Nota: Esse notebook não é para ser rodado, serve apenas como uma forma mais organizada de mostrar a evolução das funções testadas pelo método RED/GREEN

get_elements function

In [ ]:
```
import unittest
from datetime import datetime, timedelta
from tests.test_users.mocks import Schedule, ScheduleManagement, Element, ElementManagement
from src.user.user_model import *
```

RED 1:

In [ ]:
```
def test_get_all_elements(self):
        # Test getting all element ids from user schedules, without repetition
        user = User("id", "username", "email", ["id1", "id2"])
        result = user.get_elements()
        self.assertEqual(sorted(result), ['elementid1', 'elementid2',
                                          'elementid3', 'elementid4', 'elementid5'])
```

In [ ]:
```
def get_elements(self, schedules: list=None) -> list:
        '''
        returns a list of ids of elements of all schedules (or only specific ones)
        that the user is a part of
        '''
        elements = []
        for schedule in schedules:
            schedule = ScheduleManagement().get_schedule(schedule)
            elements += schedule.get_elements()

        return elements
```

GREEN 1:

In [ ]:
```
def get_elements(self, schedules: list=None) -> list:
    '''
    returns a list of ids of elements of all schedules (or only specific ones)
    that the user is a part of
    '''
    elements = []
    for schedule in schedules:
        schedule = ScheduleManagement().get_schedule(schedule)
        elements += schedule.get_elements()

    elements = list(set(elements)) # remove duplicates
    return elements
```

REFACTOR 1 (Para tratar classes Management como singletons):

In [ ]:
```
def get_elements(self, schedules: list=None) -> list:
    '''
    returns a list of ids of elements of all schedules (or only specific ones)
    that the user is a part of
    '''
    elements = []
    for schedule in schedules:
        schedule = self.schedule_management.get_schedule(schedule)
        elements += schedule.get_elements()

    elements = list(set(elements)) # remove duplicates
    return elements
```

RED 2

In [ ]:
```
def test_get_filtered_elements(self):
        # Test getting all element ids from user schedules, with filter
        user = User("id", "username", "email", ["id1", "id2", "id3"])
        result = user.get_elements(["id1", "id3"])
        self.assertEqual(sorted(result), ['elementid1', 'elementid2',
                                                'elementid5', 'elementid6', 'elementid7'])
```

In [ ]:
```
def get_elements(self, schedules: list=None) -> list:
    '''
    returns a list of ids of elements of all schedules (or only specific ones)
    that the user is a part of
    '''
    elements = []
    for schedule in schedules:
    schedule = self.schedule_management.get_schedule(schedule)
    elements += schedule.get_elements()

elements = list(set(elements)) # remove duplicates
return elements
```

GREEN 2

In [ ]:
```
def get_elements(self, schedules: list=None) -> list:
    '''
    returns a list of ids of elements of all schedules (or only specific ones)
    that the user is a part of
    '''
    if not schedules:
        schedules = self.schedules

    elements = []
    for schedule in schedules:
        schedule = self.schedule_management.get_schedule(schedule)
        elements += schedule.get_elements()

    elements = list(set(elements)) # remove duplicates
    return elements
```

RED 3

In [ ]:
```
def test_get_elements_from_schedule_user_isnt_in(self):
    # Test getting all element ids from a nonexistent schedule
    user = User("id", "username", "email", ["id1", "id2"])
    with self.assertRaises(UserNotInSchedule) as context:
        user.get_elements(["id3"])
    self.assertEqual(str(context.exception), 'Usuário não está nessa agenda: id3')
```

GREEN 3:

In [ ]:
```
def get_elements(self, schedules: list=None) -> list:
        '''
        Get all elements from the user schedules, without repetition, or
        from a list of filtered schedules

        Args:
            schedules: list of schedules ids

        Returns:
            A list of elements ids the user is a part of
        '''
        if not schedules:
            schedules = self.get_schedules()
        else:
            for schedule in schedules:
                if schedule not in self.get_schedules():
                    raise UserNotInSchedule(
                        f"Usuário não está nessa agenda: {schedule}")

        elements = []
        for schedule in schedules:
            schedule = self.schedule_management.get_schedule(schedule)
            elements += schedule.get_elements()

        elements = list(set(elements))
        return elements
```

Função set_mail:

GREEN 1:

In [ ]:
```
def test_set_email(self):
        # Test setting an email
        user = User("id", "username", "email", ["id1", "id2"])
        user.set_email("new_email")
        self.assertEqual(user.email, "new_email")
```

In [ ]:
```
def set_email(self, email: str):
        """
        Set the user name

        Args:
            username: user name
        """
        self.email = email
```

RED 2:

In [ ]:
```
def test_set_email_with_trailing_space(self):
        # Test setting an email with trailing space
        user = User("id", "username", "email", ["id1", "id2"])
        user.set_email("new_email ")
        self.assertEqual(user.email, "new_email")
```

GREEN 2:

In [ ]:
```
def set_email(self, email: str):
        """
        Set the user name

        Args:
            username: user name
        """
        self.email = email.strip()
```

RED 3:

In [ ]:
```
def test_set_email_with_int(self):
        # Test setting an email with trailing space
        user = User("id", "username", "email", ["id1", "id2"])
        with self.assertRaises(TypeError) as context:
            user.set_email(123)
        self.assertEqual(str(context.exception),
                         "O email deve ser uma string")
```

GREEN 3:

In [ ]:
```
def set_email(self, email: str):
        """
        Set the user name

        Args:
            username: user name
        """
        if type(email) != str:
            raise TypeError("O email deve ser uma string")
        else:
            self.email = email.strip()
```

RED 4:

In [ ]:
```
def test_set_blank_email(self):
        # Test setting a blank email
        user = User("id", "username", "email", ["id1", "id2"])
        with self.assertRaises(EmailCantBeBlank) as context:
            user.set_email("")
        self.assertEqual(str(context.exception),
                         "O email não pode ser vazio")
```

GREEN 4:

In [ ]:
```
def set_email(self, email: str):
    """
    Set the user name

    Args:
        username: user name
    """
    if type(email) != str:
        raise TypeError("O email deve ser uma string")
    elif email == "":
        raise EmailCantBeBlank("O email não pode ser vazio")
    else:
        self.email = email.strip()
```

Check_disponibility():

Red 1

In [ ]:
```
def test_check_disponibility(self):
        user = User("id", "username", "email", ["id1", "id2"])
        time = (datetime.now() + timedelta(hours=2),
                datetime.now() + timedelta(hours=3))
        result = user.check_disponibility(time)
        self.assertTrue(result)
```

Green 1

```
In [ ]: def check_disponibility(self, time: tuple) -> bool:
            """
            Checks if the user is available at a given time, based on the user's
            schedules and elements. It should not raise a conflict if the type
            of the element is not 'evento'.

            Args:
                time: tuple with the start and end time to be checked

            Returns:
                True if the user is available, False otherwise
            """
            element_ids = self.get_elements()
            element_management = ElementManagement.get_instance()

            for element_id in element_ids:
                element = element_management.get_element(element_id)

                # Check if the start time of the element is within the given time period
                if time[0] <= element.start_time <= time[1]:
                    return False

                # Check if the end time of the element is within the given time period
                if time[0] <= element.end_time <= time[1]:
                    return False

                # Check if the given time period is within the start
                # and end time of the element
                if element.start_time <= time[0] <= element.end_time or \
                    element.start_time <= time[1] <= element.end_time:
                    return False

            return True
```

RED 2:

```
In [ ]: def test_check_disponibility_end_time_same_as_other_event_start_time(self):
                user = User("id", "username", "email", ["id1", "id2"])
                time = (datetime.now() + timedelta(hours=11),
                        self.ElementManagement.get_element("elementid5").start_time)
                result = user.check_disponibility(time)
                self.assertTrue(result)
```

GREEN 2:

```
In [ ]: def check_disponibility(self, time: tuple) -> bool:
            """
            Checks if the user is available at a given time, based on the user's
            schedules and elements. It should not raise a conflict if the type
            of the element is not 'evento'.

            Args:
                time: tuple with the start and end time to be checked

            Returns:
                True if the user is available, False otherwise
            """
            element_ids = self.get_elements()
            element_management = ElementManagement.get_instance()

            for element_id in element_ids:
                element = element_management.get_element(element_id)

                # Check if the start time of the element is within the given time period
                if time[0] <= element.start_time < time[1]:
                    return False

                # Check if the end time of the element is within the given time period
                if time[0] < element.end_time <= time[1]:
                    return False

                # Check if the given time period is within the start
                # and end time of the element
                if element.start_time <= time[0] < element.end_time or \
                    element.start_time < time[1] <= element.end_time:
                    return False

            return True
```

RED 3:

```
In [ ]: def test_check_disponibility_ignoring_non_event_elements(self):
                user = User("id", "username", "email", ["id1", "id2"])
                time = (self.ElementManagement.get_element("elementid3").start_time,
                        self.ElementManagement.get_element("elementid3").end_time)
                result = user.check_disponibility(time)
                self.assertTrue(result)
```

GREEN 3:

```
In [ ]: def check_disponibility(self, time: tuple) -> bool:
                """
                Checks if the user is available at a given time, based on the user's
                schedules and elements. It should not raise a conflict if the type
                of the element is not 'evento'.

                Args:
                    time: tuple with the start and end time to be checked

                Returns:
                    True if the user is available, False otherwise
                """
                element_ids = self.get_elements()
                element_management = ElementManagement.get_instance()

                for element_id in element_ids:
                    element = element_management.get_element(element_id)
                    if element.type != 'evento':
                        continue

                    # Check if the start time of the element is within the given time period
                    if time[0] <= element.start_time < time[1]:
                        return False

                    # Check if the end time of the element is within the given time period
                    if time[0] < element.end_time <= time[1]:
                        return False

                    # Check if the given time period is within the start
                    # and end time of the element
                    if element.start_time <= time[0] < element.end_time or \
                        element.start_time < time[1] <= element.end_time:
                        return False

                return True
```

RED 4:

```
In [ ]: def test_check_disponibility_input_type_exception(self):
                user = User("id", "username", "email", ["id1", "id2"])
                time = "time"
                with self.assertRaises(TypeError) as context:
                    user.check_disponibility(time)
                self.assertEqual(str(context.exception),
                                 "O horário deve ser uma tupla")
```

GREEN 4:

```
In [ ]: def check_disponibility(self, time: tuple) -> bool:
                """
                Checks if the user is available at a given time, based on the user's
                schedules and elements. It should not raise a conflict if the type
                of the element is not 'evento'.

                Args:
                    time: tuple with the start and end time to be checked

                Returns:
                    True if the user is available, False otherwise
                """
                if type(time) != tuple:
                    raise TypeError("O horário deve ser uma tupla")

                element_ids = self.get_elements()
                element_management = ElementManagement.get_instance()

                for element_id in element_ids:
                    element = element_management.get_element(element_id)
                    if element.type != 'evento':
```

```
                continue

            # Check if the start time of the element is within the given time period
            if time[0] <= element.start_time < time[1]:
                return False

            # Check if the end time of the element is within the given time period
            if time[0] < element.end_time <= time[1]:
                return False

            # Check if the given time period is within the start
            # and end time of the element
            if element.start_time <= time[0] < element.end_time or \
                element.start_time < time[1] <= element.end_time:
                return False

        return True
```

RED 5:

In [ ]:
```
def test_check_disponibility_not_datetime(self):
        user = User("id", "username", "email", ["id1", "id2"])
        time = (123, 123)
        with self.assertRaises(TypeError) as context:
            user.check_disponibility(time)
        self.assertEqual(str(context.exception),
                         "A tupla de horário deve conter objetos datetime")
```

GREEN 5:

In [ ]:
```
def check_disponibility(self, time: tuple) -> bool:
        """
        Checks if the user is available at a given time, based on the user's
        schedules and elements. It should not raise a conflict if the type
        of the element is not 'evento'.

        Args:
            time: tuple with the start and end time to be checked

        Returns:
            True if the user is available, False otherwise
        """
        if type(time) != tuple:
            raise TypeError("O horário deve ser uma tupla")
        if type(time[0]) != datetime or type(time[1]) != datetime:
            raise TypeError("A tupla de horário deve conter objetos datetime")

        element_ids = self.get_elements()
        element_management = ElementManagement.get_instance()

        for element_id in element_ids:
            element = element_management.get_element(element_id)
            if element.type != 'evento':
                continue

            # Check if the start time of the element is within the given time period
            if time[0] <= element.start_time < time[1]:
                return False

            # Check if the end time of the element is within the given time period
            if time[0] < element.end_time <= time[1]:
                return False

            # Check if the given time period is within the start
            # and end time of the element
            if element.start_time <= time[0] < element.end_time or \
                element.start_time < time[1] <= element.end_time:
                return False

        return True
```

RED 6:

In [ ]:
```
def test_check_disponibility_short_tuple(self):
        user = User("id", "username", "email", ["id1", "id2"])
        time = (datetime.now(),)
        with self.assertRaises(TupleWithLessThanTwoDatetimeObjects) as context:
            user.check_disponibility(time)
        self.assertEqual(str(context.exception),
                         "A tupla de horário deve conter pelo menos "\
                         "dois objetos datetime")
```

GREEN 6:

```
In [ ]: def check_disponibility(self, time: tuple) -> bool:
                """
                Checks if the user is available at a given time, based on the user's
                schedules and elements. It should not raise a conflict if the type
                of the element is not 'evento'.

                Args:
                    time: tuple with the start and end time to be checked

                Returns:
                    True if the user is available, False otherwise
                """
                if type(time) != tuple:
                    raise TypeError("O horário deve ser uma tupla")
                if len(time) < 2:
                    raise TupleWithLessThanTwoDatetimeObjects(
                        "A tupla de horário deve conter pelo menos dois objetos datetime")
                if type(time[0]) != datetime or type(time[1]) != datetime:
                    raise TypeError("A tupla de horário deve conter objetos datetime")

                element_ids = self.get_elements()
                element_management = ElementManagement.get_instance()

                for element_id in element_ids:
                    element = element_management.get_element(element_id)
                    if element.type != 'evento':
                        continue

                    # Check if the start time of the element is within the given time period
                    if time[0] <= element.start_time < time[1]:
                        return False

                    # Check if the end time of the element is within the given time period
                    if time[0] < element.end_time <= time[1]:
                        return False

                    # Check if the given time period is within the start
                    # and end time of the element
                    if element.start_time <= time[0] < element.end_time or \
                        element.start_time < time[1] <= element.end_time:
                        return False

                return True
```

Refactor

Mudanças feitas após Observer Pattern:

Get_elements:

```
In [ ]: def test_get_all_elements(self):
            """Test getting all element ids from user schedules, without repetition"""
            # Arrange
            user = User("id", "username", "email", ["schedule1", "schedule2"])
            mock_element1 = MagicMock()
            mock_element1.id = 'element_id1'
            mock_element2 = MagicMock()
            mock_element2.id = 'element_id2'
            mock_element3 = MagicMock()
            mock_element3.id = 'element_id3'
            mock_schedule1 = MagicMock()
            mock_schedule1.get_elements.return_value = [mock_element1, mock_element2]
            mock_schedule2 = MagicMock()
            mock_schedule2.get_elements.return_value = [mock_element3]
            mock_schedule_management = MagicMock()

            with patch.object(self.schedule_management, 'get_instance', return_value=mock_schedule_management), \
                    patch.object(self.schedule_management, 'get_schedule', side_effect=lambda x: mock_schedule1 if x ==

                # Act
                elements = user.get_elements()

                # Assert
                self.assertEqual(sorted([element.id for element in elements]), ['element_id1', 'element_id2', 'element_

        def test_get_filtered_elements(self):
            """Test getting element ids from specified user schedules"""
            # Arrange
```

```
    user = User("id", "username", "email", ["schedule1", "schedule2"])
    mock_element1 = MagicMock()
    mock_element1.id = 'element_id1'
    mock_element2 = MagicMock()
    mock_element2.id = 'element_id2'
    mock_element3 = MagicMock()
    mock_element3.id = 'element_id3'
    mock_schedule1 = MagicMock()
    mock_schedule1.get_elements.return_value = [mock_element1, mock_element2]
    mock_schedule2 = MagicMock()
    mock_schedule2.get_elements.return_value = [mock_element3]
    mock_schedule_management = MagicMock()

    with patch.object(self.schedule_management, 'get_instance', return_value=mock_schedule_management), \
        patch.object(self.schedule_management, 'get_schedule', side_effect=lambda x: mock_schedule1 if x == 'sc

        # Act
        elements = user.get_elements(schedules=['schedule1'])

        # Assert
        self.assertEqual(sorted([element.id for element in elements]), ['element_id1', 'element_id2'])
```

Check_disponibility:

```
In [ ]: def test_check_disponibility_true(self):
    """Test that check_disponibility returns True when there are no conflicts"""
    # Arrange
    user = User("id", "username", "email", ["schedule1", "schedule2"])
    mock_element1 = MagicMock()
    mock_element1.type = 'event'
    mock_element1.start_time = datetime.now() + timedelta(hours=3)
    mock_element1.end_time = datetime.now() + timedelta(hours=4)
    mock_element2 = MagicMock()
    mock_element2.type = 'event'
    mock_element2.start_time = datetime.now() + timedelta(hours=5)
    mock_element2.end_time = datetime.now() + timedelta(hours=6)
    mock_schedule1 = MagicMock()
    mock_schedule1.get_elements.return_value = [mock_element1]
    mock_schedule2 = MagicMock()
    mock_schedule2.get_elements.return_value = [mock_element2]
    mock_schedule_management = MagicMock()
    mock_element_management = MagicMock()
    with patch.object(self.schedule_management, 'get_instance', return_value=mock_schedule_management), \
        patch.object(self.schedule_management, 'get_schedule', side_effect=lambda x: mock_schedule1 if x == 'sc
        patch.object(self.element_management, 'get_instance', return_value=mock_element_management), \
        patch.object(self.element_management, 'get_element', side_effect=lambda x: mock_element1 if x == 'eleme
        # Act
        result = user.check_disponibility((datetime.now(), datetime.now() + timedelta(hours=2)))
        # Assert
        self.assertTrue(result)

def test_check_disponibility_end_time_same_as_other_event_start_time(self):
    """Test that check_disponibility returns True when the end time of the checked period is the same as the st
    # Arrange
    user = User("id", "username", "email", ["schedule1", "schedule2"])
    mock_element1 = MagicMock()
    mock_element1.type = 'event'
    mock_element1.start_time = datetime.now() + timedelta(hours=2)
    mock_element1.end_time = datetime.now() + timedelta(hours=3)
    mock_element2 = MagicMock()
    mock_element2.type = 'event'
    mock_element2.start_time = datetime.now() + timedelta(hours=4)
    mock_element2.end_time = datetime.now() + timedelta(hours=5)
    mock_schedule1 = MagicMock()
    mock_schedule1.get_elements.return_value = [mock_element1]
    mock_schedule2 = MagicMock()
    mock_schedule2.get_elements.return_value = [mock_element2]
    mock_schedule_management = MagicMock()
    mock_element_management = MagicMock()
    with patch.object(self.schedule_management, 'get_instance', return_value=mock_schedule_management), \
        patch.object(self.schedule_management, 'get_schedule', side_effect=lambda x: mock_schedule1 if x == 'sc
        patch.object(self.element_management, 'get_instance', return_value=mock_element_management), \
        patch.object(self.element_management, 'get_element', side_effect=lambda x: mock_element1 if x == 'eleme
        # Act
        result = user.check_disponibility((datetime.now(), datetime.now() + timedelta(hours=2)))
        # Assert
        self.assertTrue(result)

def test_check_disponibility_ignoring_non_event_elements(self):
    """Test that check_disponibility returns True when the checked period conflicts with a non-event element"""
    # Arrange
    user = User("id", "username", "email", ["schedule1", "schedule2"])
    mock_element1 = MagicMock()
    mock_element1.type = 'reminder'
    mock_element1.start_time = datetime.now() + timedelta(hours=1)
    mock_element1.end_time = datetime.now() + timedelta(hours=3)
    mock_element2 = MagicMock()
    mock_element2.type = 'event'
    mock_element2.start_time = datetime.now() + timedelta(hours=4)
    mock_element2.end_time = datetime.now() + timedelta(hours=5)
    mock_schedule1 = MagicMock()
    mock_schedule1.get_elements.return_value = [mock_element1]
    mock_schedule2 = MagicMock()
    mock_schedule2.get_elements.return_value = [mock_element2]
    mock_schedule_management = MagicMock()
    mock_element_management = MagicMock()
    with patch.object(self.schedule_management, 'get_instance', return_value=mock_schedule_management), \
        patch.object(self.schedule_management, 'get_schedule', side_effect=lambda x: mock_schedule1 if x == 'scl
        patch.object(self.element_management, 'get_instance', return_value=mock_element_management), \
        patch.object(self.element_management, 'get_element', side_effect=lambda x: mock_element1 if x == 'elemen
        # Act
        result = user.check_disponibility((datetime.now(), datetime.now() + timedelta(hours=2)))

        # Assert
        self.assertTrue(result)
```

Algumas mudanças foram feitas para os métodos do UserManagement:

```
In [ ]: def test_user_exists_returns_true(self):
            """Test that user_exists returns True when a user with the given id exists"""
            # Arrange
            user_id = 'existing_user_id'
            mock_db_module = MagicMock()
            mock_db_module.select_data.return_value = [{'_id': user_id,
                'username': 'username', 'email': 'email', 'schedules': []}]
            user_management = UserManagement(mock_db_module)

            # Act
            result = user_management.user_exists(user_id)

            # Assert
            self.assertTrue(result)

        def test_user_exists_returns_false(self):
            """Test that user_exists returns False when a user with the given id does not exist"""
            # Arrange
            user_id = 'non_existent_user_id'
            mock_db_module = MagicMock()
            mock_db_module.select_data.return_value = []
            user_management = UserManagement(mock_db_module)

            # Act
            result = user_management.user_exists(user_id)

            # Assert
            self.assertFalse(result)

        def test_update_user(self):
            """Test that update_user calls update_data with the correct arguments"""
            # Arrange
            user_id = 'existing_user_id'
            user_info = {'_id': user_id, 'username': 'username',
                         'email': 'email', 'schedules': [],
                         'hashed_password': None, 'user_preferences': {}}
            user = User(**user_info)
            mock_db_module = MagicMock()
            mock_db_module.select_data.return_value = [user_info]
            user_management = UserManagement(mock_db_module)
            user_management.users[user_id] = user

            # Act
            user_management.update_user(user_id)

            # Assert
            mock_db_module.update_data.assert_called_once_with('users', {"_id": user_id}, user_info)


        def test_update_nonexistent_user(self):
            """Test that update_user raises NonExistentIDError when the user does not exist"""
            # Arrange
            user_id = 'non_existent_user_id'
            mock_db_module = MagicMock()
            mock_db_module.select_data.return_value = []
            user_management = UserManagement(mock_db_module)
```

```python
        # Act and Assert
        with self.assertRaises(NonExistentIDError):
            user_management.update_user(user_id)

    def test_delete_user(self):
        """Test that delete_user calls delete_data with the correct arguments"""
        # Arrange
        user_id = 'existing_user_id'
        user_info = {'_id': user_id, 'username': 'username', 'email': 'email', 'schedules': []}
        user = User(**user_info)
        mock_db_module = MagicMock()
        mock_db_module.select_data.return_value = [user_info]
        user_management = UserManagement(mock_db_module)
        user_management.users[user_id] = user
        # Act
        user_management.delete_user(user_id)
        # Assert
        mock_db_module.delete_data.assert_called_once_with('users', {"_id": user_id})

    def test_delete_nonexistent_user(self):
        """Test that delete_user raises NonExistentIDError when the user does not exist"""
        # Arrange
        user_id = 'non_existent_user_id'
        mock_db_module = MagicMock()
        mock_db_module.select_data.return_value = []
        user_management = UserManagement(mock_db_module)

        # Act and Assert
        with self.assertRaises(NonExistentIDError):
            user_management.delete_user(user_id)

    def test_create_user_success(self):
        """Test that create_user calls insert_data with the correct arguments"""
        # Arrange
        username = 'username'
        email = 'email@example.com'
        password = 'password'
        user_preferences = {'preference': 'value'}
        user_id = 'new_user_id'
        hashed_password = 'hashed_password'
        mock_db_module = MagicMock()
        mock_db_module.select_data.return_value = []
        user_management = UserManagement(mock_db_module)
        user_management.hash_password = MagicMock(return_value=hashed_password.encode('utf-8'))
        expected_user_info = {"_id": user_id,
                              "username": username,
                              "email": email,
                              "schedules": [],
                              "hashed_password": hashed_password,
                              "user_preferences": user_preferences}
        # Act
        user_management.create_user(username, email, password, user_preferences, user_id)
        # Assert
        mock_db_module.insert_data.assert_called_once_with('users', expected_user_info)

    def test_create_existing_user(self):
        """Test that create_user raises DuplicatedIDError when the user id already exists"""
        # Arrange
        username = 'username'
        email = 'email@example.com'
        password = 'password'
        user_preferences = {'preference': 'value'}
        user_id = 'existing_user_id'
        mock_db_module = MagicMock()
        mock_db_module.select_data.return_value = [{'_id': user_id}]
        user_management = UserManagement(mock_db_module)

        # Act and Assert
        with self.assertRaises(DuplicatedIDError):
            user_management.create_user(username, email, password, user_preferences, user_id)

    def test_create_blank_username(self):
        """Test that create_user raises UsernameCantBeBlank when the username is blank"""
        # Arrange
        username = ''
        email = 'email@example.com'
        password = 'password'
        user_preferences = {'preference': 'value'}
        user_id = 'new_user_id'
        mock_db_module = MagicMock()
        user_management = UserManagement(mock_db_module)
        # Act and Assert
        with self.assertRaises(UsernameCantBeBlank):
```

```
        user_management.create_user(username, email, password, user_preferences, user_id)
```

O teste `test_add_schedule_to_user` foi quebrado em dois testes:

In [ ]:
```
def test_add_schedule_to_user_updates_user_schedules(self):
    """Test that add_schedule_to_user updates the user's schedules"""
    # Arrange
    user_id = 'existing_user_id'
    schedule_id = 'new_schedule_id'
    permission = 'read'
    mock_schedule = MagicMock()
    self.user_management.users[user_id] = User(user_id, 'username', 'email', [], {})
    user_info = {'_id': user_id, 'username': 'username', 'email': 'email', 'schedules': []}
    self.user_management.db_module.select_data.return_value = [user_info]  # Return a list so it can be subscrip
    with patch.object(ScheduleManagement, 'get_schedule', return_value=mock_schedule), \
        patch.object(UserManagement, 'user_exists', return_value=True):
        # Act
        self.user_management.add_schedule_to_user(user_id, schedule_id, permission)
        # Assert
        self.assertIn(schedule_id, self.user_management.users[user_id].schedules)

def test_add_schedule_to_user_updates_schedules_permissions(self):
    """Test that add_schedule_to_user updates the schedule's permissions"""
    # Arrange
    user_id = 'existing_user_id'
    schedule_id = 'new_schedule_id'
    permission = 'read'
    mock_schedule = MagicMock()
    self.user_management.users[user_id] = User(user_id, 'username', 'email', [], {})
    user_info = {'_id': user_id, 'username': 'username', 'email': 'email', 'schedules': []}
    self.user_management.db_module.select_data.return_value = [user_info]  # Return a list so it can be subscrip
    with patch.object(ScheduleManagement, 'get_schedule', return_value=mock_schedule), \
        patch.object(UserManagement, 'user_exists', return_value=True):
        # Act
        self.user_management.add_schedule_to_user(user_id, schedule_id, permission)
        # Assert
        self.assertEqual(mock_schedule.permissions[user_id], permission)
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js