Description of each module in Alzheimer’s Disease Testing Software

The modules are designed with principles of modularity, reusability, low coupling, and high cohesion in mind. This design approach enhances maintainability and scalability, allowing for easier updates and integration of new features in the future.

1. Database Module

```
import mysql.connector
from mysql.connector import Error

class Database:
    """Handles database connections and operations."""

    @staticmethod
    def connect_db():
        """
        Establishes a connection to the MySQL database.

        Returns:
            connection: A MySQL connection object or None if connection fails.
        """
        try:
            connection = mysql.connector.connect(
                host='localhost',
                database='alzheimer_testing',
                user='root',
                password='your_password'
            )
            if connection.is_connected():
                return connection
        except Error as e:
            print("Error while connecting to MySQL", e)
        return None
```

- **Modularity and Reusability**: This module is self-contained, allowing other modules to reuse the connect_db method without duplicating code.
- **Coupling and Cohesion**: The module has low coupling with other modules, as it only provides database connection functionality. It has high cohesion since all methods relate to database operations.
- **Data Coupling**: The connection details are encapsulated within the module, minimizing dependencies on external data.
- **Functional Cohesion**: The module focuses solely on database interactions, ensuring that all functions serve a single purpose.

2. User Model Module

```
from database import Database

class UserModel:
    """Represents a user in the system."""
    def __init__(self, user_id=None, username=None, name=None, password=None, email=None, 
contact_info=None):
        """
        Initializes a user_model instance.

        Args:
            user_id: Unique identifier for the user.
            username: Username of the user.
            name: Full name of the user.
            password: User's password (should be hashed).
            email: User's email address.
            contact_info: User's contact information.
        """
        self.user_id = user_id
        self.username = username
        self.name = name
        self.password = password
        self.email = email
        self.contact_info = contact_info

    def register(self):
        """Registers a new user in the database."""
        connection = Database.connect_db()
        if connection:
            cursor = connection.cursor()
            cursor.execute("INSERT INTO users (username, password, email, contact_info) 
VALUES (%s, %s, %s, %s)",
                           (self.username, self.password, self.email, self.contact_info))
            connection.commit()
            cursor.close()
                   connection.close()
```

- **Modularity and Reusability**: The UserModel class can be reused across different parts of the application for user-related operations.
- **Coupling and Cohesion**: It has high cohesion as it focuses on user-related functionalities. Coupling is low since it only interacts with the Database module.
- **Data Coupling**: The user data is encapsulated within the class, reducing dependencies on external data structures.
- **Functional Cohesion**: All methods in the class are related to user management, ensuring functional cohesion.

3. Login ViewModel Module

```
from user_model import UserModel

class LoginViewModel:
    """Handles user login logic."""

    def __init__(self):
        self.user = None

    def login_user(self, username, password):
        """
        Authenticates a user based on username and password.
        Args:
            username: The username of the user.
            password: The password of the user.

        Returns:
            bool: True if login is successful, False otherwise.
        """
        self.user = UserModel.fetch_user_by_username(username)
        if self.user and self.user.password == password:  # Password should be hashed in
a real application
            return True
        return False
```

- **Modularity and Reusability**: This module can be reused in different parts of the application where login functionality is needed.

- **Coupling and Cohesion**: It has high cohesion as it focuses solely on login functionality. Coupling is low since it only interacts with the UserModel.

- **Data Coupling**: The user data is fetched through the UserModel, maintaining a clean separation of concerns.

- **Functional Cohesion**: The methods are specifically designed for login operations, ensuring functional cohesion.

4. Cognitive Test Module

```
class CognitiveTest:
    """Represents a cognitive test taken by a user."""

    def __init__(self, test_id=None, user_id=None, test_type=None, score=None):
        """
        Initializes a cognitive_test instance.

        Args:
            test_id: Unique identifier for the test.
            user_id: ID of the user taking the test.
            test_type: Type of cognitive test.
            score: Score obtained in the test.
        """
        self.test_id = test_id
        self.user_id = user_id
        self.test_type = test_type
        self.score = score

    def administer_test(self):
        """Simulates administering the cognitive test and assigns a score."""
        print("Administering cognitive test...")
        self.score = 95.0  # Placeholder score
        print("Cognitive test administered")
```

- **Modularity and Reusability**: This module can be reused for different types of cognitive tests.

- **Coupling and Cohesion**: It has high cohesion as it focuses on cognitive test functionalities. Coupling is low since it does not depend on other modules.

- **Data Coupling**: The test data is encapsulated within the class, reducing external dependencies.

- **Functional Cohesion**: All methods relate to cognitive testing, ensuring functional cohesion.

5. Genetic Data Module

```
class GeneticData:
    """Represents genetic data collected from a user."""

    def __init__(self, genetic_data_id=None, user_id=None, genetic_markers=None):
        """
        Initializes a genetic_data instance.

        Args:
            genetic_data_id: Unique identifier for the genetic data.
            user_id: ID of the user.
            genetic_markers: Genetic markers collected.
        """
        self.genetic_data_id = genetic_data_id
        self.user_id = user_id
        self.genetic_markers = genetic_markers

    def collect_data(self):
        """Simulates collecting genetic data."""
        print("Collecting genetic data...")
        self.genetic_markers = "APOE4"  # Placeholder genetic marker
        print("Genetic data collected")
```

- **Modularity and Reusability**: This module can be reused for collecting genetic data from different users.

- **Coupling and Cohesion**: It has high cohesion as it focuses on genetic data functionalities. Coupling is low since it operates independently.

- **Data Coupling**: The genetic data is encapsulated within the class, minimizing external dependencies.

- **Functional Cohesion**: All methods are related to genetic data collection, ensuring functional cohesion.

6. Lifestyle Data Module

```
class LifestyleData:
    """Represents lifestyle data collected from a user."""

    def __init__(self, lifestyle_data_id=None, user_id=None, diet_info=None,
exercise_info=None):
        """
        Initializes a lifestyle_data instance.

        Args:
            lifestyle_data_id: Unique identifier for the lifestyle data.
            user_id: ID of the user.
            diet_info: Information about the user's diet.
            exercise_info: Information about the user's exercise habits.
        """
        self.lifestyle_data_id = lifestyle_data_id
        self.user_id = user_id
        self.diet_info = diet_info
        self.exercise_info = exercise_info

    def collect_data(self):
        """Simulates collecting lifestyle data."""
        print("Collecting lifestyle data...")
        self.diet_info = "vegetarian"
        self.exercise_info = "daily running"
        print("Lifestyle data collected")
```

- **Modularity and Reusability**: This module can be reused for collecting lifestyle data from different users.

- **Coupling and Cohesion**: It has high cohesion as it focuses on lifestyle data functionalities. Coupling is low since it operates independently.

- **Data Coupling**: The lifestyle data is encapsulated within the class, minimizing external dependencies.

- **Functional Cohesion**: All methods relate to lifestyle data collection, ensuring functional cohesion.

7. User View Module

```
import tkinter as tk
from tkinter import messagebox



class UserView:
    """Handles the user interface for user interactions."""

    def __init__(self, view_model):
        """
        Initializes the user_view instance.

        Args:
            view_model: The view model that handles user logic.
        """
        self.view_model = view_model  # Dependency injection for the view model
        self.root = tk.Tk()  # Create the main window
        self.root.title("Alzheimer's Disease Testing Software")

        self.frame = tk.Frame(self.root)  # Frame for layout
        self.frame.pack(pady=20)

        # User Registration UI Elements
        tk.Label(self.frame, text="Name").grid(row=0, column=0, padx=10, pady=5)
        self.name_entry = tk.Entry(self.frame)  # Entry for user name
        self.name_entry.grid(row=0, column=1, padx=10, pady=5)

        tk.Button(self.frame, text="Register", command=self.register_user).grid(row=1,
columnspan=2, pady=10)

    def register_user(self):
        """Handles user registration."""
        name = self.name_entry.get()  # Get the name from the entry
        self.view_model.register_user(name)  # Call the view model to register the user
        messagebox.showinfo("Registration", "User registered successfully!")  # Show
success message

    def start(self):
        """Starts the Tkinter main loop."""
        self.root.mainloop()  # Run the application
```

Analysis

- **Modularity and Reusability**: The UserView class is modular, encapsulating all UI-related functionality. It can be reused in different contexts where user interaction is needed.
- **Coupling and Cohesion**: The class has low coupling with other modules, as it only interacts with the view_model. It has high cohesion since all methods and attributes are related to user interface management.
- **Data Coupling**: The data (user input) is passed to the view model, minimizing dependencies on external data structures.
- **Functional Cohesion**: All methods in the class are focused on user registration and interaction, ensuring functional cohesion.
- **High Cohesion and Low Coupling**: The class is designed to handle specific tasks related to user input, making it cohesive, while its dependency on the view model keeps it loosely coupled.

8. AlzheimerApp Module

```
import mysql.connector
from mysql.connector import Error

import tkinter as tk
from tkinter import messagebox
from login_view_model import LoginViewModel

class AlzheimerApp:
    """Main application class for the Alzheimer's Disease Testing Software."""

    def __init__(self, root):
        """
        Initializes the alzheimer_app instance.

        Args:
            root: The main Tkinter window.
        """
        self.root = root
        self.root.title("Alzheimer's Disease Testing Software")
        self.dashboard_frame = None  # Initialize dashboard_frame
        self.show_login()  # Start with the login screen

    def show_login(self):
        """Displays the login screen."""
        self.clear_frame()  # Clear any existing frames
        self.login_frame = tk.Frame(self.root)  # Create a new frame for login
        self.login_frame.pack(pady=20)

        tk.Label(self.login_frame, text="Username").grid(row=0, column=0, padx=10,
pady=5)
        self.username_entry = tk.Entry(self.login_frame)  # Entry for username
        self.username_entry.grid(row=0, column=1, padx=10, pady=5)

        tk.Label(self.login_frame, text="Password").grid(row=1, column=0, padx=10,
pady=5)
        self.password_entry = tk.Entry(self.login_frame, show="*")  # Entry for password
        self.password_entry.grid(row=1, column=1, padx=10, pady=5)

        tk.Button(self.login_frame, text="Login", command=self.login).grid(row=2,
columnspan=2, pady=10)

    def login(self):
        """Handles user login."""
        username = self.username_entry.get()  # Get username
        password = self.password_entry.get()  # Get password

        # Use the LoginViewModel to authenticate
        self.login_view_model = LoginViewModel()  # Initialize the view model
        if self.login_view_model.login_user(username, password):
            self.show_dashboard()  # Show dashboard on successful login
        else:
            messagebox.showerror("Login Error", "Invalid username or password")  # Show
error message

    def show_dashboard(self):
        """Displays the dashboard after successful login."""
        self.clear_frame()  # Clear the login frame
        self.dashboard_frame = tk.Frame(self.root)  # Create a new frame for the
dashboard
        self.dashboard_frame.pack(pady=20)

        tk.Label(self.dashboard_frame, text="Welcome to the Dashboard!").pack(pady=10)

    def clear_frame(self):
        """Clears the current frame."""
        for widget in self.root.winfo_children():
            widget.destroy()  # Destroy all widgets in the current frame

if __name__ == "__main__":
    root = tk.Tk()
    app = AlzheimerApp(root)  # Create an instance of the application
    root.mainloop()  # Start the application
```

Analysis

- **Modularity and Reusability**: The AlzheimerApp class is modular, encapsulating the main application logic. It can be reused or extended for different functionalities.
- **Coupling and Cohesion**: The class has low coupling with the LoginViewModel, as it only interacts with it for login purposes. It has high cohesion since all methods are related to the application's main functionality.
- **Data Coupling**: User input data is managed within the class and passed to the view model, minimizing dependencies on external data structures.
- **Functional Cohesion**: The methods are focused on specific tasks related to user login and dashboard management, ensuring functional cohesion.
- **High Cohesion and Low Coupling**: The class is designed to handle specific tasks related to the application flow, making it cohesive, while its dependency on the view model keeps it loosely coupled.