In [ ]:
```python
from ..database.database_module import DatabaseModule

class Authentication:

    def __init__(self, database_module: DatabaseModule):
        ...

    def authenticate_user(self, username: str, password: str) -> bool:
        ...

    def verify_password(self, input_password: str, hashed_password: str) -> bool:
        ...
```

Codigo seguindo o padrão do diagrama (atualizado, removendo o loggout) criado, criando teste para autenticação com sucesso do usuário.

In [ ]:
```python
def test_authenticate_user_success(self):
        # Mocking user_exists and get_user methods
        self.auth_module.user_management_module.user_exists = MagicMock(return_value=True)
        self.auth_module.user_management_module.get_user = MagicMock(return_value=self.create_mock_user())

        # Mocking verify_password method
        self.auth_module.verify_password = MagicMock(return_value=True)

        # Test
        result = self.auth_module.authenticate_user("test_user", "test_password")
        self.assertTrue(result)
```

Não passa no teste de autenticação com sucesso (não está implementado), autenticação implementada:

In [ ]:
```python
from ..database.database_module import DatabaseModule
from ..user.user_management import UserManagement

class AuthenticationModule:

    def __init__(self, database_module: DatabaseModule):
        self.database_module = database_module
        self.user_management_module = UserManagement(self.database_module)

    def authenticate_user(self, username: str, password: str) -> bool:
        user = self.user_management_module.get_user(username)
        return self.verify_password(password, user.get_hashed_password())

    def verify_password(self, input_password: str, hashed_password: str) -> bool:
        ...
```

Entretanto é necessário que a verificação falhe quando a senha está incorreta, e também devemos verificar a implementação do metodo de verificação.

In [ ]:
```python
def test_authenticate_user_wrong_password(self):
        # Mocking user_exists and get_user methods
        self.auth_module.user_management_module.user_exists = MagicMock(return_value=True)
        self.auth_module.user_management_module.get_user = MagicMock(return_value=self.create_mock_user())

        # Mocking verify_password method
        self.auth_module.verify_password = MagicMock(return_value=False)

        # Test
        result = self.auth_module.authenticate_user("test_user", "wrong_password")
        self.assertFalse(result)

    def test_password_verification_success(self):
        # Mocking bcrypt.checkpw method
        bcrypt.checkpw = MagicMock(return_value=True)

        # Test
        result = self.auth_module.verify_password("test_password", "test_hashed_password")
        self.assertTrue(result)

    def test_password_verification_failure(self):
        # Mocking bcrypt.checkpw method
        bcrypt.checkpw = MagicMock(return_value=False)

        # Test
        result = self.auth_module.verify_password("test_password", "test_hashed_password")
        self.assertFalse(result)
```

O código sem a verificação da password, assumindo que ela funciona, já passa no teste de login fracassado caso a senha esteja incorreta, mas falha quando é necessária a verificação da senha segundo o metodo implementado (funciona quando a senha está

errada, pois é sempre None, mas nunca verifica uma senha como correta). A implementação do hashing usando bcrypt é essa:

In [ ]:
```python
from ..database.database_module import DatabaseModule
from ..user.user_management import UserManagement
import bcrypt

class AuthenticationModule:

    def __init__(self, database_module: DatabaseModule):
        self.database_module = database_module
        self.user_management_module = UserManagement(self.database_module)

    def authenticate_user(self, username: str, password: str) -> bool:
        user = self.user_management_module.get_user(username)
        return self.verify_password(password, user.get_hashed_password())

    def verify_password(self, input_password: str, hashed_password: str) -> bool:
        return bcrypt.checkpw(input_password.encode(), hashed_password.encode())
```

O código está funcional, mas devemos pensar no caso em que um usuário não existente tenta fazer login, é necessário que ele receba o feedback de que não está cadastrado, por isso o código precisa lidar com isso:

In [ ]:
```python
def test_authenticate_user_user_not_found(self):
        # Mocking user_exists method
        self.auth_module.user_management_module.user_exists = MagicMock(return_value=False)

        # Test
        with self.assertRaises(UserNotFound):
            self.auth_module.authenticate_user("nonexistent_user", "password")
```

O codigo não é capaz de reagir corretamente quando tentamos autenticar um usuário não existente, para corrigir isso um erro foi criado a devidamente levantado no caso disso ocorrer:

In [ ]:
```python
from ..database.database_module import DatabaseModule
from ..user.user_management import UserManagement
import bcrypt

# create a custom exception
class UserNotFound(Exception):
    pass

class AuthenticationModule:

    def __init__(self, database_module: DatabaseModule):
        self.database_module = database_module
        self.user_management_module = UserManagement(self.database_module)

    def authenticate_user(self, username: str, password: str) -> bool:
        if not self.user_management_module.user_exists(username):
            raise UserNotFound(f"Usuário {username} não encontrado!")
        else:
            user = self.user_management_module.get_user(username)
            user_password = user.get_hashed_password()

            if self.verify_password(password, user_password):
                return True
            else:
                return False

    def verify_password(self, input_password: str, hashed_password: str) -> bool:
        return bcrypt.checkpw(input_password.encode(), hashed_password.encode())
```