Histórico do TaskEvent

Notebook a ponto de registrar o avanço da classe TaskEvent. Não é para ser executado. Podemos ver usos de TDD e refatoração.

Properties

Testes a serem realizados:

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

def test_type_property(self):
    # Test the schedules property
    self.assertEqual(self.task.type, self.element_type)

def test_schedules_property(self):
    # Test the schedules property
    self.assertEqual(self.task.schedules, self.schedules)
```

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

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

@property
def schedules(self):
    return self.__schedules

@property
def type(self):
    return self.__element_type
```

set_due_date

Testes a serem realizados:

In [ ]:
```
def test_set_due_date_valid(self):
    # Test setting a valid due date
    valid_due_date = datetime(2023, 1, 1)
    self.task.set_due_date(valid_due_date)
    self.assertEqual(self.task.due_date, valid_due_date)
```

Este código foi o suficiente para passar no teste:

In [ ]:
```
def set_due_date(self, due_date: datetime) -> None:
    '''
        Sets the due date of the task.

        Arguments:
            due_date -- due date of the task.
    '''
    self.due_date = due_date
```

Em seguida, foram criados os seguintes testes, que impedem que a data seja vazia ou diferente de datatime:

In [ ]:
```
def test_set_due_date_not_datetime(self):
    # Test setting a due date that is not a datetime
    with self.assertRaises(TypeError):
        self.task.set_due_date(123)

def test_set_due_date_none(self):
    # Test setting a due date that is None
    with self.assertRaises(ValueError):
        self.task.set_due_date(None)
```

O código foi alterado para passar nos testes:

In [ ]:
```
def set_due_date(self, due_date: datetime) -> None:
    '''
        Sets the due date of the task.
```

```
        Arguments:
            due_date -- due date of the task.
    '''
    if due_date is None:
        raise ValueError("Due date cannot be None")
    elif type(due_date) != datetime:
        raise TypeError("Due date must be a datetime object")
    else:
        self.due_date = due_date
```

Após refactor:

In [ ]:
```
def set_due_date(self, due_date: datetime) -> None:
        """
        Sets the due date of the task.

        Arguments:
            due_date -- due date of the task.
        """
        if due_date is None:
            raise ValueError("Due date cannot be None")
        elif not isinstance(due_date, datetime):
            raise TypeError("Due date must be a datetime object")
        else:
            self.due_date = due_date
```

set_state

Testes a serem realizados:

In [ ]:
```
def test_set_state(self):
    # Verify if the state is set correctly
    states = ['incomplete', 'complete', 'cancelled']
    for state in states:
        self.task.set_state(state)
        self.assertEqual(self.task.state, state)
```

O código abaixo foi o suficiente para passar no teste:

In [ ]:
```
def set_state(self, state: str):
    """
    Sets the state of the task.

    Arguments:
        state -- The new state of the task.
    """
    self.state = state
```

Foi adicionado o seguinte teste para que quando o estado da tarefa não foi definido, o estado seja "incomplete":

In [ ]:
```
def test_set_state_none(self):
    # Verify if a None state set the state as incomplete
    self.task.set_state(None)
    self.assertEqual(self.task.state, 'incomplete')
```

Foram necessárias as seguintes alterações para passar no teste:

In [ ]:
```
def set_state(self, state: str):
    """
    Sets the state of the task.

    Arguments:
        state -- The new state of the task.
    """
    if state is None:
        self.state = 'incomplete'
    else:
        self.state = state
```

Foram adicionados os seguintes testes, que impedem que o estado seja diferente de "incomplete", "complete" e "cancelled":

In [ ]:
```
def test_set_state_invalid(self):
    # Verify if an invalid state raises a ValueError
    with self.assertRaises(ValueError):
        self.task.set_state("invalid")

def test_set_state_not_string(self):
```

```
    # Verify if a non-string state raises a TypeError
    with self.assertRaises(TypeError):
        self.task.set_state(123)
```

Foram necessárias as seguintes alterações para passar no teste:

In [ ]:
```
def set_state(self, state: str):
    """
    Sets the state of the task.

    Arguments:
        state -- The new state of the task.
    """
    if state is None:
        self.state = 'incomplete'
    else:
        if type(state) != str:
            raise TypeError("State must be a string")
        elif state not in ['incomplete', 'complete', 'cancelled']:
            raise ValueError("State must be either 'incomplete' or 'complete'")
        else:
            self.state = state
```

Após refactor:

In [ ]:
```
def set_state(self, state: str):
    """
    Sets the state of the task.

    Arguments:
        state -- The new state of the task.
    """
    valid_states = ['incomplete', 'complete', 'cancelled']

    if state is None:
        self.state = 'incomplete'
    elif not isinstance(state, str):
        raise TypeError("State must be a string")
    elif state not in valid_states:
        raise ValueError("State must be either 'incomplete', 'complete', or 'cancelled'")
    else:
        self.state = state
```

set_title

Testes a serem realizados:

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

Código suficiente para passar no teste:

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

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

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

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

        Arguments:
            title -- title of the task.
    '''
    if title is None:
        raise ValueError("Title cannot be None")
    elif type(title) != str:
        raise TypeError("Title must be a string")
    else:
        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.task.set_title("   ")

def test_set_title_empty(self):
    # Test setting an empty title
    with self.assertRaises(ValueError):
        self.task.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 task.

        Arguments:
            title -- title of the task.
    '''
    if title is None:
        raise ValueError("Title cannot be None")
    elif type(title) != str:
        raise TypeError("Title must be a string")
    elif not title.strip():
            raise ValueError("Title cannot be empty or blank")
    else:
        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.task.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.task.set_title(max_length_title)
    self.assertEqual(self.task.title, max_length_title)
```

Atualização do código para passar nos testes:

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

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

        if title is None:
            raise ValueError("Title cannot be None")
        elif type(title) != str:
            raise TypeError("Title must be a string")
        elif not title.strip():
            raise ValueError("Title cannot be empty or blank")
        elif len(title) > 50:
            raise ValueError("Title cannot have more than 50 characters")
        else:
            self.title = title
```

Set description of the task

Testes a serem realizados:

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

Código suficiente para passar no teste:

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

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

Em seguida, foram criados os seguintes testes para descrição 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.task.set_description(123)

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

Código atualizado para passar nos testes:

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

        Arguments:
            description -- description of the task.
    '''
    if description is not None:
        if type(description) != str:
            raise TypeError("Description must be a string")
    self.description = description
```

Adicionado testes para impedir descrições muito grandes (mais de 500 caracteres):

In [ ]:
```
def test_set_description_too_long(self):
    # Test setting a description that is too long
    with self.assertRaises(ValueError):
        self.task.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.task.set_description(max_length_description)
    self.assertEqual(self.task.description, max_length_description)
```

Atualização do código para passar nos testes:

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

        Arguments:
            description -- description of the task.
    '''
    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
```

Após refatoração:

In [ ]:
```
def set_description(self, description: str) -> None:
```

```
    """
    Sets the description of the task.

    Arguments:
        description -- description of the task.
    """
    if description is not None:
        if not isinstance(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
```

get_users

Foi realizado o teste inicial para verificar se a função retorna a lista de usuários correta, o código do teste será omitido nesse caso pois é bastante longo devido ao uso de 'MagicMock', mas pode ser visto em `test_task_event.py`:

In [ ]:
```
def test_get_users(self):
    """
    Test if the users returned match the ones that were set in the
    constructor
    """
```

O código abaixo foi suficiente para passar no teste.

In [ ]:
```
def get_users(self, filter_schedules = []) -> list:
    """
    Returns the users that are assigned to the event.

    Arguments:
        filter_schedules -- A list of schedules to filter the users.

    Returns:
        [user] -- The users that are assigned to the reminder.
    """
    schedule_manager = ScheduleManagement.get_instance()
    filter_schedules = [schedule for schedule in self.__schedules if schedule in filter_schedules]
    users = []
    for schedule_id in filter_schedules:
        schedule = schedule_manager.get_schedule(schedule_id)
        users += list(schedule.permissions.keys())

    user_manager = UserManagement.get_instance()

    users = [user_manager.get_user(id) for id in users]
    return users
```

Posteriormente, foi criado o teste para verificar se a função funciona quando não é especificado a lista de schedules.

In [ ]:
```
def test_get_users_nonfilter(self):
        """Test get_users when no filter of schedules are not specified.
        Default to all schedules.
        """
```

Foi necessário atualizar o código para passar no teste:

In [ ]:
```
def get_users(self, filter_schedules = []) -> list:
    """
    Returns the users that are assigned to the task.

    Arguments:
        filter_schedules -- A list of schedules to filter the users.

    Returns:
        [str] -- The users that are assigned to the task.
    """
    schedule_manager = ScheduleManagement.get_instance()
    if filter_schedules == []:
        filter_schedules = self.__schedules
    else:
        filter_schedules = [schedule for schedule in self.__schedules if schedule in filter_schedules]
    users = []
    for schedule_id in filter_schedules:
        schedule = schedule_manager.get_schedule(schedule_id)
        users += list(schedule.permissions.keys())

    user_manager = UserManagement.get_instance()
```

```
    users = [user_manager.get_user(id) for id in users]
    return users
```

Foi realizado o teste para verificar se o filtro funciona corretamente. O códidigo acima foi suficiente para passar no teste.

In [ ]:
```
def test_get_users_with_filter_by_schedules(self):
        """
        Test if the users returned match the ones that were set in the
        constructor and if they belong to the specified schedules
        """
```

Por fim, foi adicionado o teste para verificar se a função retorna uma lista sem usuários repetidos.

In [ ]:
```
def test_get_users_nonrepeat_users(self):
        """Test if the users returned are unique"""
```

Foi necessário atualizar o código para passar no teste:

In [ ]:
```
def get_users(self, filter_schedules = []) -> list:
    """
    Returns the users that are assigned to the task.

    Arguments:
        filter_schedules -- A list of schedules to filter the users.

    Returns:
        [str] -- The users that are assigned to the task.
    """
    schedule_manager = ScheduleManagement.get_instance()
    if filter_schedules == []:
        filter_schedules = self.__schedules
    else:
        filter_schedules = [schedule for schedule in self.__schedules if schedule in filter_schedules]
    users = []
    for schedule_id in filter_schedules:
        schedule = schedule_manager.get_schedule(schedule_id)
        users += list(schedule.permissions.keys())

    users = list(set(users)) # Remove duplicates

    user_manager = UserManagement.get_instance()

    users = [user_manager.get_user(id) for id in users]
    return users
```

Após refatorar o código, ficamos com:

In [ ]:
```
def get_users(self, filter_schedules = []) -> list:
        """
        Returns the users that are assigned to the task.

        Arguments:
            filter_schedules -- A list of schedules to filter the users.

        Returns:
            [user] -- The users that are assigned to the task.
        """
        schedule_manager = ScheduleManagement.get_instance()
        schedules_to_use = filter_schedules or self.__schedules

        users = set()
        for schedule_id in schedules_to_use:
            schedule = schedule_manager.get_schedule(schedule_id)
            users.update(schedule.permissions.keys())

        user_manager = UserManagement.get_instance()
        return [user_manager.get_user(id) for id in users]
```