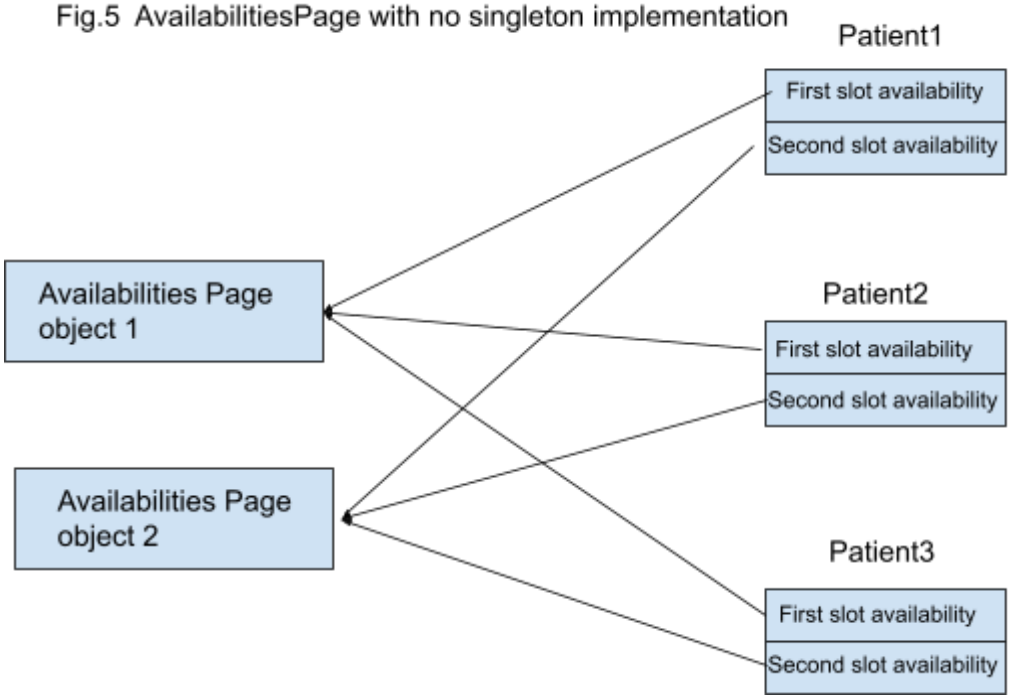
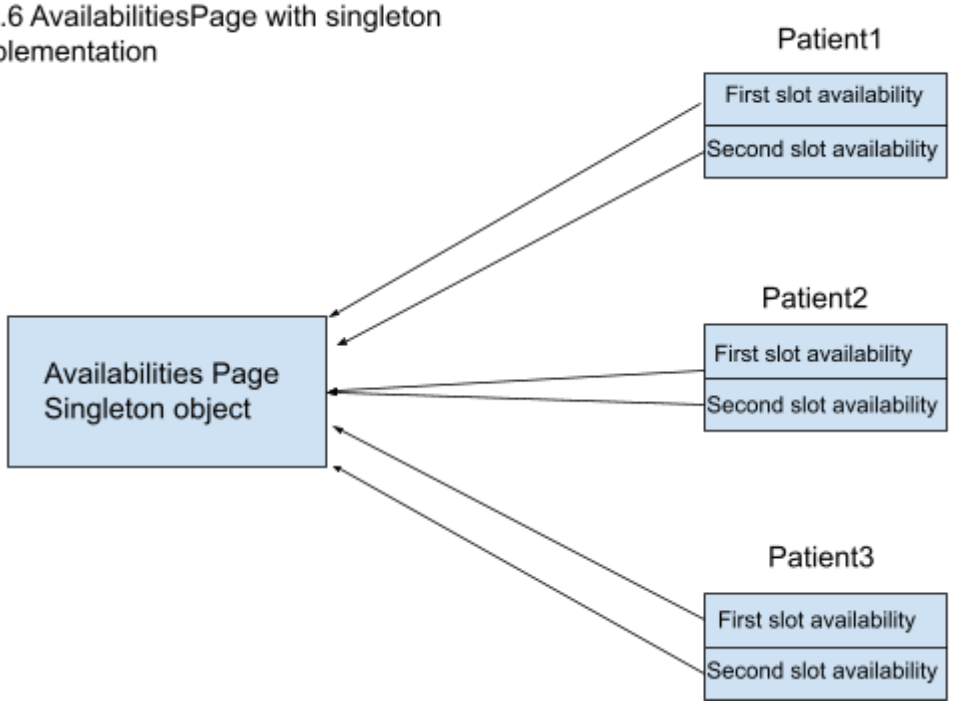


Fig.6 AvailabilitiesPage with singleton Implementation



More on my implementation and why it is useful

The reason why I chose to design a singleton pattern for this class object can be described by the diagram above. When a user runs the **doctoshotgun** program, they choose the amount of patients they need to book vaccination appointments for. If it is just one patient, the **AvailabilityPage** class **needs to get instantiated two times**. The first time is for **first shot availability**, and the second time is for **second shot availability**. If there is more than one patient. This repeats for each patient. The Diagram above (fig. 5) depicts this fact. Therefore the AvailabilitiesPage class object gets instantiated multiple times per run at least. However it is **not necessary** for there to be **more than one instance** of the **AvailabilitiesPage** object. One is enough to suffice. But the program, prior to my implementation, has been **creating two instances** of the **AvailabilitiesPage** class for each booking shot and each patient. This can be seen by looking at **lines 271-273** in my code implementation. We can see that multiple attributes are defined, **“availabilities”** and **“second_shot_availabilities”**. Both these attributes hold an instance of the **AvailabilitiesPage** class object. Multiple instances both get instantiated later on in the program within the **try_to_book()** function. It is important to note that

the reason why this program does not need multiple instantiations of the AvailabilitiesPage object is because once the AvailabilitiesPage is used to retrieve information using JSON for the first appointment, it becomes purposeless asset afterwards. Therefore instead of instantiating a new instance to fetch more data (which is what the program previously did) we should use this existing instance again, for the second shot availability.

fig.7

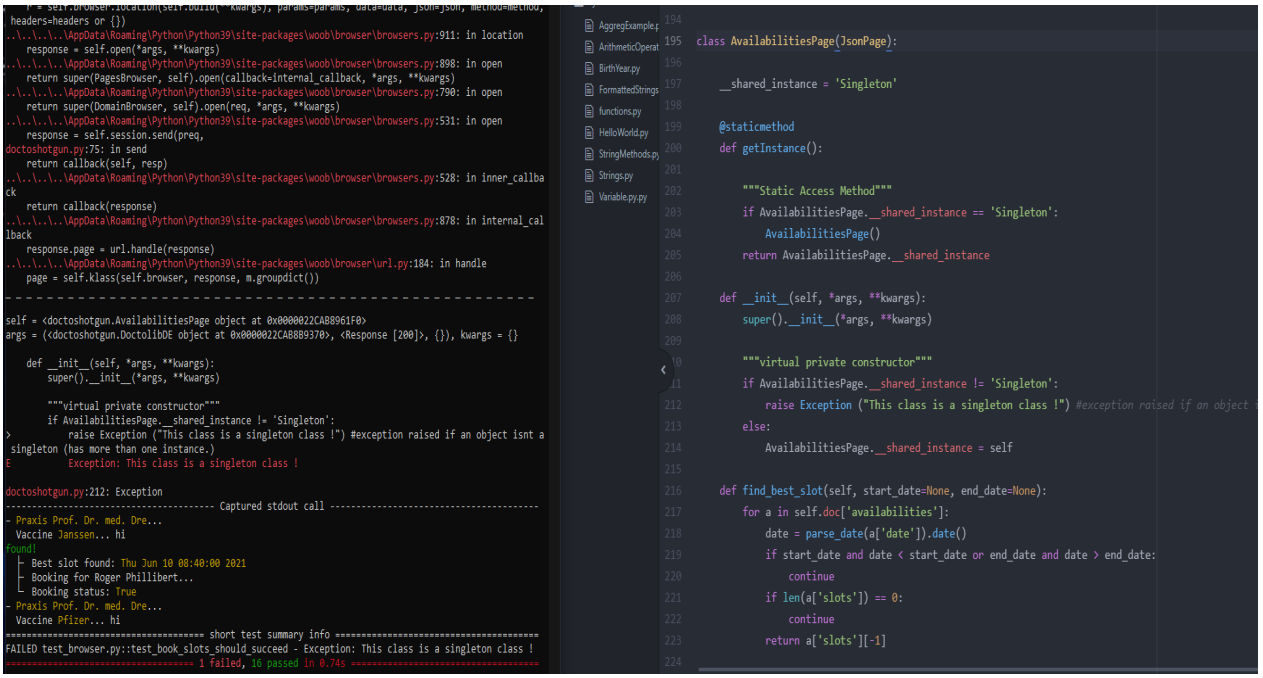
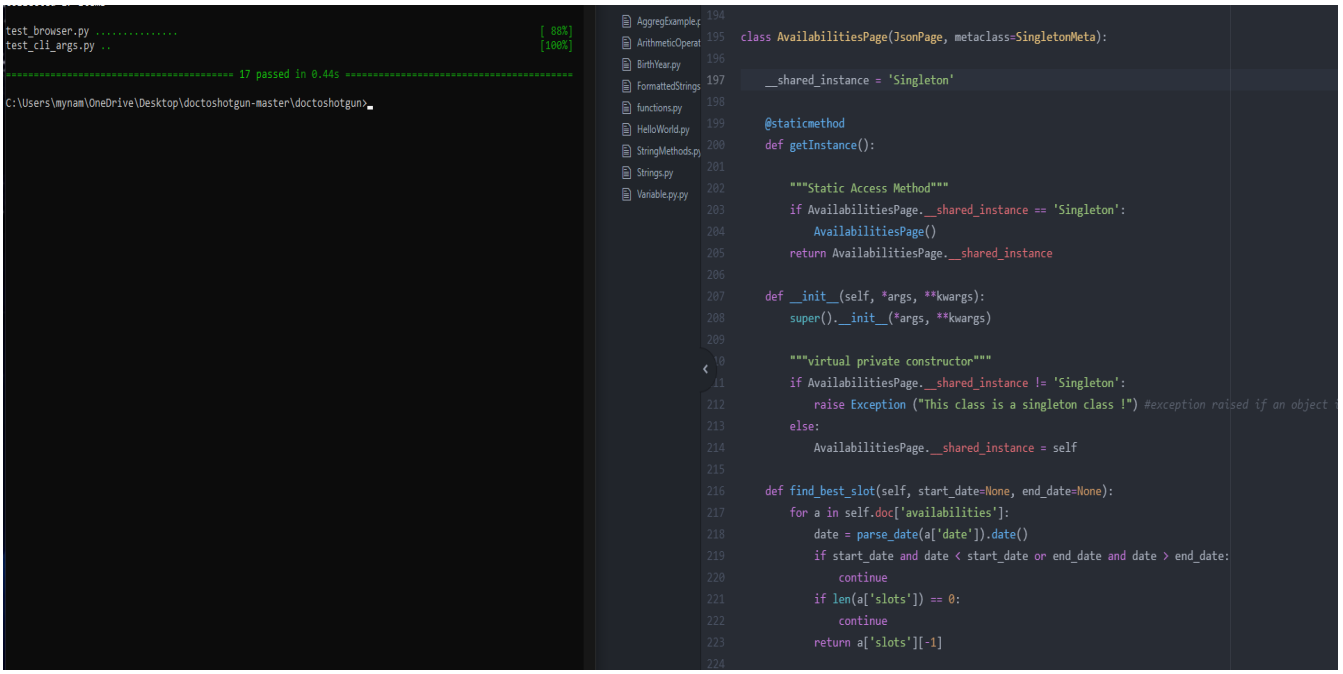


fig.8



In the above figures we can see how my implementation is useful. When the Availability Page class **inherits** the singleton class. We get no errors and the program runs fine. But it **does not** inherit the singleton class. We can see that we get an **exception** error thrown **“This class is a singleton class”**. This exception is a check I implemented in the **AvailabilitiesPage** class that will check whether multiple instances of this object are being created. If there are, then this exception will be thrown. If there are not, then no exception will be thrown. Since we got the exception that means that the program is creating two instances of the AvailabilitiesPage object. As I said earlier, one for the first booking, another for the second. So when **AvailabilitiesPage** class implements my singleton class by inheritance. This ensures that there will be only one instance of the object handling both cases (first booking availability and second), and that's why the program passes its test. Therefore the singleton works successfully for this program.

My implementation for the singleton consists of added features to the **AvailabilitiesPage** class. As seen from fig.5, these include a **virtual private constructor** added as well as a **getInstance()** method. The virtual private constructor is denoted as “virtual” because it bears the responsibility of a real private constructor by throwing an exception if multiple instances are created for the same object. Essentially performing the same functionality. The get instance method can be called to provide access to the singleton object instance. There is an exception check that will cause the program to fail if multiple instances of a singleton object are created.

The **Singleton class** is the other part of my implementation. This class gets passed to the **AvailabilitiesPage** object by **multiple inheritance** of **JSON and metaclass=singleton** (this can be seen in fig.5). The goal of the **singleton class** is to ensure that if a client attempts to create a new instance of a singleton object that is already created. Then the singleton object will become the newly created instance and its previous information will be overwritten.

I.e

Singleton1 = AvailabilitiesPage(X)
Singleton2 = AvailabilitiesPage(Y)

Singleton1 = singleton2 (same instance)

Singleton 1 and 2 will be the same instance and singleton1 information will be overwritten to singleton2. This is what the singleton class is responsible for. This ensures that at all times one instance will only be available. So with this information in mind, we can see that the singleton instance will handle **availabilitiesPage** functionalities for first shot availability and second shot availability for a patient (refer to lines 270-273, 433, 503 and fig.5 and fig.6). I used the Singleton design pattern for the **AvailabilitiesPage** class because having multiple instances of class objects is a waste of memory and completely unnecessary. It is a sloppy design to have instances created for one purpose only and then to be unnecessary afterwards. It was also viable to have **one instance** of the class to carry out **all functionalities** for its respective purpose. Hence it made perfect sense to use a singleton here. Also having a singleton for this design scenario prevents clients code from instantiation of more AvailabilityPage objects. More instances may lead to problems and having one instance is more manageable and better to understand universally if many other individuals are coding on the project. It is appropriate design philosophy that helps in this case.

References for this pattern:

#1: https://sourcemaking.com/design_patterns/singleton

#2: https://www.youtube.com/watch?v=hUE_j6q0LTQ

#3: https://www.geeksforgeeks.org/singleton-design-pattern/

#4 https://www.javatpoint.com/singleton-design-pattern-in-java

#5 Week10 Lecture notes