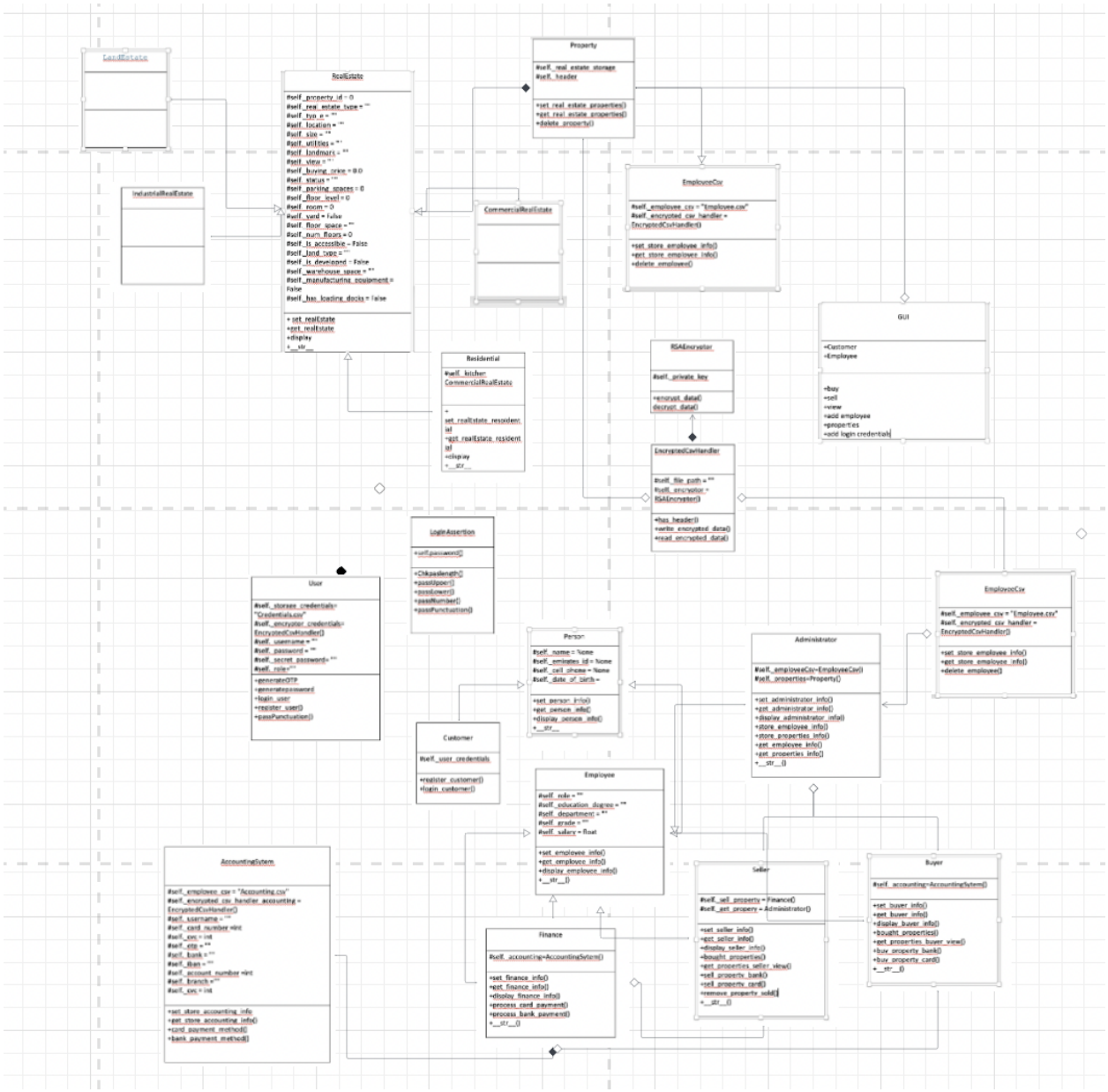
Assignment 3

UML Class Diagram



UML Desription

Use Case 1: Search for Real Estate

Actors:

- Buyer

Description:

Precondition:
- The buyer has successfully logged into the system.

Main Flow:
- The buyer enters search criteria, such as location, price range, and property type.
- The system retrieves and displays a list of real estate properties that match the specified criteria.
- The buyer reviews property details and selects a property of interest.

Alternative Flow:
- If no properties match the search criteria, the system informs the buyer and allows them to modify the search.

Postcondition:
- The buyer has identified a real estate property of interest.

Use Case 2: View Property Details

Actors:

- Buyer
- Seller
- Administrator

Description:

Precondition:
- The user (buyer, seller, or administrator) is logged into the system.

Main Flow:
- The user selects a specific real estate property from the system.

- The system displays detailed information about the selected property, including address, price, area, and description.

Alternative Flow:

- If the property is not found or an error occurs, the system provides an appropriate error message.

Postcondition:

- The user has obtained detailed information about the selected real estate property.

Use Case 3: Manage Employee Information

Actors:

- Administrator

Description:

Precondition:

- The administrator is logged into the system.

Main Flow:

- The administrator accesses the employee management section.
- The system presents options to add, edit, or delete employee information.
- The administrator performs the desired action (add, edit, or delete) on employee information.

Alternative Flow:

- If an error occurs during the process, the system provides an appropriate error message.

Postcondition:

- Employee information is updated according to the administrator's actions.

Use Case 4: List Properties for Sale

Actors:

- Seller

Description:

Precondition:

- The seller is logged into the system.

Main Flow:

- The seller accesses the property management section.
- The system displays a list of properties owned by the seller.
- The seller selects a property and chooses to list it for sale.

Alternative Flow:

- If the seller encounters an issue or decides not to list a property, they can cancel the operation.

Postcondition:

- The selected property is listed for sale in the system.

Use Case 5: Make an Offer

Actors:

- Buyer
- Seller

Description:

Precondition:

- The buyer has identified a property of interest.

Main Flow:

- The buyer selects a property and initiates the process of making an offer.
- The system prompts the buyer to enter details of the offer, including the proposed purchase price and any additional conditions.
- The offer is submitted to the seller.

Alternative Flow:

- If the seller rejects the offer, the system notifies the buyer.
- If the seller accepts the offer, the system proceeds to the next steps in the purchase process.

Postcondition:

- The buyer has submitted an offer, and the seller has either accepted or rejected it.

Use Case 6: Manage Real Estate Listings

Actors:

- Administrator
- Seller

Description:

Precondition:
- The administrator or seller is logged into the system.

Main Flow:
- The administrator/seller accesses the real estate management section.
- The system provides options to add, edit, or remove real estate listings.
- The administrator/seller performs the desired action on real estate listings.

Alternative Flow:
- If an error occurs or if the administrator/seller decides not to proceed, they can cancel the operation.

Postcondition:
- Real estate listings are updated according to the actions of the administrator/seller.

Use Case 7: Process Real Estate Sale

Actors:

- Buyer
- Seller
- Administrator

Description:

Precondition:
- The buyer and seller have agreed on the terms of the sale.

Main Flow:
- The buyer and seller, with the assistance of the administrator, finalize the details of the sale.
- The system generates a sales agreement and relevant documentation.
- The buyer completes the payment, and the system updates the property ownership records.

Alternative Flow:

- If there are issues during the payment process or document generation, the system provides appropriate error messages.

Postcondition:

- The real estate property is transferred to the buyer, and the sale is recorded in the system.

Use Case 8: User Authentication

Actors:

- User

Description:

Precondition:

- The user attempts to access the system.

Main Flow:

- The system prompts the user to enter their username and password.
- The user submits the credentials.
- The system verifies the credentials and grants access if they are valid.

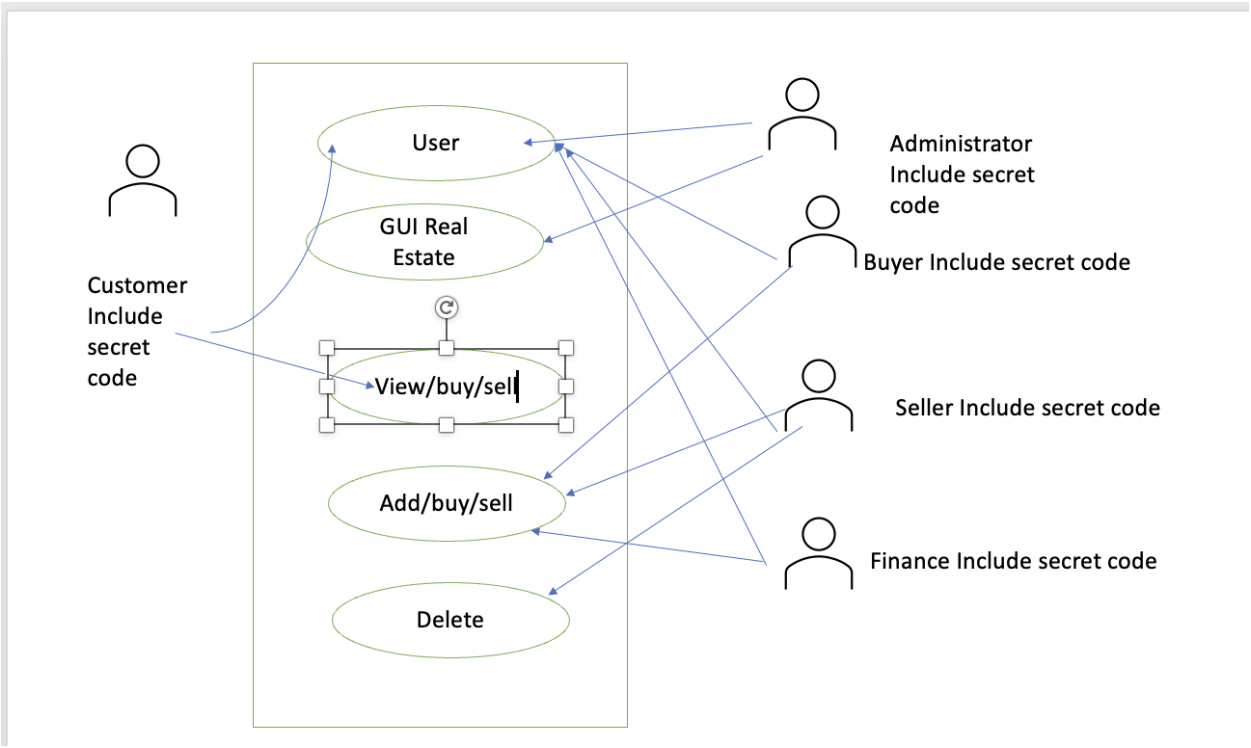
Alternative Flow:

- If the credentials are invalid, the system denies access and provides an appropriate error message.
- If the user forgets their password, the system provides a password reset option.

Postcondition:

- The user is either granted access to the system or denied access based on the provided credentials.

UMLS




```
        """
import csv
import os
import pandas as pd

import unittest
from unittest.mock import patch
from io import StringIO

from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization, hashes
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives.asymmetric import padding
import pickle




class Person:
    """
```

```
    Represents a general person with basic information.
    """

    def __init__(self):
        self._name = None
        self._emirates_id = None
        self._cell_phone = None
        self._date_of_birth = None

    def set_person_info(self, fullname, emirates_id, phone, date_of_birth):
        """
        Set the information of a person.

        Args:
            fullname (str): The full name of the person.
            emirates_id (str): The Emirates ID of the person.
            phone (str): The cell phone number of the person.
            date_of_birth (str): The date of birth of the person.
        """
        self._name = fullname
        self._emirates_id = emirates_id
        self._cell_phone = phone
        self._date_of_birth = date_of_birth if date_of_birth is not None
else None

    def get_person_info(self):
        """
        Get the information of a person.

        Returns:
            tuple: A tuple containing the person's information (name,
Emirates ID, cell phone, date of birth).
        """
        return self._name, self._emirates_id, self._cell_phone,
self._date_of_birth

    def display_person_info(self):
        """
        Generate a formatted string representing the person's information.
```

```
        Returns:
            str: A formatted string containing the person's information.
        """
        return f'Full Name: {self._name}, Emirates ID: {self._emirates_id},
Phone: {self._cell_phone}, Date Of Birth: {self._date_of_birth}'

    def __str__(self):
        """
        Generate a string representation of a person.

        Returns:
            str: A string containing the person's information.
        """
        return f'Full Name: {str(self._name)}, ID:
{str(self._emirates_id)}, Phone: {str(self._cell_phone)}, Date Of Birth:
{str(self._date_of_birth)}'









class RealEstate:
    """
    Class representing different types of real estate properties.
    """

    def __init__(self):
        """
        Constructor for the RealEstate class.
        """
        self._property_id = 0
        self._real_estate_type = ""
```

```
        self._typ_e = ""
        self._location = ""
        self._size = ""
        self._utilities = ""
        self._landmark = ""
        self._view = ""
        self._buying_price = 0.0
        self._status = ""
        self._parking_spaces = 0
        self._floor_level = 0
        self._room = 0
        self._yard = False
        self._floor_space = ""
        self._num_floors = 0
        self._is_accessible = False
        self._land_type = ""
        self._is_developed = False
        self._warehouse_space = ""
        self._manufacturing_equipment = False
        self._has_loading_docks = False

    def  set_realEstate(self, property_id, real_estate_type, typ_e,
location, size, utilities, landmark, view, buying_price, status,
                        parking_spaces, floor_level, room, yard,
floor_space, num_floors, is_accessible, land_type, is_developed,
                        warehouse_space, manufacturing_equipment,
has_loading_docks):
        """
        Set the attributes of the RealEstate object.

        Parameters:
        - property_id (int): Property ID.
        - real_estate_type (str): Type of real estate.
        - typ_e (str): Another type (you might want to clarify this
attribute).
        - location (str): Location of the real estate.
        - size (str): Measurement of the real estate.
        - utilities (str): Utilities available.
        - landmark (str): Landmark near the real estate.
        - view (str): View from the real estate.
```

```
        - buying_price (float): Buying price of the real estate.
        - status (str): Status of the real estate.
        - parking_spaces (int): Number of parking spaces.
        - floor_level (int): Floor level of the real estate.
        - room (int): Number of rooms (for residential properties).
        - yard (bool): Indicates whether the property has a yard (for
residential properties).
        - floor_space (str): Floor space type (for commercial properties).
        - num_floors (int): Number of floors (for commercial properties).
        - is_accessible (bool): Indicates whether the property is
accessible (for commercial properties).
        - land_type (str): Type of land (for land properties).
        - is_developed (bool): Indicates whether the land is developed (for
land properties).
        - warehouse_space (str): Type of warehouse space (for industrial
properties).
        - manufacturing_equipment (bool): Indicates whether manufacturing
equipment is present (for industrial properties).
        - has_loading_docks (bool): Indicates whether the property has
loading docks (for industrial properties).
        """
        self._property_id = property_id
        self._real_estate_type = real_estate_type
        self._typ_e = typ_e
        self._location = location
        self._size = size
        self._utilities = utilities
        self._landmark = landmark
        self._view = view
        self._buying_price = buying_price
        self._status = status
        self._parking_spaces = parking_spaces
        self._floor_level = floor_level
        self._room = room
        self._yard = yard
        self._floor_space = floor_space
        self._num_floors = num_floors
        self._is_accessible = is_accessible
        self._land_type = land_type
        self._is_developed = is_developed
```

```
        self._warehouse_space = warehouse_space
        self._manufacturing_equipment = manufacturing_equipment
        self._has_loading_docks = has_loading_docks




    def get_realEstate(self):
        """
        Get a tuple containing the values of all attributes of the
RealEstate object.

        Returns:
        tuple: A tuple containing the values of all attributes.
        """
        return self._property_id, self._real_estate_type, self._typ_e,
self._location, self._size, self._utilities, self._landmark, self._view,
self._buying_price, self._status, self._parking_spaces, self._floor_level,
self._room, self._yard, self._floor_space,self._num_floors,
self._is_accessible, self._land_type,
self._is_developed,self._warehouse_space, self._manufacturing_equipment,
self._has_loading_docks

    def display_real_estate(self):
        """
        Display information about the RealEstate object.
        """
        for attr_name in dir(self):
            if attr_name.startswith("_") and attr_name not in [
                '_property_id', '_real_estate_type', '_typ_e', '_location',
'_size', '_utilities', '_landmark',
                '_view', '_buying_price', '_status', '_parking_spaces',
'_floor_level'
            ]:
                print(f"{attr_name[1:]}:", getattr(self, attr_name))

    def __str__(self):
        """
        Return a string representation of the RealEstate object.
```

```
        Returns:
        str: String representation of the RealEstate object.
        """
        real_estate_info = f"RealEstate(Property ID: {self._property_id},
Real Estate Type: {self._real_estate_type}, Type: {self._typ_e}, Location:
{self._location}, Size: {self._size}, Utilities: {self._utilities},
Landmark: {self._landmark}, View: {self._view}, Buying Price:
{self._buying_price}, Status: {self._status}, Parking Spaces:
{self._parking_spaces}, Floor Level: {self._floor_level}"

        for attr_name in dir(self):
            if attr_name.startswith("_") and attr_name not in [
                '_property_id', '_real_estate_type', '_typ_e', '_location',
'_size', '_utilities', '_landmark',
                '_view', '_buying_price', '_status', '_parking_spaces',
'_floor_level'
            ]:
                real_estate_info += f", {attr_name[1:]}: {getattr(self,
attr_name)}"

        return real_estate_info




class ResidentialRealEstate(RealEstate):
    """
    Class representing residential real estate properties.
    Inherits from the RealEstate class.
    """

    def __init__(self):
        """
        Constructor for the ResidentialRealEstate class.
        """
        super().__init__()  # Call the constructor of the base class
RealEstate
```

```
        self._kitchen = ""

    def set_residential_realEstate(self, property_id, real_estate_type,
typ_e, location, size, utilities, landmark, view, buying_price, status,
                                   parking_spaces, floor_level, room,
kitchen, yard, floor_space, num_floors, is_accessible, land_type,
is_developed,
                                   warehouse_space,
manufacturing_equipment, has_loading_docks):
        """
        Set attributes specific to residential real estate.

        Parameters:
        - All parameters are the same as the set_realEstate method in the
base class, with an additional 'kitchen' parameter.
        """
        super().set_realEstate(property_id, real_estate_type, typ_e,
location, size, utilities, landmark, view, buying_price, status,
                               parking_spaces, floor_level, room, yard,
floor_space, num_floors, is_accessible, land_type, is_developed,
                               warehouse_space, manufacturing_equipment,
has_loading_docks)
        self._kitchen = kitchen

    def get_kitchen(self):
        """
        Get the type of kitchen.

        Returns:
        str: Type of kitchen.
        """
        return super().get_realEstate(),self._kitchen

    def display_residential_real_estate(self):
        """
        Display information about the ResidentialRealEstate object.
        """
        super().display_real_estate()  # Call the display_real_estate
method of the base class
        print(f"Kitchen: {self._kitchen}")
```

```
    def __str__(self):
        """
        Return a string representation of the ResidentialRealEstate object.

        Returns:
        str: String representation of the ResidentialRealEstate object.
        """
        residential_info = super().__str__()  # Get the string
representation from the base class
        residential_info += f", Kitchen: {self._kitchen}"
        return residential_info



class CommercialRealEstate(RealEstate):
    """
    Class representing commercial real estate properties.
    """
    pass

class IndustrialRealEstate(RealEstate):
    """
    Class representing Industrial RealEstate.
    """
    pass

class LandEstate(RealEstate):
    """
    Class representing land estates.
    """
    pass
```

```
class RSAEncryptor:
    def __init__(self):
        """
        Initialize the RSAEncryptor.

        This method generates an RSA key pair (public and private key)
during object creation.
        """
        # Generate RSA key pair
        self._private_key = rsa.generate_private_key(
            public_exponent=65537,
            key_size=10000,  # Adjust key size based on security
requirements
            backend=default_backend()
        )
        self.public_key = self.private_key.public_key()

    def encrypt_data(self, data):
        """
        Encrypt data using RSA public key.

        Args:
            data: The data to be encrypted.

        Returns:
            bytes: The encrypted data.
        """
        # Serialize and encrypt the data using RSA
        serialized_data = pickle.dumps(data)
        ciphertext = self._public_key.encrypt(
            serialized_data,
            padding.OAEP(
                mgf=padding.MGF1(algorithm=hashes.SHA256()),
                algorithm=hashes.SHA256(),
                label=None
            )
        )
        return ciphertext

    def decrypt_data(self, encrypted_data):
```

```
        """
        Decrypt encrypted data using RSA private key.

        Args:
            encrypted_data: The encrypted data to be decrypted.

        Returns:
            Any: The decrypted data.
        """
        # Decrypt and deserialize the data using RSA
        decrypted_data = self._private_key.decrypt(
            encrypted_data,
            padding.OAEP(
                mgf=padding.MGF1(algorithm=hashes.SHA256()),
                algorithm=hashes.SHA256(),
                label=None
            )
        )
        return pickle.loads(decrypted_data)

class EncryptedCsvHandler:
    def __init__(self):
        """
        Initialize the EncryptedCsvHandler.

        Args:
            file_path (str): The path to the encrypted CSV file.
            encryptor (RSAEncryptor): The RSAEncryptor object used for
encryption and decryption.
        """
        self._file_path = ""
        self._encryptor = RSAEncryptor()


    def has_header(self, file_path):
        if not os.path.exists(file_path):
            return False

        with open(self._file_path, 'rb') as file:
            encrypted_data = file.read()
```

```
            decrypted_data = self.encryptor.decrypt_data(encrypted_data)
            try:
                # Assuming the first row is the header
                header = next(decrypted_data)
                return True
            except StopIteration:
                return False

    def write_encrypted_data(self,file_path, data,header=None):
        """
        Write encrypted data to the encrypted CSV file.

        Args:
            data: The data to be encrypted and written to the CSV file.
        """
        file_has_header = self.has_header(file_path)
        encrypted_data = self._encryptor.encrypt_data(data)

        with open(file_path, 'ab') as writer:
            writer.write(encrypted_data)

            if not file_has_header and header:
                writer.writerow(header)

            # Write the data
            writer.writerow(data)
            writer.close()


    def read_encrypted_data(self,file_path):
        """
        Read encrypted data from the encrypted CSV file.

        Returns:
            Any: The decrypted data.
        """
        self.file_path = file_path
        with open(self._file_path, 'rb') as file:
            encrypted_data = file.read()
```

```
        decrypted_data = self._encryptor.decrypt_data(encrypted_data)
        file.close()

        return decrypted_data



class TestEncryption(unittest.TestCase):

    def setUp(self):
        self.file_path = 'test_encrypted.csv'
        self.handler = rsa.EncryptedCsvHandler(self.file_path)

    def tearDown(self):
        # Clean up the test file after each test
        if os.path.exists(self.file_path):
            os.remove(self.file_path)

    def test_encryption_decryption(self):
        data = {'username': 'test_user', 'password': 'test_password'}
        self.handler.write_encrypted_data(self.file_path, data)

        decrypted_data = self.handler.read_encrypted_data()

        self.assertEqual(decrypted_data, data)

    def test_has_header(self):
        # Test if the handler correctly detects the presence of a header in
the file
        self.assertFalse(self.handler.has_header(self.file_path))

        # Writing data with a header
        data_with_header = {'name': 'John', 'age': 30}
        header = ['name', 'age']

        self.handler.write_encrypted_data(self.file_path, data_with_header,
header=header)

        # Check if the handler detects the header after writing
        self.assertTrue(self.handler.has_header(self.file_path))
```

```
    def test_encryption_decryption_multiple_entries(self):
        # Test handling multiple entries
        data1 = {'username': 'user1', 'password': 'pass1'}
        data2 = {'username': 'user2', 'password': 'pass2'}

        self.handler.write_encrypted_data(self.file_path, data1)
        self.handler.write_encrypted_data(self.file_path, data2)

        decrypted_data = self.handler.read_encrypted_data()

        # The decrypted data should be a list containing both entries
        self.assertEqual(decrypted_data, [data1, data2])




class Property:
    """Class for properties in the Real Estate"""

    def __init__(self):
        # File path to store real estate properties
        self._real_estate_storage = "real_estate_storage.csv"

        # Header for the CSV file
        self._header = ("Property ID", "Real Estate Type", "Building Type",
"Location", "Measurement", "Utilities", "Landmark",
                        "View", "Parking Space", "Floor Level", "Number of
Rooms", "Kitchen", "Yard", "Floor Shape",
                        "Number of Floors", "Is Accessible", "Land Type",
"Is Developed", "Warehouse Space",
                        "Manufacturing Equipment", "Loading Docks", "Buying
Price", "Status")

    def set_real_estate_properties(self, property_id, real_estate_type,
building_type, location, measurement, utilities, landmark, view,
buying_price, status,
```

```
                                    parking_spaces, floor_level, num_rooms,
kitchen, yard, floor_shape, num_floors, is_accessible, land_type,
is_developed,
                                    warehouse_space,
manufacturing_equipment, loading_docks):
        """
        Set real estate properties and write them to the CSV file.

        Args:
            property_id (str): Property ID.
            real_estate_type (str): Real Estate Type.
            building_type (str): Building Type.
            location (str): Location.
            measurement (str): Measurement.
            utilities (str): Utilities.
            landmark (str): Landmark.
            view (str): View.
            buying_price (float): Buying Price.
            status (str): Status.
            parking_spaces (int): Parking Spaces.
            floor_level (int): Floor Level.
            num_rooms (int): Number of Rooms.
            kitchen (str): Kitchen.
            yard (str): Yard.
            floor_shape (str): Floor Shape.
            num_floors (int): Number of Floors.
            is_accessible (bool): Is Accessible.
            land_type (str): Land Type.
            is_developed (bool): Is Developed.
            warehouse_space (str): Warehouse Space.
            manufacturing_equipment (str): Manufacturing Equipment.
            loading_docks (bool): Loading Docks.
        """
        data = (property_id, real_estate_type, building_type, location,
measurement, utilities, landmark, view, buying_price, status,
                parking_spaces, floor_level, num_rooms, kitchen, yard,
floor_shape, num_floors, is_accessible, land_type, is_developed,
                warehouse_space, manufacturing_equipment, loading_docks)

        file_path = self._real_estate_storage
```

```
        header = self._header

        # Use EncryptedCsvHandler to write data to the CSV file
        EncryptedCsvHandler().write_encrypted_data(file_path, data, header)

    def get_real_estate_properties(self):
        """
        Get real estate properties from the CSV file.

        Returns:
            list: List of real estate properties.
        """
        file_path = self._real_estate_storage

        # Use EncryptedCsvHandler to read data from the CSV file
        data = EncryptedCsvHandler().read_encrypted_data(file_path)
        return data

    def delete_property(self, property_id_to_delete):
        """
        Method to delete a specific property from the CSV file.

        Args:
            property_id_to_delete (str): Property ID to be deleted.
        """
        file_path = self._real_estate_storage
        properties = EncryptedCsvHandler().read_encrypted_data(file_path)

        # Find the index of the property with the specified Property ID
        property_index_to_delete = -1
        for i, property_data in enumerate(properties):
            if property_data[0] == property_id_to_delete:
                property_index_to_delete = i
                break

        # If the property is found, delete it
        if property_index_to_delete != -1:
            del properties[property_index_to_delete]

            # Write the updated data back to the CSV file
```

```
            header = self._header
            EncryptedCsvHandler().write_encrypted_data(file_path,
properties, header)
            print(f"Property with ID {property_id_to_delete} deleted
successfully.")
        else:
            print(f"Property with ID {property_id_to_delete} not found.")




class LoginAssertion:
    """Class to capture login details"""

    def __init__(self, password):
        """
        Initialize Login object with a username and password.

        Parameters:

        - password (str): The password for the login.
        """
        self.password = password

    def chkpaslength(self):
        """
        Check if the password length is at least 8 characters.

        Returns:
        - bool: True if the password length is at least 8, False otherwise.
        """
        return len(self.password) > 8

    def passUpper(self):
        """
        Check if the password contains at least one uppercase letter.

        Returns:
```

```
        - bool: True if the password contains at least one uppercase
letter, False otherwise.
        """
        return any(x.isupper() for x in self.password)

    def passLower(self):
        """
        Check if the password contains at least one lowercase letter.

        Returns:
        - bool: True if the password contains at least one lowercase
letter, False otherwise.
        """
        return any(x.islower() for x in self.password)

    def passNumber(self):
        """
        Check if the password contains at least one digit.

        Returns:
        - bool: True if the password contains at least one digit, False
otherwise.
        """
        return any(x.isdigit() for x in self.password)

    def passPunctuation(self):
        """
        Check if the password contains at least one special character.

        Returns:
        - bool: True if the password contains at least one special
character, False otherwise.
        """
        return any(x in string.punctuation for x in self.password)

class User():
    def __init__(self):
        """
        Initialize a User object with a username, password, and email.
```

```
        Parameters:
        - username (str): The username of the user.
        - password (str): The password of the user.
        - email (str): The email address of the user.
        """
        self._storage_credentials= "Credentials.csv"
        self._encryptor_credentials= EncryptedCsvHandler()
        self._username = ""
        self._password = ""
        self._secret_password= ""
        self._role=""

# function to generate OTP
    def generateOTP(self) :
        import math, random
        import string

        # Declare a digits variable
        # which stores all digits
        all_characters = string.ascii_letters + string.digits +
string.punctuation

        # Generate a random string

        # Print the result

        OTP = ""

      # length of password can be changed
      # by changing value in range
        for i in range(3) :
            OTP += ''.join(random.choice(all_characters) for i in range(2))

        return OTP

    def generatepassword(self) :
        import math, random
        import string

        # Declare a digits variable
```

```
        # which stores all digits
        all_characters = string.ascii_letters + string.digits +
string.punctuation

        # Generate a random string


        # Print the result

        password = ""

      # length of password can be changed
      # by changing value in range
        for i in range(5) :
            OTP += ''.join(random.choice(all_characters) for i in range(3))

        return password

    def
login_user(self,entered_role,entered_username,entered_password,entered_sec
ret_password):
        entered_role =  self._role
        entered_username = self._username
        entered_password = self._password
        entered_secret_password = self._secret_password


          # Read encrypted credentials from file
        encrypted_data = self._storage_credentials.read_encrypted_data()

          # Decrypt credentials
        decrypted_data =
self._encryptor_credentials.decrypt_data(encrypted_data)

          # Check if entered credentials match stored credentials
        for user_data in decrypted_data:
              stored_role, stored_username, stored_password,
stored_secret_code = user_data
```

```
            if stored_role== entered_role and entered_username ==
stored_username and entered_password == stored_password and
entered_secret_password==stored_secret_code:
                  print("Login successful!")
                  return "login successful"
            else:
                  print("Login failed. Please check your credentials.")
                  return "Login failed"



   def register_user(self, entered_role,entered_username,
entered_password, password_check, entered_secret_code):

     encrypted_data = self._storage_credentials.read_encrypted_data()

         # Decrypt credentials
     decrypted_data =
self._encryptor_credentials.decrypt_data(encrypted_data)
     for user_data in decrypted_data:
            stored_role, stored_username, stored_password,
stored_secret_code = user_data
            if entered_username == stored_username or entered_password ==
stored_password or entered_secret_code==stored_secret_code:
              print(f"Credentials Exist")
            else:
                max_attempts = 3

                for attempt in range(1, max_attempts + 1):
                    try:
                        # Check if entered password and re-entered
password match
                        if entered_password == password_check:
                            # Encrypt new user credentials

                            new_user_data =entered_role,
entered_username, entered_password, entered_secret_code
                            header = "Role","Username","Password","Secret
Code"
```

```
                                encrypted_data =
self._encryptor_credentials.write_encrypted_data(self._storage_credentials
,new_user_data,header)

                                # Write encrypted credentials to file

self._storage_credentials.write_encrypted_data()

                                print("Registration successful!")
                                break  # Exit the loop on successful
registration
                            else:
                                raise ValueError("Passwords don't match.
Please try again.")
                        except Exception as e:
                            print(f"Error during registration attempt
{attempt}: {str(e)}")

                    else:
                        print("Maximum registration attempts reached. Please
try again later.")

class TestUserClass(unittest.TestCase):

   def test_generate_otp(self):
       user = User()
       otp = user.generateOTP()
       self.assertEqual(len(otp), 6)

   def test_generate_password(self):
       user = User()
       password = user.generatepassword()
       self.assertEqual(len(password), 12)
       # Assuming the following methods are present in your LoginAssertion
class
       self.assertTrue(password.chkpaslength(), 'The length is too
short.')
       self.assertTrue(password.passUpper(), 'Password must have an
uppercase letter.')
```

```
        self.assertTrue(password.passLower(), 'Password must have a
lowercase letter.')
        self.assertTrue(password.passNumber(), 'Password must have a
number.')
        self.assertTrue(password.passPunctuation(), 'Password must have a
special character.')

    @patch('builtins.input', side_effect=['testuser', 'testpassword',
'testpassword'])
    @patch('builtins.open', create=True)
    def test_register_system_valid_password(self, mock_open, mock_input):
        user = User()
        user.register_user()
        mock_open.assert_called_with("register.pkl", "wb")

    @patch('builtins.input', side_effect=['testuser', 'short', 'short'])
    def test_register_system_invalid_password(self, mock_input):
        user = User()
        with self.assertRaises(SystemExit) as cm:
            user.register_user()
        self.assertEqual(cm.exception.code, 1)

class Customer(Person):
    def __init__(self):
        # Call the constructor of the base class (Person)
        super().__init__()
        # Initialize user credentials using the User class
        self._user_credentials = User()

    def register_customer(self, entered_username, entered_password,
check_password, entered_secret_password):
        """
        Register a customer with the provided credentials.

        Args:
            entered_username (str): Entered username for registration.
            entered_password (str): Entered password for registration.
            check_password (str): Re-entered password for confirmation.
            entered_secret_password (str): Entered secret password for
registration.
```

```
        Returns:
            str: Registration success message or error message.
        """
        return self._user_credentials.register_user(entered_username,
entered_password, check_password, entered_secret_password)

    def login_customer(self, entered_username, entered_password,
entered_secret_password):
        """
        Log in a customer with the provided credentials.

        Args:
            entered_username (str): Entered username for login.
            entered_password (str): Entered password for login.
            entered_secret_password (str): Entered secret password for
login.

        Returns:
            str: Login success message or error message.
        """
        # Perform login using user credentials
        login_status = self._user_credentials.login_user(entered_username,
entered_password, entered_secret_password)

        # Convert the login status to lowercase for consistency
        login_status_lower = login_status.lower()

        if login_status_lower == "login successful":
            # Return None if login is successful
            return "login successful"
        else:
            # Return error message if login is unsuccessful
            return "Login Unsuccessful"


class EmployeeCsv:
    def __init__(self):
        """
        Initialize the EmployeeCsv handler.
```

```
        This class handles the storage and retrieval of employee
information in an encrypted CSV file.
        """
        self._employee_csv = "Employee.csv"
        self._encrypted_csv_handler = EncryptedCsvHandler()

    def set_store_employee_info(self, fullname, emirates_id, phone,
date_of_birth, role, education_degree, department, grade, salary):
        """
        Set and store employee information in the encrypted CSV file.

        Args:
            fullname (str): Full name of the employee.
            emirates_id (str): Emirates ID of the employee.
            phone (str): Phone number of the employee.
            date_of_birth (str): Date of birth of the employee.
            role (str): Role or position of the employee.
            education_degree (str): Highest education degree of the
employee.
            department (str): Department in which the employee works.
            grade (str): Grade or level of the employee.
            salary (float): Salary of the employee.
        """
        file_path = self._employee_csv
        header = ["Full Name", "Emirates ID", "Phone", "Date of Birth",
"Role", "Education Degree", "Department", "Grade", "Salary"]
        data = [fullname, emirates_id, phone, date_of_birth, role,
education_degree, department, grade, salary]
        self._encrypted_csv_handler.write_encrypted_data(file_path, data,
header)

    def get_store_employee_info(self):
        """
        Retrieve and return stored employee information from the encrypted
CSV file.

        Returns:
            pandas.DataFrame: A DataFrame containing the stored employee
information.
```

```
        """
        read_data = self._encrypted_csv_handler.read_encrypted_data()

        return read_data

    def delete_employee(self, employee_id_to_delete):
        """
        Method to delete a specific employee from the CSV file.

        Args:
            employee_id_to_delete (str): Property ID to be deleted.
        """
        file_path = self._employee_csv
        employee = EncryptedCsvHandler().read_encrypted_data(file_path)

        # Find the index of the property with the specified Property ID
        employee_index_to_delete = -1
        for i, employee_data in enumerate(employee):
            if employee_data[1] == employee_id_to_delete:
                employee_index_to_delete = i
                break

        # If the property is found, delete it
        if employee_index_to_delete != -1:
            del employee[employee_index_to_delete]

            # Write the updated data back to the CSV file
            header = self._header
            EncryptedCsvHandler().write_encrypted_data(file_path, employee,
header)
            print(f"Employee with ID {employee_id_to_delete} deleted
successfully.")
        else:
            print(f"Employee with ID {employee_id_to_delete} not found.")



class Employee(Person):
    """
```

```
   Represents an employee of the hospital with additional department and
room information.
   """

   def __init__(self):
       super().__init__()
       self._role = ""
       self._education_degree = ""
       self._department = ""
       self._grade = ""
       self._salary = float

   def set_employee_info(self, fullname, emirates_id, phone,
date_of_birth, role, education_degree, department, grade,salary):
       """
       Set the information of an employee.

       Args:
           fullname (str): The full name of the employee.
           emirates_id (str): The Emirates ID of the employee.
           phone (str): The cell phone number of the employee.
           date_of_birth (str): The date of birth of the employee.
           role (str): The role or job title of the employee.
           education_degree (str): The highest education degree of the
employee.
           department (str): The department in which the employee works.
           grade (str): The grade or level of the employee.
       """
       super().set_person_info(fullname, emirates_id, phone,
date_of_birth,salary)
       self._role = role
       self._education_degree = education_degree
       self._department = department
       self._grade = grade
       self._salary = salary

   def get_employee_info(self):
       """
       Get the information of an employee.
```

```
        Returns:
            tuple: A tuple containing the employee's information
            (name, Emirates ID, cell phone, date of birth, role, education
degree, department, grade).
        """
        return super().get_person_info() + (self._role,
self._education_degree, self._department, self._grade)

    def display_employee_info(self):
        """
        Generate a formatted string representing the employee's
information.

        Returns:
            str: A formatted string containing the employee's information.
        """
        person_info = super().display_person_info()
        return f'{person_info}, Role: {self._role}, Education Degree:
{self._education_degree}, Department: {self._department}, Grade:
{self._grade}'

    def __str__(self):
        """
        Generate a string representation of an employee.

        Returns:
            str: A string containing the employee's information.
        """
        person_info = super().__str__()
        return f'{person_info}, Role: {str(self._role)}, Education Degree:
{str(self._education_degree)}, Department: {str(self._department)}, Grade:
{str(self._grade)}'



class Administrator(Employee):
    """
    Represents an administrator in the hospital.
    """
```

```
    def __init__(self):
        super().__init__()
        self._employeeCsv=EmployeeCsv()
        self._properties=Property()

    def set_administrator_info(self, fullname, emirates_id, phone,
date_of_birth, education_degree, department, grade):
        """
        Set the information of an administrator.

        Args:
            fullname (str): The full name of the administrator.
            emirates_id (str): The Emirates ID of the administrator.
            phone (str): The cell phone number of the administrator.
            date_of_birth (str): The date of birth of the administrator.
            education_degree (str): The highest education degree of the
administrator.
            department (str): The department in which the administrator
works.
            grade (str): The grade or level of the administrator.

        """
        super().set_employee_info(fullname, emirates_id, phone,
date_of_birth,  education_degree, department, grade)


    def get_administrator_info(self):
        """
        Get the information of an administrator.

        Returns:
            tuple: A tuple containing the administrator's information
            (name, Emirates ID, cell phone, date of birth, role, education
degree, department, grade, responsibility).
        """
        return super().get_employee_info() + (self._responsibility,)

    def display_administrator_info(self):
        """
```

```
        Generate a formatted string representing the administrator's
information.

        Returns:
            str: A formatted string containing the administrator's
information.
        """
        employee_info = super().display_employee_info()
        return f'{employee_info}, Responsibility: {self._responsibility}'

    def store_employee_info(self, fullname, emirates_id, phone,
date_of_birth, role, education_degree, department, grade, salary):
        """
        Store employee information in the encrypted CSV file.

        Args:
            fullname (str): Full name of the employee.
            emirates_id (str): Emirates ID of the employee.
            phone (str): Phone number of the employee.
            date_of_birth (str): Date of birth of the employee.
            role (str): Role or position of the employee.
            education_degree (str): Highest education degree of the
employee.
            department (str): Department in which the employee works.
            grade (str): Grade or level of the employee.
            salary (float): Salary of the employee.
        """
        self._employee_csv.set_store_employee_info(fullname, emirates_id,
phone, date_of_birth, role, education_degree, department, grade, salary)

    def store_properties_info(self, fullname, emirates_id, phone,
date_of_birth, role, education_degree, department, grade, salary):
        """
        Store employee information in the encrypted CSV file.

        Args:
            fullname (str): Full name of the employee.
            emirates_id (str): Emirates ID of the employee.
            phone (str): Phone number of the employee.
            date_of_birth (str): Date of birth of the employee.
```

```
            role (str): Role or position of the employee.
            education_degree (str): Highest education degree of the
employee.
            department (str): Department in which the employee works.
            grade (str): Grade or level of the employee.
            salary (float): Salary of the employee.
        """
        self._employee_csv.set_store_employee_info(fullname, emirates_id,
phone, date_of_birth, role, education_degree, department, grade, salary)



    def get_employee_info(self):
        """
        Retrieve stored employee information from the encrypted CSV file.

        Returns:
            pandas.DataFrame: A DataFrame containing the stored employee
information.
        """
        return self._employee_csv.get_store_employee_info()

    def get_properties_info(self):

      return self._properties().get_real_estate_properties()




    def __str__(self):
        """
        Generate a string representation of an administrator.

        Returns:
            str: A string containing the administrator's information.
        """
        employee_info = super().__str__()
        return f'{employee_info}, Responsibility:
{str(self._responsibility)}'
```

```
class AccountingSytem:
   def __init__(self):
       """
       Initialize the EmployeeCsv handler.

       This class handles the storage and retrieval of employee
information in an encrypted CSV file.
       """
       self._employee_csv = "Accounting.csv"
       self._encrypted_csv_handler_accounting = EncryptedCsvHandler()
       self._username = ""
       self._card_number =int
       self._cvc = int
       self._otp = ""
       self._bank = ""
       self._iban = ""
       self._account_number =int
       self._branch =""
       self._cvc = int


   def set_store_accounting_info(self, fullname, emirates_id,
date,amount,description):
       """
       Set and store employee information in the encrypted CSV file.

       Args:
           fullname (str): Full name of the employee.
           emirates_id (str): Emirates ID of the employee.
           date(None): date of payment
           amount(int):
       """
       file_path = self._employee_csv
       header = ["Full Name", "Emirates ID",
"Date","Amount","Description"]
       data = [fullname, emirates_id, date, amount,description]

self._encrypted_csv_handler_accounting.write_encrypted_data(file_path,
data, header)
```

```
   def get_store_accounting_info(self):
       """
       Retrieve and return stored employee information from the encrypted
CSV file.

       Returns:
           pandas.DataFrame: A DataFrame containing the stored employee
information.
       """
       read_data =
self._encrypted_csv_handler_accounting.read_encrypted_data()
       df = pd.DataFrame(read_data)
       return df

   def card_payment_method(self, fullname, emirates_id, date, amount,
description, card_username, card_number, cvc, otp):
       """
       Simulate a card payment method.

       Args:
           fullname (str): Full name of the payer.
           emirates_id (str): Emirates ID of the payer.
           date (str): Date of the payment.
           amount (float): Amount to be paid.
           description (str): Description of the payment.
           card_username (str): Card username.
           card_number (str): Card number.
           cvc (str): Card Verification Code.
           otp (str): One-Time Password.

       Note: This is a simplified simulation and should not handle real
card information.
       """
       self._card_username = card_username
       self._card_number = card_number
       self._cvc = cvc
       self._otp = otp

       # Additional logic for card payment processing
```

```
        self.set_store_accounting_info(fullname, emirates_id, date, amount,
description)
        print("Card payment processed successfully.")
        print(f"Card Username: {self._card_username}")
        print(f"Card Number: **** **** **** {self._card_number[-4:]}")  #
Mask all but last four digits
        print("CVC: ****")  # Mask CVC for security
        print("OTP: ****")  # Mask OTP for security

    def bank_payment_method(self, fullname, emirates_id, date, amount,
description, bank_details, username, iban, account_number, branch):
        """
        Simulate a bank payment method.

        Args:
            fullname (str): Full name of the payer.
            emirates_id (str): Emirates ID of the payer.
            date (str): Date of the payment.
            amount (float): Amount to be paid.
            description (str): Description of the payment.
            bank_details (str): Bank details.
            username (str): User's bank username.
            iban (str): International Bank Account Number (IBAN).
            account_number (str): Bank account number.
            branch (str): Bank branch.

        Note: This is a simplified simulation and should not handle real
bank information.
        """
        self._bank_details = bank_details
        self._username = username
        self._iban = iban
        self._account_number = account_number
        self._branch = branch

        # Additional logic for bank payment processing
        self.set_store_accounting_info(fullname, emirates_id, date, amount,
description)
        print("Bank payment processed successfully.")
        print(f"Bank Details: {self._bank_details}")
```

```
        print(f"Username: {self._username}")
        print(f"IBAN: {self._iban}")
        print(f"Account Number: {self._account_number}")
        print(f"Branch: {self._branch}")




class Finance(Employee):
    """
    Represents a finance employee in the hospital.
    """

    def __init__(self):
        super().__init__()
        self._accounting=AccountingSytem()


    def set_finance_info(self, fullname, emirates_id, phone, date_of_birth,
education_degree, department, grade):
        """
        Set the information of a finance employee.

        Args:
            fullname (str): The full name of the finance employee.
            emirates_id (str): The Emirates ID of the finance employee.
            phone (str): The cell phone number of the finance employee.
            date_of_birth (str): The date of birth of the finance employee.
            education_degree (str): The highest education degree of the
finance employee.
            department (str): The department in which the finance employee
works.
            grade (str): The grade or level of the finance employee.
            accounting_system (str): The specific accounting system used by
the finance employee.
        """
        super().set_employee_info(fullname, emirates_id, phone,
date_of_birth, role, education_degree, department, grade)
```

```
    def get_finance_info(self):
        """
        Get the information of a finance employee.

        Returns:
            tuple: A tuple containing the finance employee's information
            (name, Emirates ID, cell phone, date of birth, role, education
degree, department, grade, accounting system).
        """
        return super().get_employee_info()

    def display_finance_info(self):
        """
        Generate a formatted string representing the finance employee's
information.

        Returns:
            str: A formatted string containing the finance employee's
information.
        """
        employee_info = super().display_employee_info()
        return f'{employee_info}'

    def process_card_payment(self,fullname, emirates_id, date, amount,
description, card_username,card_number,cvc,otp):
        """
        Simulate processing a card payment.

        This version prompts the user for card details.

        Note: This is a simplified simulation and should not handle real
card information.
        """
        return self._accounting.card_payment_method(fullname, emirates_id,
date, amount, description,card_username,card_number,cvc,otp)
    def process_bank_payment(self,fullname, emirates_id, date, amount,
description,  bank_details, username, iban, account_number, branch):
        """

        Simulate processing a bank payment.
```

```
        This version prompts the user for bank details and basic account
detais .

        Note: This is a simplified simulation and should not handle real
card information.
        """
        return self._accounting.bank_payment_method(fullname, emirates_id,
date, amount, description, bank_details, username, iban, account_number,
branch)


    def __str__(self):
        """
        Generate a string representation of a finance employee.

        Returns:
            str: A string containing the finance employee's information.
        """
        employee_info = super().__str__()
        return f'{employee_info}'


class Buyer(Employee):
    """
    Represents a buyer in the hospital.
    """

    def __init__(self):
        super().__init__()


    def set_buyer_info(self, fullname, emirates_id, phone, date_of_birth,
education_degree, department, grade, buying_power):
        """
        Set the information of a buyer.

        Args:
            fullname (str): The full name of the buyer.
            emirates_id (str): The Emirates ID of the buyer.
```

```
            phone (str): The cell phone number of the buyer.
            date_of_birth (str): The date of birth of the buyer.
            education_degree (str): The highest education degree of the
buyer.
            department (str): The department in which the buyer works.
            grade (str): The grade or level of the buyer.

        """
        super().set_employee_info(fullname, emirates_id, phone,
date_of_birth, "Buyer", education_degree, department, grade)

    def get_buyer_info(self):
        """
        Get the information of a buyer.

        Returns:
            tuple: A tuple containing the buyer's information
            (name, Emirates ID, cell phone, date of birth, role, education
degree, department, grade, buying power).
        """
        return super().get_employee_info()
    def display_buyer_info(self):
        """
        Generate a formatted string representing the buyer's information.

        Returns:
            str: A formatted string containing the buyer's information.
        """
        employee_info = super().display_employee_info()
        return f'{employee_info}'

    def buy_property_card(self, fullname, emirates_id, date, amount,
description, card_username, card_number, cvc, otp):
        """
        Sell a property using card payment method.

        Args:
            fullname (str): Full name of the buyer.
            emirates_id (str): Emirates ID of the buyer.
            date (str): Date of the sale.
```

```
            amount (float): Amount of the sale.
            description (str): Description of the sale.
            card_username (str): Card username for payment.
            card_number (str): Card number for payment.
            cvc (str): Card CVC for payment.
            otp (str): OTP for card payment.

        Returns:
            str: Result message of the sale.
        """
        return self._sell_property.process_card_payment(fullname,
emirates_id, date, amount, description, card_username, card_number, cvc,
otp)

    def buy_property_bank(self, fullname, emirates_id, date, amount,
description, bank_details, username, iban, account_number, branch):
        """
        Sell a property using bank payment method.

        Args:
            fullname (str): Full name of the buyer.
            emirates_id (str): Emirates ID of the buyer.
            date (str): Date of the sale.
            amount (float): Amount of the sale.
            description (str): Description of the sale.
            bank_details (str): Bank details for payment.
            username (str): Bank username for payment.
            iban (str): IBAN for bank payment.
            account_number (str): Account number for bank payment.
            branch (str): Bank branch for payment.

        Returns:
            str: Result message of the sale.
        """
        return self._sell_property.process_bank_payment(fullname,
emirates_id, date, amount, description, bank_details, username, iban,
account_number, branch)
```

```
   def bought_properties(self, property_id, real_estate_type,
building_type, location, measurement, utilities, landmark, view,
buying_price, status,
                                    parking_spaces, floor_level, num_rooms,
kitchen, yard, floor_shape, num_floors, is_accessible, land_type,
is_developed,
                                    warehouse_space,
manufacturing_equipment, loading_docks):
       """
       Method to record the properties bought by a buyer.

       Args:
           property_id (str): Property ID.
           real_estate_type (str): Real Estate Type.
           building_type (str): Building Type.
           location (str): Location.
           measurement (str): Measurement.
           utilities (str): Utilities.
           landmark (str): Landmark.
           view (str): View.
           buying_price (float): Buying Price.
           status (str): Status.
           parking_spaces (int): Parking Spaces.
           floor_level (int): Floor Level.
           num_rooms (int): Number of Rooms.
           kitchen (str): Kitchen.
           yard (str): Yard.
           floor_shape (str): Floor Shape.
           num_floors (int): Number of Floors.
           is_accessible (bool): Is Accessible.
           land_type (str): Land Type.
           is_developed (bool): Is Developed.
           warehouse_space (str): Warehouse Space.
           manufacturing_equipment (str): Manufacturing Equipment.
           loading_docks (bool): Loading Docks.
       """
       # Use the set_real_estate_properties method of the Property class
to record the bought property
```

```
        Property().set_real_estate_properties(property_id,
real_estate_type, building_type, location, measurement, utilities,
landmark, view, buying_price, status,
                                              parking_spaces, floor_level,
num_rooms, kitchen, yard, floor_shape, num_floors, is_accessible,
land_type, is_developed,
                                              warehouse_space,
manufacturing_equipment, loading_docks)

    def get_properties_buyer_view(self):
        """
        Method to get the properties information from the perspective of a
buyer.

        Returns:
            list: List of properties information.
        """
        # Use the get_properties_info method of the Administrator class to
get the properties information
        return Administrator().get_properties_info()


    def __str__(self):
        """
        Generate a string representation of a buyer.

        Returns:
            str: A string containing the buyer's information.
        """
        employee_info = super().__str__()
        return f'{employee_info}'


class Seller(Employee):
    """
    Represents a seller in the hospital.
    """

    def __init__(self):
        super().__init__()
```

```
        self._sell_property = Finance()
        self._get_propery = Administrator()

    def set_seller_info(self, fullname, emirates_id, phone, date_of_birth,
education_degree, department, grade, property_listings):
        """
        Set the information of a seller.

        Args:
            fullname (str): The full name of the seller.
            emirates_id (str): The Emirates ID of the seller.
            phone (str): The cell phone number of the seller.
            date_of_birth (str): The date of birth of the seller.
            education_degree (str): The highest education degree of the
seller.
            department (str): The department in which the seller works.
            grade (str): The grade or level of the seller.
            property_listings (list): A list of property listings
associated with the seller.
        """
        super().set_employee_info(fullname, emirates_id, phone,
date_of_birth, "Seller", education_degree, department, grade)
        self._property_listings = property_listings

    def get_seller_info(self):
        """
        Get the information of a seller.

        Returns:
            tuple: A tuple containing the seller's information
            (name, Emirates ID, cell phone, date of birth, role, education
degree, department, grade, property listings).
        """
        return super().get_employee_info()

    def display_seller_info(self):
        """
        Generate a formatted string representing the seller's information.

        Returns:
```

```
            str: A formatted string containing the seller's information.
        """
        employee_info = super().display_employee_info()
        return f'{employee_info}'

    def sell_property_card(self, fullname, emirates_id, date, amount,
description, card_username, card_number, cvc, otp):
        """
        Sell a property using card payment method.

        Args:
            fullname (str): Full name of the buyer.
            emirates_id (str): Emirates ID of the buyer.
            date (str): Date of the sale.
            amount (float): Amount of the sale.
            description (str): Description of the sale.
            card_username (str): Card username for payment.
            card_number (str): Card number for payment.
            cvc (str): Card CVC for payment.
            otp (str): OTP for card payment.

        Returns:
            str: Result message of the sale.
        """
        return self._sell_property.process_card_payment(fullname,
emirates_id, date, amount, description, card_username, card_number, cvc,
otp)

    def sell_property_bank(self, fullname, emirates_id, date, amount,
description, bank_details, username, iban, account_number, branch):
        """
        Sell a property using bank payment method.

        Args:
            fullname (str): Full name of the buyer.
            emirates_id (str): Emirates ID of the buyer.
            date (str): Date of the sale.
            amount (float): Amount of the sale.
            description (str): Description of the sale.
            bank_details (str): Bank details for payment.
```

```
            username (str): Bank username for payment.
            iban (str): IBAN for bank payment.
            account_number (str): Account number for bank payment.
            branch (str): Bank branch for payment.

        Returns:
            str: Result message of the sale.
        """
        return self._sell_property.process_bank_payment(fullname,
emirates_id, date, amount, description, bank_details, username, iban,
account_number, branch)

    def get_properties_seller_view(self):
        """
        Method to get the properties information from the perspective of a
buyer.

        Returns:
            list: List of properties information.
        """
        # Use the get_properties_info method of the Administrator class to
get the properties information
        return Administrator().get_properties_info()

    def remove_property_sold(self, property_id_to_delete):
        """
        Method to remove a sold property from the CSV file.

        Args:
            property_id_to_delete (str): Property ID to be removed.
        """
        # Utilize the delete_property method to remove the property
        Property().delete_property(property_id_to_delete)

    def __str__(self):
        """
        Generate a string representation of a seller.

        Returns:
            str: A string containing the seller's information.
```

```
        """
        employee_info = super().__str__()
        return f'{employee_info}, Property Listings:
{str(self._property_listings)}'




import tkinter as tk
from tkinter import ttk
import csv
import os







class RealEstateGUI(Property):
    def __init__(self, root):
        super().__init__()

        self.root = tk.Tk()
        self.root.title("Real Estate Properties")

        self.create_entries()



        # Create Submit Button
        self.submit_button = ttk.Button(root, text="Submit",
command=self.submit_properties)
        self.submit_button.grid(row=2, column=0, columnspan=2, pady=10)
        self.root.mainloop()

    def submit_properties(self):
      attributes = [ "Property ID", "Real Estate Type", "Building Type",
"Location", "Measurement", "Utilities", "Landmark",
                          "View", "Parking Space", "Floor Level", "Number of
Rooms", "Kitchen", "Yard", "Floor Shape",
```

```
                        "Number of Floors", "Is Accessible", "Land Type",
"Is Developed", "Warehouse Space",
                        "Manufacturing Equipment", "Loading Docks", "Buying
Price", "Status"]

    # Initialize an empty dictionary to store the values
    values = {}

    for attribute in attributes:
        # Use getattr to dynamically access the entry widget based on the
attribute name
        entry_value = getattr(self, f"{attribute}_entry").get()

        # Store the value in the dictionary
        values[attribute] = entry_value

    # Now you have a dictionary 'values' containing all the entered
values
    print("Submitted Properties:", values)

    # Convert the dictionary values to a tuple and write to CSV
    self.set_real_estate_properties(tuple(values.values()))

  def create_entries(self):
      attributes = [
          "property_id", "real_estate_type", "typ_e", "location", "size",
"utilities",
          "landmark", "view", "buying_price", "status", "parking_spaces",
"floor_level",
          "room", "yard", "floor_space", "num_floors", "is_accessible",
"land_type",
          "is_developed", "warehouse_space", "manufacturing_equipment",
"has_loading_docks", "kitchen"
      ]

      for i, attribute in enumerate(attributes):
          label = ttk.Label(self.root, text=f"{attribute.replace('_', '
').title()}:")
          label.grid(row=i, column=0, padx=5, pady=5, sticky=tk.W)
```

```
            entry = ttk.Entry(self.root)
            entry.grid(row=i, column=1, padx=5, pady=5, sticky=tk.EW)

            # If you want to keep references to these entries, you can
store them in a dictionary
            setattr(self, f"{attribute}_entry", entry)

class CsvReaderApp:
    def __init__(self, root):
        self.root = tk.Tk()
        self.root.title("CSV Reader App")

        # Create Treeview
        self.tree = ttk.Treeview(root, columns=self.get_headers(),
show="headings", selectmode="browse")
        self.tree.pack(padx=10, pady=10)

        # Add Scrollbars
        yscrollbar = ttk.Scrollbar(root, orient="vertical",
command=self.tree.yview)
        yscrollbar.pack(side="right", fill="y")
        self.tree.configure(yscrollcommand=yscrollbar.set)

        xscrollbar = ttk.Scrollbar(root, orient="horizontal",
command=self.tree.xview)
        xscrollbar.pack(side="bottom", fill="x")
        self.tree.configure(xscrollcommand=xscrollbar.set)

        # Add Headers to Treeview
        for header in self.get_headers():
            self.tree.heading(header, text=header)
            self.tree.column(header, anchor="center")

        # Create Filter Frame
        filter_frame = ttk.Frame(root)
        filter_frame.pack(pady=10)

        # Create Filter Entry
        self.filter_entry = ttk.Entry(filter_frame)
        self.filter_entry.pack(side="left", padx=5)
```

```
        # Create Filter Button
        filter_button = ttk.Button(filter_frame, text="Filter",
command=self.apply_filter)
        filter_button.pack(side="left", padx=5)

        # Create Search Frame
        search_frame = ttk.Frame(root)
        search_frame.pack(pady=10)

        # Create Search Entry
        self.search_entry = ttk.Entry(search_frame)
        self.search_entry.pack(side="left", padx=5)

        # Create Search Button
        search_button = ttk.Button(search_frame, text="Search",
command=self.search_data)
        search_button.pack(side="left", padx=5)

        # Load CSV Data
        self.load_data()

    def get_headers(self):
        # Add your CSV headers here
        return ["Property ID", "Real Estate Type", "Building Type",
"Location", "Buying Price", "Status"]

    def load_data(self):
        # Load data from CSV and populate Treeview
        file_path = "real_estate_storage.csv"
        with open(file_path, "r") as file:
            reader = csv.DictReader(file)
            for row in reader:
                self.tree.insert("", "end", values=list(row.values()))

    def apply_filter(self):
        # Apply filter to Treeview based on the filter entry
        filter_text = self.filter_entry.get().lower()
        for row_id in self.tree.get_children():
            values = self.tree.item(row_id)["values"]
```

```
            if filter_text in str(values).lower():
                self.tree.item(row_id, open=True)
            else:
                self.tree.item(row_id, open=False)

    def search_data(self):
        # Search for data in Treeview based on the search entry
        search_text = self.search_entry.get().lower()
        for row_id in self.tree.get_children():
            values = self.tree.item(row_id)["values"]
            if search_text in str(values).lower():
                self.tree.selection_set(row_id)
                self.tree.focus(row_id)



from tkinter import simpledialog

import tkinter as tk
from tkinter import ttk
import pandas as pd

class CSVDataViewer:
    def __init__(self, data):
        self.master = tk.TK()
        self.master.title("CSV Data Viewer")


        # Load CSV data
        self._df = pd.DataFrame(data)

        # Create Treeview
        self.tree = ttk.Treeview(self.master)
        self.tree["columns"] = tuple(self.df.columns)
        self.tree.heading("#0", text="Index")
        for col in self.df.columns:
            self.tree.heading(col, text=col)
        self.tree.pack(expand=True, fill="both")

        # Search Entry
```

```
        self.search_var = tk.StringVar()
        self.search_entry = tk.Entry(self.master,
textvariable=self.search_var, width=20)
        self.search_entry.pack(pady=5)
        search_button = tk.Button(self.master, text="Search",
command=self.search_data)
        search_button.pack()

        # Filter Entry
        self.filter_var = tk.StringVar()
        self.filter_combobox = ttk.Combobox(self.master,
textvariable=self.filter_var, values=list(self.df.columns))
        self.filter_combobox.set(self.df.columns[0])
        self.filter_combobox.pack(pady=5)
        filter_button = tk.Button(self.master, text="Filter",
command=self.filter_data)
        filter_button.pack()
        self.master.mainloop()

    def search_data(self):
        query = self.search_var.get().lower()
        self.filter_and_display(query=query)

    def filter_data(self):
        filter_col = self.filter_var.get().lower()
        self.filter_and_display(filter_col=filter_col)

    def filter_and_display(self, query=None, filter_col=None):
        if query:
            self.df = self.df[self.df.apply(lambda row: any(query in
str(cell).lower() for cell in row), axis=1)]
        elif filter_col:
            selected_value = self.filter_combobox.get()
            self.df = self.df[self.df[filter_col] == selected_value]

        self.update_treeview()

    def update_treeview(self):
        # Clear existing items in the Treeview
        for item in self.tree.get_children():
```

```
            self.tree.delete(item)

        # Insert data into the Treeview
        for index, row in self.df.iterrows():
            values = tuple(row.values)
            self.tree.insert("", "end", text=index, values=values)




import tkinter as tk
from tkinter import ttk, messagebox


class FinanceDashboard:
    def __init__(self):
        self.master = tk.Tk()
        self.master.title("Finance Dashboard")

        # Create variables
        self.payment_method_var = tk.StringVar()
        self.fullname_var = tk.StringVar()
        self.emirates_id_var = tk.StringVar()
        self.date_var = tk.StringVar()
        self.amount_var = tk.StringVar()
        self.description_var = tk.StringVar()
        self.bank_username_var = tk.StringVar()
        self.iban_var = tk.StringVar()
        self.account_number_var = tk.StringVar()
        self.branch_var = tk.StringVar()
        self.card_username_var = tk.StringVar()
        self.card_number_var = tk.StringVar()
        self.cvc_var = tk.StringVar()
        self.otp_var = tk.StringVar()
        self.property_id_to_delete_var = tk.StringVar()

        # Create widgets
        self.payment_method_label = tk.Label(self.master, text="Select
Payment Method:")
```

```
        self.payment_method_label.grid(row=0, column=0, padx=10, pady=10)

        # Radio buttons to select payment method
        self.bank_radio = tk.Radiobutton(self.master, text="Bank",
variable=self.payment_method_var, value="bank")
        self.bank_radio.grid(row=0, column=1, padx=10, pady=10)
        self.card_radio = tk.Radiobutton(self.master, text="Card",
variable=self.payment_method_var, value="card")
        self.card_radio.grid(row=0, column=2, padx=10, pady=10)

        # Button to confirm selection
        self.confirm_button = tk.Button(self.master, text="Confirm",
command=self.show_payment_entries)
        self.confirm_button.grid(row=1, column=0, columnspan=3, pady=10)

        # Create a treeview to display inputs
        self.tree = ttk.Treeview(self.master, columns=("Label", "Value"),
show="headings")
        self.tree.grid(row=2, column=0, columnspan=3, padx=10, pady=10)

        # Set column headings
        self.tree.heading("Label", text="Label")
        self.tree.heading("Value", text="Value")

        self.master.mainloop()

    def show_payment_entries(self):
        # Get selected payment method
        selected_method = self.payment_method_var.get()

        # Clear existing entries in the treeview
        for item in self.tree.get_children():
            self.tree.delete(item)

        # Check which method was selected and show corresponding entries
        if selected_method == "bank":
            self.show_bank_entries()
        elif selected_method == "card":
            self.show_card_entries()
        else:
```

```
            messagebox.showerror("Error", "Please select a payment
method.")

    def show_bank_entries(self):
        # Create a Toplevel window for bank entries
        bank_window = tk.Toplevel(self.master)
        bank_window.title("Bank Payment")

        # Create widgets for bank payment entries
        self.bank_username_label = tk.Label(bank_window, text="Bank
Username:")
        self.bank_username_label.grid(row=0, column=0, padx=10, pady=10)
        self.bank_username_entry = tk.Entry(bank_window,
textvariable=self.bank_username_var)
        self.bank_username_entry.grid(row=0, column=1, padx=10, pady=10)

        self.iban_label = tk.Label(bank_window, text="IBAN:")
        self.iban_label.grid(row=1, column=0, padx=10, pady=10)
        self.iban_entry = tk.Entry(bank_window, textvariable=self.iban_var)
        self.iban_entry.grid(row=1, column=1, padx=10, pady=10)

        self.account_number_label = tk.Label(bank_window, text="Account
Number:")
        self.account_number_label.grid(row=2, column=0, padx=10, pady=10)
        self.account_number_entry = tk.Entry(bank_window,
textvariable=self.account_number_var)
        self.account_number_entry.grid(row=2, column=1, padx=10, pady=10)

        self.branch_label = tk.Label(bank_window, text="Branch:")
        self.branch_label.grid(row=3, column=0, padx=10, pady=10)
        self.branch_entry = tk.Entry(bank_window,
textvariable=self.branch_var)
        self.branch_entry.grid(row=3, column=1, padx=10, pady=10)

        # Create widgets for common entries
        self.create_common_entries(bank_window)

        bank_window.mainloop()

    def show_card_entries(self):
```

```
        # Create a Toplevel window for card entries
        card_window = tk.Toplevel(self.master)
        card_window.title("Card Payment")

        # Create widgets for card payment entries
        self.card_username_label = tk.Label(card_window, text="Card
Username:")
        self.card_username_label.grid(row=0, column=0, padx=10, pady=10)
        self.card_username_entry = tk.Entry(card_window,
textvariable=self.card_username_var)
        self.card_username_entry.grid(row=0, column=1, padx=10, pady=10)

        self.card_number_label = tk.Label(card_window, text="Card Number:")
        self.card_number_label.grid(row=1, column=0, padx=10, pady=10)
        self.card_number_entry = tk.Entry(card_window,
textvariable=self.card_number_var)
        self.card_number_entry.grid(row=1, column=1, padx=10, pady=10)

        self.cvc_label = tk.Label(card_window, text="CVC:")
        self.cvc_label.grid(row=2, column=0, padx=10, pady=10)
        self.cvc_entry = tk.Entry(card_window, textvariable=self.cvc_var)
        self.cvc_entry.grid(row=2, column=1, padx=10, pady=10)

        self.otp_label = tk.Label(card_window, text="OTP:")
        self.otp_label.grid(row=3, column=0, padx=10, pady=10)
        self.otp_entry = tk.Entry(card_window, textvariable=self.otp_var)
        self.otp_entry.grid(row=3, column=1, padx=10, pady=10)

        # Create widgets for common entries
        self.create_common_entries(card_window)

        card_window.mainloop()

    def create_common_entries(self, window):
        # Create widgets for common entries
        common_entries = [
            ("Full Name:", self.fullname_var),
            ("Emirates ID:", self.emirates_id_var),
            ("Date:", self.date_var),
            ("Amount:", self.amount_var),
```

```
            ("Description:", self.description_var),
            ("Property ID to Delete:", self.property_id_to_delete_var),
        ]

        for idx, (label, var) in enumerate(common_entries):
            label_entry = tk.Label(window, text=label)
            label_entry.grid(row=idx + 4, column=0, padx=10, pady=10)
            entry_widget = tk.Entry(window, textvariable=var)
            entry_widget.grid(row=idx + 4, column=1, padx=10, pady=10)

        delete_property_button = tk.Button(window, text="Delete Property",
command=self.delete_property)
        delete_property_button.grid(row=idx + 5, column=0, columnspan=2,
pady=10)

    def delete_property(self):
        # Get property ID to delete
        property_id_to_delete = self.property_id_to_delete_var.get()

        # Call the delete_property method
        Property().delete_property(property_id_to_delete)
        print(f"Deleting property with ID: {property_id_to_delete}")


FinanceDashboard()


import unittest
from unittest.mock import patch, Mock
import tkinter as tk


class TestFinanceDashboard(unittest.TestCase):
    @patch("tkinter.Tk.mainloop")
    @patch("tkinter.Label.pack")
    @patch("tkinter.Entry.pack")
    @patch("tkinter.Radiobutton.pack")
    @patch("tkinter.Button.pack")
    def test_show_payment_entries_bank(self, mock_button_pack,
mock_radiobutton_pack, mock_entry_pack, mock_label_pack, mock_mainloop):
```

```
        # Mock the FinanceDashboard class
        finance_dashboard_mock = Mock(spec=FinanceDashboard)
        finance_dashboard_mock.payment_method_var.get.return_value = "bank"
        finance_dashboard_mock.master = tk.Tk()

        # Call the show_payment_entries method
        finance_dashboard_mock.show_payment_entries()

        # Assert that the correct widgets are packed
        mock_label_pack.assert_called_with(side="top")
        mock_radiobutton_pack.assert_called_with(side="top")
        mock_radiobutton_pack.assert_called_with(side="top")
        mock_button_pack.assert_called_with(side="top")

    @patch("tkinter.Toplevel")
    @patch("tkinter.Label.pack")
    @patch("tkinter.Entry.pack")
    @patch("tkinter.Button.pack")
    def test_show_bank_entries(self, mock_button_pack, mock_entry_pack,
mock_label_pack, mock_toplevel):
        # Mock the FinanceDashboard class
        finance_dashboard_mock = Mock(spec=FinanceDashboard)
        finance_dashboard_mock.master = tk.Tk()

        # Set the selected method to "bank"
        finance_dashboard_mock.payment_method_var.get.return_value = "bank"

        # Call the show_bank_entries method
        finance_dashboard_mock.show_bank_entries()

        # Assert that the correct widgets are packed
        mock_label_pack.assert_called_with(side="top")
        mock_entry_pack.assert_called_with(side="top")
        mock_entry_pack.assert_called_with(side="top")
        mock_entry_pack.assert_called_with(side="top")
        mock_entry_pack.assert_called_with(side="top")
        mock_button_pack.assert_called_with(side="top")

    # Add similar tests for show_card_entries and other methods
```

```
if __name__ == "__main__":
    unittest.main()


import tkinter as tk
from tkinter import messagebox

class FinanceDashboard:
    def __init__(self):
        self.master = tk.Tk()
        self.master.title("Finance Dashboard")

        # Create a button to choose from different class windows
        self.choose_button = tk.Button(self.master, text="Choose Window",
command=self.show_window_options)
        self.choose_button.pack()

    def show_window_options(self):
        # Create a Toplevel window to display options
        options_window = tk.Toplevel(self.master)
        options_window.title("Choose Window")

        # Create buttons for different class windows
        bank_button = tk.Button(options_window, text="Bank Window",
command=self.show_bank_window)
        bank_button.pack()

        card_button = tk.Button(options_window, text="Card Window",
command=self.show_card_window)
        card_button.pack()

    def show_bank_window(self):
        # Open the BankWindow class window
        BankWindow()

    def show_card_window(self):
        # Open the CardWindow class window
        CardWindow()

class BankWindow:
```

```
    def __init__(self):
        self.bank_window = tk.Toplevel()
        self.bank_window.title("Bank Window")

        # Add BankWindow specific widgets here

class CardWindow:
    def __init__(self):
        self.card_window = tk.Toplevel()
        self.card_window.title("Card Window")

        # Add CardWindow specific widgets here

if __name__ == "__main__":
    FinanceDashboard().master.mainloop()



import unittest
from unittest.mock import patch
from tkinter import Tk


import tkinter as tk
from tkinter import ttk
from tkinter import messagebox
from datetime import datetime



class EmployeeManagementWindow:
    def __init__(self):
        self.window = tk.Tk()
        self.window.title("Employee Management")

        # Radio buttons for user options
        self.selected_option = tk.StringVar(value="add")  # Default option
is "Add Employee"
        options = [("Add Employee", "add"), ("View Employees", "view"),
("Delete Employee", "delete")]
```

```
        for text, value in options:
            tk.Radiobutton(self.window, text=text,
variable=self.selected_option, value=value).pack(pady=5)

        # Button to proceed with the selected option
        tk.Button(self.window, text="Proceed",
command=self.handle_option).pack(pady=10)

        # TreeView to display employee information
        self.tree = ttk.Treeview(self.window, columns=("Full Name",
"Emirates ID", "Phone", "Date of Birth", "Role", "Education Degree",
"Department", "Grade", "Salary"))
        self.tree.heading("#0", text="Employee ID")
        self.tree.heading("Full Name", text="Full Name")
        self.tree.heading("Emirates ID", text="Emirates ID")
        self.tree.heading("Phone", text="Phone")
        self.tree.heading("Date of Birth", text="Date of Birth")
        self.tree.heading("Role", text="Role")
        self.tree.heading("Education Degree", text="Education Degree")
        self.tree.heading("Department", text="Department")
        self.tree.heading("Grade", text="Grade")
        self.tree.heading("Salary", text="Salary")

        self.tree.pack(pady=10)

    def handle_option(self):
        option = self.selected_option.get()

        if option == "add":
            # Open the EmployeeEntryWindow for adding an employee
            EmployeeEntryWindow(self.window, self.tree)

        elif option == "view":
            # View employees and update the TreeView
            self.view_all_employees()

        elif option == "delete":
            # Open the EmployeeDeleteWindow for deleting an employee
            EmployeeDeleteWindow(self.window, self.tree)
```

```
    def view_all_employees(self):
        # Clear existing data in TreeView
        for item in self.tree.get_children():
            self.tree.delete(item)

        # Get employee data and populate TreeView
        employees_data = EmployeeCsv().get_store_employee_info()
        for index, row in employees_data.iterrows():
            self.tree.insert("", "end", values=tuple(row))
class EmployeeEntryWindow:
    def __init__(self, parent, tree):
        self.parent = parent
        self.tree = tree
        self.window = tk.Toplevel(parent)
        self.window.title("Employee Entry")

        # Create entry labels and widgets
        labels = ["Full Name", "Emirates ID", "Phone", "Date of Birth",
"Role", "Education Degree", "Department", "Grade", "Salary"]
        self.entry_vars = [tk.StringVar() for _ in labels]

        for i, label in enumerate(labels):
            tk.Label(self.window, text=label).grid(row=i, column=0,
padx=10, pady=10)
            tk.Entry(self.window,
textvariable=self.entry_vars[i]).grid(row=i, column=1, padx=10, pady=10)

        # Buttons for submitting and clearing entries
        tk.Button(self.window, text="Submit",
command=self.submit_entries).grid(row=len(labels), column=0, pady=10)
        tk.Button(self.window, text="Clear",
command=self.clear_entries).grid(row=len(labels), column=1, pady=10)

    def submit_entries(self):
        try:
            # Get values from entry widgets
            fullname, emirates_id, phone, date_of_birth, role,
education_degree, department, grade, salary = [var.get() for var in
self.entry_vars]
```

```
            # Validate and convert date_of_birth to the required format
            datetime.strptime(date_of_birth, '%Y-%m-%d')

            # Validate and convert salary to float
            salary = float(salary)

            # Call the set_store_employee_info method
            EmployeeCsv().set_store_employee_info(fullname, emirates_id,
phone, date_of_birth, role, education_degree, department, grade, salary)

            messagebox.showinfo("Success", "Employee information stored
successfully.")
            self.clear_entries()
            self.tree.delete(*self.tree.get_children())  # Clear existing
data in TreeView
            self.tree.update()
            self.parent.update()

        except ValueError:
            messagebox.showerror("Error", "Invalid input. Please check your
entries.")

    def clear_entries(self):
        # Clear the entry widgets
        for var in self.entry_vars:
            var.set("")

        self.window.destroy()

class EmployeeDeleteWindow:
    def __init__(self, parent, tree):
        self.parent = parent
        self.tree = tree
        self.window = tk.Toplevel(parent)
        self.window.title("Employee Deletion")

        # Create entry label and widget for employee ID
        tk.Label(self.window, text="Employee ID to Delete").grid(row=0,
column=0, padx=10, pady=10)
```

```
        self.employee_id_var = tk.StringVar()
        tk.Entry(self.window,
textvariable=self.employee_id_var).grid(row=0, column=1, padx=10, pady=10)

        # Button to delete employee
        tk.Button(self.window, text="Delete Employee",
command=self.delete_employee).grid(row=1, columnspan=2, pady=10)

    def delete_employee(self):
        try:
            # Get the employee ID to delete
            employee_id_to_delete = self.employee_id_var.get()

            # Call the delete_employee method
            EmployeeCsv().delete_employee(employee_id_to_delete)

            messagebox.showinfo("Success", f"Employee with ID
{employee_id_to_delete} deleted successfully.")
            self.window.destroy()

            # Update the TreeView after deletion
            self.tree.delete(*self.tree.get_children())
            self.tree.update()
            self.parent.update()

        except Exception as e:
            messagebox.showerror("Error", str(e))

if __name__ == "__main__":
    management_window = EmployeeManagementWindow()
    management_window.window.mainloop()


class TestEmployeeManagementWindow(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        # This method is called before any tests in the class are run
        # Create a Tkinter root window for testing
        cls.root = Tk()
        cls.root.withdraw()  # Hide the main window during tests
```

```
    @classmethod
    def tearDownClass(cls):
        # This method is called after all tests in the class are run
        cls.root.destroy()

    def test_view_all_employees(self):
        # Mock the EmployeeCsv class to return a sample DataFrame
        with patch.object(EmployeeCsv, 'get_store_employee_info',
return_value=self.get_sample_data()):
            window = EmployeeManagementWindow()
            window.view_all_employees()

            # Check if the TreeView is populated with the correct number of
items
            items = window.tree.get_children()
            self.assertEqual(len(items), 3)

    def test_handle_option_add(self):
        with patch.object(EmployeeEntryWindow, '__init__',
return_value=None):
            window = EmployeeManagementWindow()
            window.handle_option()

            # Check if the EmployeeEntryWindow was initialized
            self.assertTrue(isinstance(window.add_window,
EmployeeEntryWindow))

    def test_handle_option_view(self):
        # Mock the view_all_employees method
        with patch.object(EmployeeManagementWindow, 'view_all_employees',
return_value=None):
            window = EmployeeManagementWindow()
            window.handle_option()

            # Check if the view_all_employees method was called

EmployeeManagementWindow.view_all_employees.assert_called_once()

    def test_handle_option_delete(self):
```

```
        with patch.object(EmployeeDeleteWindow, '__init__',
return_value=None):
            window = EmployeeManagementWindow()
            window.handle_option()

            # Check if the EmployeeDeleteWindow was initialized
            self.assertTrue(isinstance(window.delete_window,
EmployeeDeleteWindow))

    def get_sample_data(self):
        # Helper method to return sample employee data for testing
        return {'Full Name': ['John Doe', 'Jane Smith', 'Bob Johnson'],
                'Emirates ID': ['123456789012345', '987654321098765',
'456789012345678'],
                'Phone': ['1234567890', '9876543210', '5555555555'],
                'Date of Birth': ['1990-01-01', '1985-05-10',
'1995-12-25'],
                'Role': ['Manager', 'Engineer', 'Analyst'],
                'Education Degree': ['MBA', 'BSc', 'PhD'],
                'Department': ['HR', 'Engineering', 'Finance'],
                'Grade': ['A', 'B', 'C'],
                'Salary': [80000.0, 60000.0, 50000.0]}

if __name__ == '__main__':
    unittest.main()

import tkinter as tk
from tkinter import ttk, messagebox


class AddPropertyWindow:
    def __init__(self, master):
        self.master = master
        self.window = tk.Toplevel(master)
        self.window.title("Add Real Estate Property")

        # Create labels and entry widgets
        labels = ["Property ID", "Real Estate Type", "Building Type",
"Location", "Measurement", "Utilities", "Landmark",
```

```
                "View", "Buying Price", "Status", "Parking Spaces",
"Floor Level", "Number of Rooms", "Kitchen", "Yard",
                "Floor Shape", "Number of Floors", "Is Accessible", "Land
Type", "Is Developed", "Warehouse Space",
                "Manufacturing Equipment", "Loading Docks"]

        self.entry_vars = [tk.StringVar() for _ in labels]

        for i, label in enumerate(labels):
            ttk.Label(self.window, text=label).grid(row=i, column=0,
padx=1, pady=1)
            ttk.Entry(self.window,
textvariable=self.entry_vars[i]).grid(row=i, column=1, padx=10, pady=2)

        # Buttons to submit the property and clear entries
        ttk.Button(self.window, text="Submit",
command=self.submit_property).grid(row=len(labels), column=0, pady=2)
        ttk.Button(self.window, text="Clear",
command=self.clear_entries).grid(row=len(labels), column=1, pady=2)

    def submit_property(self):
        try:
            # Get values from entry widgets
            property_values = [var.get() for var in self.entry_vars]

            # Call the set_real_estate_properties method
            Property().set_real_estate_properties(*property_values)

            messagebox.showinfo("Success", "Real Estate Property added
successfully.")
            self.clear_entries()
            self.window.destroy()

        except ValueError:
            messagebox.showerror("Error", "Invalid input. Please check your
entries.")

    def clear_entries(self):
        # Clear the entry widgets
        for var in self.entry_vars:
```

```
            var.set("")

# Example usage:
if __name__ == "__main__":
    root = tk.Tk()
    add_property_window = AddPropertyWindow(root)
    root.mainloop()
import unittest
from unittest.mock import patch
import tkinter as tk
from tkinter import ttk, messagebox

class TestAddPropertyWindow(unittest.TestCase):
    def setUp(self):
        self.root = tk.Tk()

    def tearDown(self):
        self.root.destroy()

    def test_submit_property_success(self):
        with patch.object(Property, 'set_real_estate_properties'):
            window = AddPropertyWindow(self.root)
            window.entry_vars[0].set("123")
            window.entry_vars[1].set("House")
            # ... set other entry values

            window.submit_property()

            # Check if Property method is called
            Property().set_real_estate_properties.assert_called_with("123",
"House", ...)

            # Check if messagebox.showinfo is called
            self.assertEqual(messagebox.showinfo.call_args[0][1], "Real
Estate Property added successfully.")

            # Check if clear_entries is called
            window.clear_entries.assert_called()

            # Check if the window is destroyed
```

```
            self.assertTrue(window.window.winfo_ismapped())

    def test_submit_property_failure(self):
        with patch.object(Property, 'set_real_estate_properties',
side_effect=ValueError("Invalid input")):
            window = AddPropertyWindow(self.root)
            window.submit_property()

            # Check if Property method is called
            Property().set_real_estate_properties.assert_called()

            # Check if messagebox.showerror is called
            self.assertEqual(messagebox.showerror.call_args[0][1], "Invalid
input. Please check your entries.")

            # Check if clear_entries is called
            window.clear_entries.assert_called()

            # Check if the window is not destroyed
            self.assertFalse(window.window.winfo_ismapped())

if __name__ == '__main__':
    unittest.main()

class UserGUI:
    def __init__(self):
        self.root = tk.Tk()
        self.root.title("User Registration/Login")

        # Labels
        tk.Label(self.root, text="Username:").grid(row=0, column=0, pady=5)
        tk.Label(self.root, text="Password:").grid(row=1, column=0, pady=5)
        tk.Label(self.root, text="OTP:").grid(row=2, column=0, pady=5)
        tk.Label(self.root, text="Role:").grid(row=3, column=0, pady=5)

        # Entries and Dropdown
        self.username_entry = tk.Entry(self.root)
        self.password_entry = tk.Entry(self.root, show="*")
        self.otp_entry = tk.Entry(self.root, show="*")
        self.role_var = tk.StringVar(self.root)
```

```
        self.role_var.set("Customer")
        self.role_menu = tk.OptionMenu(self.root, self.role_var,
*["Customer", "Buyer", "Seller", "Administrator"])

        self.username_entry.grid(row=0, column=1, pady=5)
        self.password_entry.grid(row=1, column=1, pady=5)
        self.otp_entry.grid(row=2, column=1, pady=5)
        self.role_menu.grid(row=3, column=1, pady=5)

        # Buttons
        tk.Button(self.root, text="Register",
command=self.register_user).grid(row=4, column=0, columnspan=2, pady=10)
        tk.Button(self.root, text="Login",
command=self.login_user).grid(row=5, column=0, columnspan=2, pady=10)
        self.root.mainloop()

    def generate_otp(self):
        otp = self.user.generate_otp()
        self.otp_entry.delete(0, tk.END)
        messagebox.showinfo("Generated OTP", f"Generated OTP: {otp}")

    def register_user(self):
        entered_username = self.username_entry.get()
        entered_password = self.password_entry.get()
        entered_secret_code = self.otp_entry.get()
        entered_role =self.role_var.get()

        # Use the User class to handle registration
        user = User()
        registration_result =
user.register_user(entered_role,entered_username,
entered_password,entered_password, entered_secret_code)

        # Display registration result
        messagebox.showinfo("Registration Result",
f"{registration_result}")

    def login_user(self):
        entered_username = self.username_entry.get()
        entered_password = self.password_entry.get()
```

```
        entered_otp = self.otp_entry.get()
        selected_role = self.role_var.get()

        # Ask for the user's role
        user_role = simpledialog.askstring("User Role", "Enter your role
(buyer, seller, administrator, or customer):")

        if user_role.lower()== "customer":

          return
CSVDataViewer(Administrator().get_properties_info()),FinanceDashboard()

        elif user_role.lower()== "buyer":
          return FinanceDashboard(),AddPropertyWindow
        elif user_role.lower()== "seller":
          return
FinanceDashboard(),CSVDataViewer(Administrator().get_properties_info())
        elif user_role.lower()== "administrator":
          return
CSVDataViewer(Administrator().get_properties_info()),EmployeeManagementWin
dow()

        # Use the User class to handle login
        user = User()
        login_result = user.login_user(entered_username, entered_password,
entered_otp, user_role)

        # Display login result
        messagebox.showinfo("Login Result", f"{login_result}")
```

Reflection

In my recent exploration of computer science and software development, I delved into several key concepts that significantly broadened my understanding. Firstly, I gained insights into the creation of Graphical User Interfaces (GUI), learning how to design visually intuitive and user-friendly interfaces that enhance the overall user experience. Additionally, I delved into the world of UML (Unified Modeling Language) class and case diagrams, allowing me to visually represent the structure and interactions of various system components. Furthermore, I expanded my knowledge by exploring the

implementation of RSA (Rivest–Shamir–Adleman) encryption within the context of data storage. This cryptographic technique has proven instrumental in securing sensitive information, and understanding its application in storage systems has provided me with a comprehensive perspective on data security and protection. Overall, this multifaceted learning experience has equipped me with valuable skills and insights, enhancing my proficiency in software development and system design

Link

https://github.com/h123457/estate.