O Schedule e o Schedule manager estavam funcionais, entretando o Schedule não poderia se referir ao manager por uma questão de acoplamento, e é necessário que sempre que uma alteração seja feita no Schedule, o update seja chamado no Schadule manager. Para isso a saída escolhida foi utilizar o padrão Observer, que resolve esse problema. Aqui estão a classe de interface para o Observer e o Subject:

> obs: Em cada edição no codigo, os testes de schedule e schedule manager são rodados novamente para verificar que continum funcionando.

In [ ]:
```
"""
Module that contains the observer interface for the schedule.
"""
from abc import ABC, abstractmethod

class Subject(ABC): ...

class Observer(ABC):
    """
    Observer interface, used to implement the observer pattern in the schedule.
    solve the problem of circular dependencies between the schedule and the schedule management.
    provide a update method that is called when the schedule is updated.
    """

    @abstractmethod
    def update(self, subject: Subject) -> None:
        """
        Called when the schedule is updated.

        Arguments:
            schedule -- the schedule that was updated.
        """
        pass

class Subject(ABC):
    """
    Subject interface, used to implement the observer pattern in the schedule.
    solve the problem of circular dependencies between the schedule and the schedule management.
    provide a attach method that is called when the schedule is updated.
    """

    @abstractmethod
    def attach(self, observer: Observer) -> None:
        """
        Attach an observer to the subject.

        Arguments:
            observer -- the observer to attach.
        """
        pass

    @abstractmethod
    def detach(self, observer: Observer) -> None:
        """
        Detach an observer from the subject.

        Arguments:
            observer -- the observer to detach.
        """
        pass

    @abstractmethod
    def notify(self) -> None:
        """
        Notify all the observers that the subject has changed.
        """
        pass
```

Agora necessitamos que essa funcionalidade seja corretamente implementada nas classes, para isso vamos preparar um teste inicial:

In [ ]:
```
def test_subject_start_with_no_observers(self):
    permissions = {
        'user1': "read"
    }
    schedule = Schedule('schedule_id', 'title', 'description', permissions)

    initial_observers_len = len(schedule.__observers)
    self.assertEqual(initial_observers_len, 0)

def test_subject_add_observer(self):
    permissions = {
```

```
        'user1': "read"
    }
    schedule = Schedule('schedule_id', 'title', 'description', permissions)

    schedule.attach(self.observer)
        observers_len = len(schedule.__observers)
        self.assertEqual(observers_len, 1)
```

Fazendo o Schedule se comportar como Subject, e adicionando os metodos necessários para passar nos testes

```
from .schedule_observer import Observer, Subject

class Schedule(Subject):
    def __init__(self, schedule_id: str, title: str, description: str,
            permissions: dict, elements: [str] = None):
        [...]
        self.__observers = []

    [...]
    @property
    def observers(self):
        return self.__observers

    @elements.setter
    def elements(self, value):
        if isinstance(value, list) and all(isinstance(i, str) for i in value):
            self.__elements = value
            self.notify() # adicionado o notify
        else:
            raise ValueError("Elements must be a list of strings")

    [...]


    def attach(self, observer: Observer) -> None:
        '''
            Attach an observer to the subject.

            Arguments:
                observer -- the observer to attach.
        '''
        self.__observers.append(observer)

    def detach(self, observer: Observer) -> None:
        '''
            Detach an observer from the subject.

            Arguments:
                observer -- the observer to detach.
        '''
        self.__observers.remove(observer)

    def notify(self) -> None:
        '''
            Notify all the observers that the subject has changed.
        '''
        for observer in self.__observers:
            observer.update(self)
```

Continuando a adição de testes até que algum não passe:

In [ ]:
```
def test_subject_remove_observer(self):
    permissions = {
        'user1': "read"
    }
    schedule = Schedule('schedule_id', 'title', 'description', permissions)

    schedule.attach(self.observer)
    schedule.detach(self.observer)
    observers_len = len(schedule.observers)
    self.assertEqual(observers_len, 0)

def test_observer_gets_notified_on_elements_set(self):
    permissions = {
        'user1': "read"
    }
    schedule = Schedule('schedule_id', 'title', 'description', permissions)

    schedule.attach(self.observer)
    schedule.elements = ['element1', 'element2']
```

```
    # verificando se o observer chamou o método update
    self.assertEqual(len(self.observer.notifications), 1)

def test_observer_gets_notified_on_title_set(self):
    permissions = {
        'user1': "read"
    }
    schedule = Schedule('schedule_id', 'title', 'description', permissions)

    schedule.attach(self.observer)
    schedule.set_title('new title')
    # verificando se o observer chamou o método update
    self.assertEqual(len(self.observer.notifications), 1)
```

Está faltando o notify no set_title(), o que é facilmente solucionável:

```
def set_title(self, title: str) -> None:
        '''
            Sets the title of the schedule.

            Arguments:
                title -- title of the schedule.
        '''
        if title is None:
            raise ValueError("Title cannot be None")
        if not isinstance(title, str):
            raise TypeError("Title must be a string")
        if len(title.strip()) == 0:
            raise ValueError("Title cannot be empty")
        if len(title) > 50:
            raise ValueError("Title must have at most 50 characters")
        self.title = title
++        self.notify()
```

Testando para a edição de description:

In [ ]:
```
def test_observer_gets_notified_on_description_set(self):
        permissions = {
            'user1': "read"
        }
        schedule = Schedule('schedule_id', 'title', 'description', permissions)

        schedule.attach(self.observer)
        schedule.set_description('new description')
        # verificando se o observer chamou o método update
        self.assertEqual(len(self.observer.notifications), 1)
```

O teste não passa, mas a adição do notify() deve ser suficiente:

```
def set_description(self, description: str) -> None:
        '''
            Sets the description of the schedule.

            Arguments:
                description -- description of the schedule.
        '''
        if description is not None:
            if type(description) != str:
                raise TypeError("Description must be a string")
            elif len(description) > 500:
                raise ValueError("Description cannot have more than 500 characters")
        self.description = description
++      self.notify()
```

Agora precisamos garantir que o observer (user management) está reagindo corretamente a essas notificações, para isso vamos montar um teste para ver se ele está dando update no banco quando uma alteração acontece.

In [ ]:
```
def test_update_schedule_gets_called_when_schedule_is_updated(self):
    permissions = {
        'user1': "read"
    }
    schedule = Schedule('schedule_id', 'title', 'description', permissions)

    schedule.attach(self.schedule_management)
    schedule.set_description('Sample change')

    self.schedule_management.update_schedule.assert_called_once_with(schedule.id)
```

O teste não passa devido à não implementação do ScheduleManagement como Observer, vamos corrigir isso.

```python
from .schedule_observer import Observer, Subject

class ScheduleManagement(Observer):

    [...]

    def update(self, schedule: Schedule) -> None:
        """
        Called when the schedule is updated.

        Args:
            schedule: The schedule that was updated.
        """
        self.update_schedule(schedule.id)
```

Com isso o teste passa, isso indica que o comportamento do pattern Observer está funcional, e quando alterações forem feitas no Schedule, o update do management será corretamente chamado. Além disso todos os demais testes continuam passando, então o refactor feito não quebra o código original.