Histórico do Schedule

Notebook a ponto de registrar o avanço da classe Schedule. Não é para ser executado.

Properties

Testes a serem realizados:

In [ ]:
```python
def test_id_property(self):
    # Test the id property
    self.assertEqual(self.schedule.id, self.id)

def test_permissions_property(self):
    # Test the permissions property
    self.assertEqual(self.schedule.permissions, self.permissions)

def test_elements_property(self):
    # Test the elements property
    self.assertEqual(self.schedule.elements, self.elements)
```

Esses testes são muito simples e foram feitos apenas para desencargo de consciência. O código original foi suficiente para passar nos testes do enunciado:

In [ ]:
```python
@property
def id(self):
    return self.__id

@property
def permissions(self):
    return self.__permissions

@property
def elements(self):
    return self.__elements
```

Set Title

O seguinte teste inicial foi criado:

In [ ]:
```python
def test_set_title_valid(self):
    # Test setting a valid title
    valid_title = "Valid Title"
    self.schedule.set_title(valid_title)
    self.assertEqual(self.schedule.title, valid_title)
```

Este código foi o suficiente para passar no teste:

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

        Arguments:
            title -- title of the schedule.
    '''

    self.title = title
```

Em seguida, foram criados os seguintes testes, que impedem que o título seja vazio ou diferente de uma string:

In [ ]:
```python
def test_set_title_not_string(self):
    # Test setting a title that is not a string
    with self.assertRaises(TypeError):
        self.schedule.set_title(123)

def test_set_title_none(self):
    # Test setting a title that is None
    with self.assertRaises(ValueError):
        self.schedule.set_title(None)
```

O código não foi suficiente para passar nos testes, sendo necessário atualizá-lo:

In [ ]:
```python
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")
    self.title = title
```

Em seguida, foram criados os seguintes testes: um teste para impedir que o título fique muito grande (mais de 50 caracteres) e um teste na borda (50 caracteres):

In [ ]:
```
def test_set_title_too_long(self):
    # Test setting a title that is too long
    with self.assertRaises(ValueError):
        self.schedule.set_title("a" * 51)

def test_set_title_max_length(self):
    # Test setting a title that is exactly at the maximum length
    max_length_title = "a" * 50
    self.schedule.set_title(max_length_title)
    self.assertEqual(self.schedule.title, max_length_title)
```

O código foi suficiente para passar apenas no teste da borda, uma vez que não mesmo com a implementação de um limite de caracteres, a borda não é considerada. Contudo, para passar no teste de limite, foi necessário atualizar o código:

In [ ]:
```
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) > 50:
        raise ValueError("Title must have at most 50 characters")
    self.title = title
```

Em seguida, foram criados dois testes para impedir que o título seja uma string vazia ou contenha apenas espaços:

In [ ]:
```
def test_set_title_whitespace(self):
    # Test setting a title that contains only whitespace
    with self.assertRaises(ValueError):
        self.schedule.set_title("   ")

def test_set_title_empty(self):
    # Test setting an empty title
    with self.assertRaises(ValueError):
        self.schedule.set_title("")
```

O código não foi suficiente para passar nos testes, sendo necessário atualizá-lo. Para isso, utilizamos o método `strip()`, que remove os espaços em branco do início e do fim de uma string:

In [ ]:
```
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
```

Set Description

O seguinte teste inicial foi criado:

In [ ]:
```
def test_set_description_valid(self):
    # Test setting a valid description
    valid_description = "Valid Description"
    self.schedule.set_description(valid_description)
    self.assertEqual(self.schedule.description, valid_description)
```

Este código foi o suficiente para passar no teste:

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

        Arguments:
            description -- description of the schedule.
    '''
    self.description = description
```

Em seguida, foram criados os seguintes testes, que impedem que a descrição seja vazia ou diferente de uma string:

In [ ]:
```
def test_set_description_not_string(self):
    # Test setting a description that is not a string
    with self.assertRaises(TypeError):
        self.schedule.set_description(123)

def test_set_description_none(self):
    # Test setting a description that is None
    self.schedule.set_description(None)
    self.assertEqual(self.schedule.description, None)
```

O código não foi suficiente para passar nos testes, sendo necessário atualizá-lo:

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

        Arguments:
            description -- description of the schedule.
    '''
    if description is None:
        raise ValueError("Description cannot be None")
    if not isinstance(description, str):
        raise TypeError("Description must be a string")
    self.description = description
```

Em seguida, foram criados os seguintes testes: um teste para impedir que a descrição fique muito grande (mais de 500 caracteres) e um teste na borda (500 caracteres):

In [ ]:
```
def test_set_description_too_long(self):
    # Test setting a description that is too long
    with self.assertRaises(ValueError):
        self.schedule.set_description("a" * 501)

def test_set_description_max_length(self):
    # Test setting a description that is exactly at the maximum length
    max_length_description = "a" * 500
    self.schedule.set_description(max_length_description)
    self.assertEqual(self.schedule.description, max_length_description)
```

O código foi suficiente para passar apenas no teste da borda, uma vez que não mesmo com a implementação de um limite de caracteres, a borda não é considerada. Contudo, para passar no teste de limite, foi necessário atualizar o código:

In [ ]:
```
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
```

OBs.: Cometi um erro ao levantar um erro para o caso de a descrição ser vazia, pois isto deveria ser permitido. Por isso, o teste foi atualizado para que a descrição vazia seja permitida:

In [ ]:
```
def test_set_description_none(self):
```

```
    # Test setting a description that is None
    self.schedule.set_description(None)
    self.assertEqual(self.schedule.description, None)
```

To Dict

O seguinte teste inicial foi criado:

In [ ]:
```
def test_to_dict(self):
    # Call the to_dict method
    schedule_dict = self.schedule.to_dict()

    # Check that the dictionary has the correct keys and values
    self.assertEqual(schedule_dict, {
        "id": self.id,
        "title": self.title,
        "description": self.description,
        "permissions": self.permissions,
        "elements": self.elements
})
```

O código foi o suficiente para passar no teste:

In [ ]:
```
def to_dict(self) -> dict:
    '''
        Returns a dictionary representation of the schedule,
        which can be used to create a JSON object.

        Returns:
            dict -- Dictionary representation of the schedule.
    '''
    return {
        "id": self.__id,
        "title": self.title,
        "description": self.description,
        "permissions": self.__permissions,
        "elements": self.__elements
    }
```

Em seguida, foram feitos dois testes para observar o comportamento das listas permissions e elements do dicionário, a fim de ver uma conversão correta:

In [ ]:
```
def test_to_dict_permissions(self):
    # Test that to_dict correctly converts permissions
    schedule_dict = self.schedule.to_dict()
    self.assertEqual(schedule_dict["permissions"], self.permissions)

def test_to_dict_elements(self):
    # Test that to_dict correctly converts elements
    schedule_dict = self.schedule.to_dict()
    self.assertEqual(schedule_dict["elements"], self.elements)
```

Novamente, o código foi o suficiente para passar nos testes. Por fim, adicionamos um teste para verificar a conversão da entrada de atributos vazios (quando possíveis, isto é, diferente de `id` ou `title`):

In [ ]:
```
def test_to_dict_none_empty(self):
    # Test to_dict when attributes are None or empty
    empty_schedule = Schedule(self.id, self.title, None, [], [])
    schedule_dict = empty_schedule.to_dict()
    self.assertEqual(schedule_dict, {
        "id": self.id,
        "title": self.title,
        "description": None,
        "permissions": [],
        "elements": []
    })
```

O código foi o suficiente para passar no teste novamente.

Obs.: O teste `test_to_dict_none_empty` foi atualizado, pois não é possível que uma Schedule tenha `permissions` vazia:

In [ ]:
```
def test_to_dict_none_empty(self):
    # Test to_dict when attributes are None or empty
    empty_schedule = Schedule(self.id, self.title, None, [('userid1', 'permissiontype1')], [])
    schedule_dict = empty_schedule.to_dict()
    self.assertEqual(schedule_dict, {
        "id": self.id,
        "title": self.title,
```

```
        "description": None,
        "permissions": [('userid1', 'permissiontype1')],
        "elements": []
    })
```

Obs: A chave "id" foi alterada para "_id", para melhor se adequar ao MongoDB. Os testes foram atualizados para refletir essa mudança:

In [ ]:
```
def test_to_dict(self):
    # Call the to_dict method
    schedule_dict = self.schedule.to_dict()

    # Check that the dictionary has the correct keys and values
    self.assertEqual(schedule_dict, {
        "_id": self.id,
        "title": self.title,
        "description": self.description,
        "permissions": self.permissions,
        "elements": self.elements
})

def test_to_dict_none_empty(self):
    # Test to_dict when attributes are None or empty
    empty_schedule = Schedule(self.id, self.title, None, [('userid1', 'permissiontype1')], [])
    schedule_dict = empty_schedule.to_dict()
    self.assertEqual(schedule_dict, {
        "_id": self.id,
        "title": self.title,
        "description": None,
        "permissions": [('userid1', 'permissiontype1')],
        "elements": []
    })
```

O código teve uma pequena alteração para passar nos testes atualizados:

In [ ]:
```
def to_dict(self) -> dict:
    '''
        Returns a dictionary representation of the schedule,
        which can be used to create a JSON object.

        Returns:
            dict -- Dictionary representation of the schedule.
    '''
    return {
        "_id": self.__id,
        "title": self.title,
        "description": self.description,
        "permissions": self.__permissions,
        "elements": self.__elements
    }
```

Get Elements

Para este método, foi criado o seguinte teste inicial:

In [ ]:
```
def test_get_elements(self):
    # Test that get_elements returns the correct elements
    element_management = ElementManagement.get_instance()
    schedule = Schedule('id', 'title', 'description', [], ['elementid1', 'elementid2'])
    elements = schedule.get_elements()
    expected_elements = [element_management.elements['elementid1'], element_management.elements['elementid2']]
    self.assertEqual(elements, expected_elements)
```

Para realizar o teste, foi necessário utilizar de mocks, uma vez que o método `get_elements` da classe `Schedule` utiliza o método `get_element` da classe `ElementManagement`. Para isso, um mock foi criado:

In [ ]:
```
from datetime import datetime, timedelta


class Element:
    def __init__(self, start_time: datetime, end_time: datetime, element_type: str):
        self.start_time = start_time
        self.end_time = end_time
        self.type = element_type

class ElementManagement:
    _instance = None

    @classmethod
    def get_instance(cls):
```

```
        if cls._instance is None:
            cls._instance = cls()
        return cls._instance

    def __init__(self):
        self.elements = {
            'elementid1': Element(
                datetime.now(), datetime.now() + timedelta(hours=1), 'evento'),
            'elementid2': Element(
                datetime.now() + timedelta(hours=15),
                datetime.now() + timedelta(hours=16), 'evento'),
            'elementid3': Element(
                datetime.now() + timedelta(hours=8),
                datetime.now() + timedelta(hours=9), 'tarefa'),
            'elementid4': Element(
                datetime.now() + timedelta(hours=9),
                datetime.now() + timedelta(hours=10), 'evento'),
            'elementid5': Element(
                datetime.now() + timedelta(hours=12),
                datetime.now() + timedelta(hours=14), 'evento'),
            'elementid6': Element(
                datetime.now() + timedelta(hours=18),
                datetime.now() + timedelta(hours=19), 'evento'),
            'elementid7': Element(
                datetime.now() + timedelta(hours=19),
                datetime.now() + timedelta(hours=20), 'evento'),
        }

    def get_element(self, element_id: str) -> Element:
        return self.elements[element_id]
```

O código inicial foi o seguinte:

In [ ]:
```
def get_elements(self, types =  []) -> list:
    '''
        Returns a list of elements IDs for elements that are displayed in the schedule.

        Arguments:
            types -- list of element types.

        Returns:
            [Element] -- List of element instances in the schedule that have the specified types.
    '''
    element_management = ElementManagement.get_instance()
    elements = []
    for element_id in self.__elements:
        element = element_management.get_element(element_id)
        elements.append(element)
    return elements
```

O código inicial foi o suficiente para passar no teste. Adicionamos mais um teste para verificar a funcionalidade de filtro por tipo de elemento:

In [ ]:
```
def test_get_elements_with_types(self):
    # Test that get_elements returns the correct elements with the specified types
    element_management = ElementManagement.get_instance()
    schedule = Schedule('id', 'title', 'description', {'userid1': 'permissiontype1'}, ['elementid1', 'elementid
    elements = schedule.get_elements(['evento'])
    expected_elements = [element_management.elements['elementid1'], element_management.elements['elementid4']]
    self.assertEqual(elements, expected_elements)
```

O código não foi suficiente para passar no teste, sendo necessário atualizá-lo:

In [ ]:
```
def get_elements(self, types =  []) -> list:
    '''
        Returns a list of elements IDs for elements that are displayed in the schedule.

        Arguments:
            types -- list of element types.

        Returns:
            [Element] -- List of element instances in the schedule that have the specified types.
    '''
    element_management = ElementManagement.get_instance()
    elements = []
    for element_id in self.__elements:
        element = element_management.get_element(element_id)
        if not types or element.type in types:
            elements.append(element)
    return elements
```

O código foi o suficiente para passar no teste. Realizamos mais dois testes: um para veriicar o que ocorre quando não há elementos e outro para verificar o que ocorre quando o tipo de elemento não existe:

In [ ]:
```
def test_get_elements_empty(self):
    # Test that get_elements returns an empty list when there are no elements
    schedule = Schedule('id', 'title', 'description', [], [])
    elements = schedule.get_elements()
    self.assertEqual(elements, [])

def test_get_elements_nonexistent_type(self):
    # Test that get_elements returns an empty list when there are no elements with the specified type
    schedule = Schedule('id', 'title', 'description', [], ['elementid1', 'elementid2', 'elementid3', 'elementid
    elements = schedule.get_elements(['citrico'])
    self.assertEqual(elements, [])
```

O código foi o suficiente para passar nos testes.

Get Users

Para este método, foi criado o seguinte teste inicial:

In [ ]:
```
def test_get_users(self):
    # Test that get_users returns the correct users
    user_management = UserManagement.get_instance()
    schedule = Schedule('id', 'title', 'description', [('userid1', 'permissiontype1'), ('userid2', 'permissionty
    users = schedule.get_users()
    expected_users = [user_management.users['userid1'], user_management.users['userid2']]
    self.assertEqual(users, expected_users)
```

Para realizar o teste, foi necessário utilizar de mocks, uma vez que o método `get_users` da classe `Schedule` utiliza o método `get_user` da classe `UserManagement`. Para isso, um mock foi criado:

In [ ]:
```
class User:
    def __init__(self, user_id: str):
        self.__id = user_id

    @property
    def id(self):
        return self.__id

class UserManagement:
    _instance = None

    @classmethod
    def get_instance(cls):
        if cls._instance is None:
            cls._instance = cls()
        return cls._instance

    def __init__(self):
        self.users = {
            'userid1': User('userid1'),
            'userid2': User('userid2'),
            'userid3': User('userid3'),
            'userid4': User('userid4'),
            'userid5': User('userid5'),
            'userid6': User('userid6'),
            'userid7': User('userid7'),
            'userid8': User('userid8'),
            'userid9': User('userid9'),
            'userid10': User('userid10'),
        }

    def get_user(self, user_id: str) -> User:
        return self.users[user_id]
```

O código inicial foi o seguinte:

In [ ]:
```
def get_users(self, permission_types = []) -> list:
    '''
        Returns a list of users that have the specified permission types.
        If no permission types are specified, returns all the users in the schedule.

        Arguments:
            permission_types -- list of permission types.

        Returns:
            [User] -- List of users that have the specified permission types.
    '''
```

```
    user_management = UserManagement.get_instance()
    users = []
    for user_id, permission_type in self.__permissions:
        user = user_management.get_user(user_id)
        users.append(user)
    return users
```

O código inicial foi o suficiente para passar no teste. Adicionamos mais um teste para verificar a funcionalidade de filtro por tipo de permissão:

In [ ]:
```
def test_get_users_with_permission_types(self):
    # Test that get_users returns the correct users with the specified permission types
    user_management = UserManagement.get_instance()
    schedule = Schedule('id', 'title', 'description', [('userid1', 'permissiontype1'), ('userid2', 'permissionty
    users = schedule.get_users(['permissiontype1'])
    expected_users = [user_management.users['userid1'], user_management.users['userid3']]
    self.assertEqual(users, expected_users)
```

O código não foi o suficiente para passar no teste, sendo necessário atualizá-lo:

In [ ]:
```
def get_users(self, permission_types = []) -> list:
    '''
        Returns a list of users that have the specified permission types.
        If no permission types are specified, returns all the users in the schedule.

        Arguments:
            permission_types -- list of permission types.

        Returns:
            [User] -- List of users that have the specified permission types.
    '''

    user_management = UserManagement.get_instance()
    users = []
    for user_id, permission_type in self.__permissions:
        if not permission_types or permission_type in permission_types:
            user = user_management.get_user(user_id)
            users.append(user)
    return users
```

O código foi o suficiente para passar no teste. Por fim, criamos dois testes: um para verificar o comportamento do método quando não há usuários e outro para verificar o comportamento do método quando o tipo de permissão não existe:

In [ ]:
```
def test_get_users_empty(self):
    # Test that get_users returns an empty list when there are no users
    schedule = Schedule('id', 'title', 'description', [], [])
    users = schedule.get_users()
    self.assertEqual(users, [])

def test_get_users_nonexistent_permission_type(self):
    # Test that get_users returns an empty list when there are no users with the specified permission type
    schedule = Schedule('id', 'title', 'description', [('userid1', 'permissiontype1'), ('userid2', 'permissionty
    users = schedule.get_users(['permissiontype3'])
    self.assertEqual(users, [])
```

O código foi o suficiente para passar no teste.

OBS.: O teste `test_get_users_empty` foi removido, pois não é permitido que um Schedule não tenha usuários.

Refactor

Permissions foi alterado de uma tupla (user_id, permission) para um dicionário {user_id: permission}. Os testes foram atualizados para refletir essa mudança:

In [ ]:
```
def test_to_dict_none_empty(self):
    # Test to_dict when attributes are None or empty
    empty_schedule = Schedule(self.id, self.title, None, {'userid1': 'permissiontype1'}, [])
    schedule_dict = empty_schedule.to_dict()
    self.assertEqual(schedule_dict, {
        "_id": self.id,
        "title": self.title,
        "description": None,
        "permissions": {'userid1': 'permissiontype1'},
        "elements": []
    })
```

```
def test_get_elements(self):
    # Test that get_elements returns the correct elements
    element_management = ElementManagement.get_instance()
    schedule = Schedule('id', 'title', 'description', {'userid1': 'permissiontype1'}, ['elementid1', 'elementid
    elements = schedule.get_elements()
    expected_elements = [element_management.elements['elementid1'], element_management.elements['elementid2']]
    self.assertEqual(elements, expected_elements)

def test_get_elements_with_types(self):
    # Test that get_elements returns the correct elements with the specified types
    element_management = ElementManagement.get_instance()
    schedule = Schedule('id', 'title', 'description', {'userid1': 'permissiontype1'}, ['elementid1', 'elementid
    elements = schedule.get_elements(['evento'])
    expected_elements = [element_management.elements['elementid1'], element_management.elements['elementid4']]
    self.assertEqual(elements, expected_elements)

def test_get_elements_empty(self):
    # Test that get_elements returns an empty list when there are no elements
    schedule = Schedule('id', 'title', 'description', {'userid1': 'permissiontype1'}, [])
    elements = schedule.get_elements()
    self.assertEqual(elements, [])

def test_get_elements_nonexistent_type(self):
    # Test that get_elements returns an empty list when there are no elements with the specified type
    schedule = Schedule('id', 'title', 'description', {'userid1': 'permissiontype1'}, ['elementid1', 'elementid
    elements = schedule.get_elements(['citrico'])
    self.assertEqual(elements, [])

def test_get_users(self):
    # Test that get_users returns the correct users
    user_management = UserManagement.get_instance()
    schedule = Schedule('id', 'title', 'description', {'userid1': 'permissiontype1', 'userid2': 'permissiontype
    users = schedule.get_users()
    expected_users = [user_management.users['userid1'], user_management.users['userid2']]
    self.assertEqual(users, expected_users)

def test_get_users_with_permission_types(self):
    # Test that get_users returns the correct users with the specified permission types
    user_management = UserManagement.get_instance()
    schedule = Schedule('id', 'title', 'description', {'userid1': 'permissiontype1', 'userid2': 'permissiontype
    users = schedule.get_users(['permissiontype1'])
    expected_users = [user_management.users['userid1'], user_management.users['userid3']]
    self.assertEqual(users, expected_users)

def test_get_users_nonexistent_permission_type(self):
    # Test that get_users returns an empty list when there are no users with the specified permission type
    schedule = Schedule('id', 'title', 'description', {'userid1': 'permissiontype1', 'userid2': 'permissiontype
    users = schedule.get_users(['permissiontype3'])
    self.assertEqual(users, [])
```

Para isso, foi necessário atualizar o método `get_users` e o `init` da classe `Schedule`:

In [ ]:

```
def __init__(self, schedule_id: str, title: str, description: str,
        permissions: dict, elements: [str] = None):
    """
        Schedule constructor.
        Arguments:
            schedule_id -- id of the schedule.
            title -- title of the schedule.
            description -- description of the schedule.
            permissions -- list of tuples (user_id, permission_type)
                    that represent the permissions of the users in the schedule.
            elements -- list of elements ids that are displayed in the schedule.
    """
    self.__id = schedule_id
    self.set_title(title)
    self.set_description(description)
    self.__permissions = permissions
    self.__elements = elements if elements else []

def get_users(self, permission_types = []) -> list:
    '''
        Returns a list of users that have the specified permission types.
        If no permission types are specified, returns all the users in the schedule.

        Arguments:
            permission_types -- list of permission types.

        Returns:
            [User] -- List of users that have the specified permission types.
    '''

    user_management = UserManagement.get_instance()
```

```
    users = []
    for user_id, permission_type in self.__permissions.items():
        if not permission_types or permission_type in permission_types:
            user = user_management.get_user(user_id)
            users.append(user)
    return users
```

Elements Setter

Precisamos que elements possa ser alterado, para isso, precisamos de um @element.setter. Os seguintes testes iniciais foram criados:

In [ ]:
```
def test_elements_setter_accepts_valid_input(self):
    # Arrange
    schedule = Schedule("schedule1", "Title", "Description", {"user1": "read"}, ["element1"])
    new_elements = ["element2", "element3"]
    # Act
    schedule.elements = new_elements
    # Assert
    self.assertEqual(schedule.elements, new_elements)

def test_elements_setter_raises_error_on_invalid_input(self):
    # Arrange
    schedule = Schedule("schedule1", "Title", "Description", {"user1": "read"}, ["element1"])
    invalid_elements = "element2"
    # Act & Assert
    with self.assertRaises(ValueError):
        schedule.elements = invalid_elements
```

O código inicial foi o seguinte:

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

O código foi suficiente.

Refactor: Observer

Com a implementação do padrão Observer, obtivemos três métodos:

In [ ]:
```
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)
```

Para testar a conexão com o Observer, foi criado o seguinte teste:

In [ ]:
```
def test_changes_on_elements_calls_schedule_management_update_schedule(self):
    # Arrange
    schedule = Schedule("schedule1", "Title", "Description", {"user1": "read"}, ["element1"])
    schedule_management = ScheduleManagement.get_instance(database_module=MagicMock())
    schedule_management.update_schedule = MagicMock()
    schedule.attach(schedule_management)
    # Act
```

```
    schedule.elements = ["changed_element1"]
    # Assert
    schedule_management.update_schedule.assert_called_once_with("schedule1")
```

O teste passou. Testes específicos de Observer podem ser vistos no notebook `history_schedule_observer.ipynb`.

Mudanças foram feitas nos testes do método `get_users` para refletir as mudanças:

In [ ]:
```
def test_get_users(self):
    """Test that get_users returns the correct users"""
    # Arrange
    schedule = Schedule("schedule1", "Test Title", "Test Description",
        {"user1": "read", "user2": "write", "user3": "read"}, ["element1"])
    user_ids = ["user1", "user2", "user3"]
    mock_user = MagicMock()
    mock_user_management = MagicMock()
    mock_user_management.get_user.return_value = mock_user

    with patch.object(UserManagement, 'get_instance', return_value=mock_user_management):

        # Act
        users = schedule.get_users()

        # Assert
        self.assertEqual(len(users), len(user_ids))
        for user in users:
            self.assertEqual(user, mock_user)


def test_get_users_with_permission_types(self):
    """Test that get_users returns the correct users with the specified permission types"""
    # Arrange
    schedule = Schedule("schedule1", "Test Title", "Test Description",
                        {"user1": "type1", "user2": "type2", "user3": "type1"}, ["element1"])
    user_ids = ["user1", "user3"]
    mock_user = MagicMock()
    mock_user_management = MagicMock()
    mock_user_management.get_user.return_value = mock_user

    with patch.object(UserManagement, 'get_instance', return_value=mock_user_management):

        # Act
        users = schedule.get_users(['type1'])

        # Assert
        self.assertEqual(len(users), len(user_ids))
        for user in users:
            self.assertEqual(user, mock_user)
```

Também houve mudanças nos testes do método `get_elements`:

In [ ]:
```
def test_get_elements(self):
    """Test that get_elements returns the correct elements"""
    # Arrange
    schedule = Schedule("schedule1", "Test Title", "Test Description",
                        {"user1": "type1", "user2": "type2", "user3": "type1"}, ["element1", "element2"])
    element_ids = ["element1", "element2"]
    mock_element = MagicMock()
    mock_element_management = MagicMock()
    mock_element_management.get_element.return_value = mock_element

    with patch.object(ElementManagement, 'get_instance', return_value=mock_element_management):

        # Act
        elements = schedule.get_elements()

        # Assert
        self.assertEqual(len(elements), len(element_ids))
        for element in elements:
            self.assertEqual(element, mock_element)

def test_get_elements_with_types(self):
    """Test that get_elements returns the correct elements with the specified types"""
    # Arrange
    schedule = Schedule("schedule1", "Test Title", "Test Description",
                        {"user1": "type1", "user2": "type2", "user3": "type1"},
                        ["element1", "element2", "element3", "element4"])
    element_ids = ["element1", "element4"]
    mock_element_evento = MagicMock()
    mock_element_evento.type = 'evento'
    mock_element_other = MagicMock()
    mock_element_other.type = 'other'
```

```
    mock_element_management = MagicMock()
    mock_element_management.get_element.side_effect = lambda x: mock_element_evento if x in element_ids else mo

    with patch.object(ElementManagement, 'get_instance', return_value=mock_element_management):

        # Act
        elements = schedule.get_elements(['evento'])

        # Assert
        self.assertEqual(len(elements), len(element_ids))
        for element in elements:
            self.assertEqual(element.type, 'evento')

def test_get_elements_nonexistent_type(self):
    """Test that get_elements returns an empty list when there are no elements with the specified type"""
    # Arrange
    schedule = Schedule("schedule1", "Test Title", "Test Description",
                        {"user1": "type1", "user2": "type2", "user3": "type1"},
                        ["element1", "element2", "element3", "element4"])
    mock_element_evento = MagicMock()
    mock_element_evento.type = 'evento'
    mock_element_other = MagicMock()
    mock_element_other.type = 'other'
    mock_element_management = MagicMock()
    mock_element_management.get_element.side_effect = lambda x: mock_element_evento if x == 'element1' else moc

    with patch.object(ElementManagement, 'get_instance', return_value=mock_element_management):

        # Act
        elements = schedule.get_elements(['citrico'])

        # Assert
        self.assertEqual(elements, [])
```

Uma pequena modificação também foi feita no seguinte teste, pois o erro levantado foi alterado de `ValueError` para `TypeError`:

In [ ]:
```
def test_elements_setter_raises_error_on_invalid_input(self):
    # Arrange
    schedule = Schedule("schedule1", "Title", "Description", {"user1": "read"}, ["element1"])
    invalid_elements = "element2"
    # Act & Assert
    with self.assertRaises(TypeError):
        schedule.elements = invalid_elements
```