Software Design Document for Vitamin Personalizer

By Nhat-Huy Tran, Jonathan Yunes, and Ahmet Ege Aytac
Wellness Wizards
July 5th, 2025

Table of Contents

1. Introduction
1.1 Purpose
The purpose of the Vitamin Personalizer SDD Document is to detail the design and architecture of the software and its implementation. The Vitamin Personalizer takes the blood work results from the user and figures out which vitamins and diets to recommend based on the current and previous results. This document aims to explain and clarify how the software should be implemented by specifying the architectural design, components, structure, and interface of the Vitamin Personalizer.

1.2 Scope
The document will provide various diagrams, components, and design outlines of the Vitamin Personalizer, as well as explanations and overviews of the software's design and implementation of its features, functionality, and interactions with outside sources such as databases.

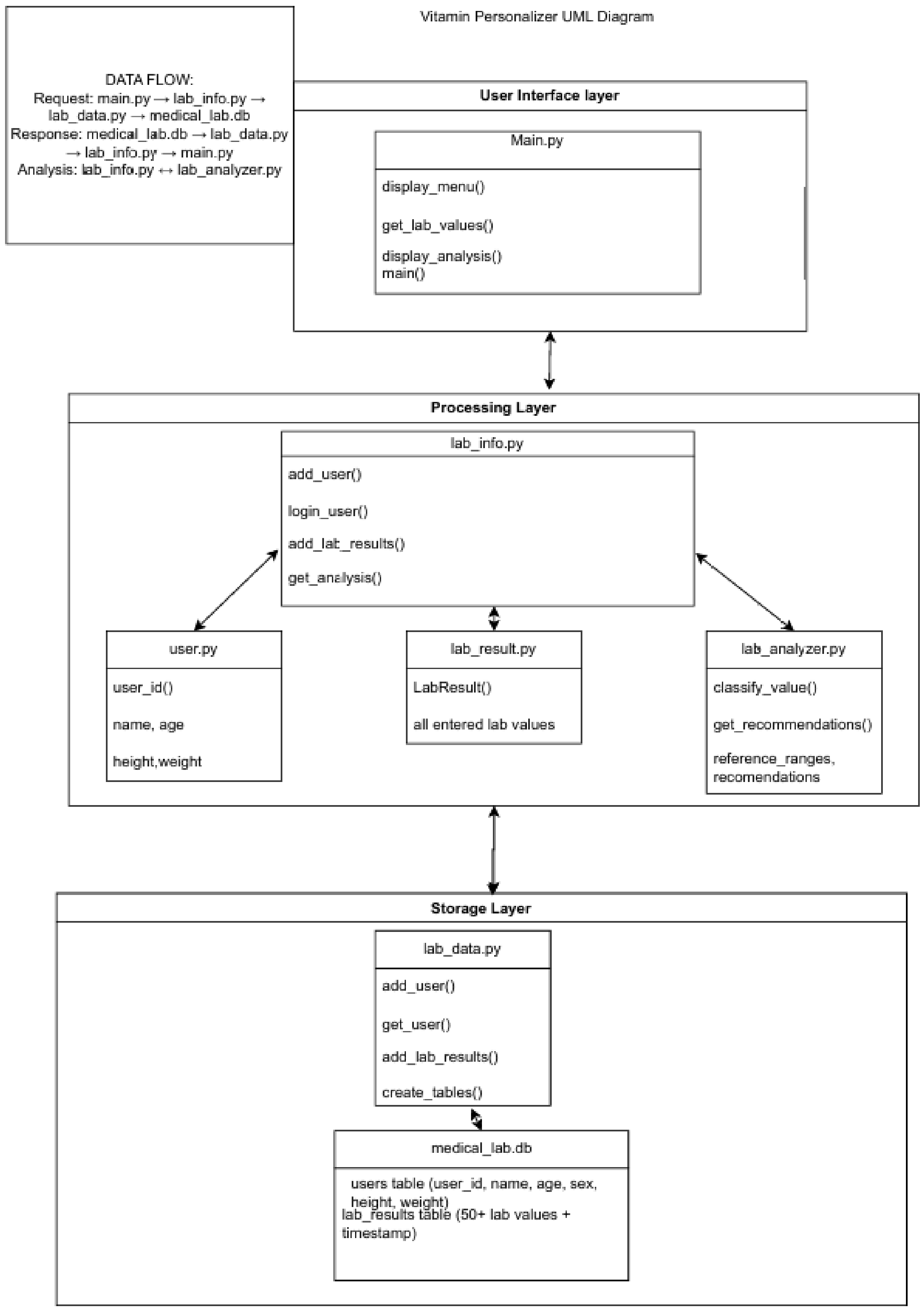
1.3 Definitions, acronyms, and abbreviations
SDD: Acronym for Software Design Development
SQL: Structured Query Language
API: Application Program Interface with a 3-layered system where main.py takes user input and shows results, the Lab Analyzer checks values and gives recommendations, and SQLite handles saving and loading the data.
User: An individual who interacts with the program to receive diet and vitamin recommendations by entering lab results.
Vitamins: Substances necessary for growth, development, and nutrition. Based on the blood work, the software recommends the person the necessary vitamins needed for growth.
Python: A Common programming language that is used to design the majority of the Vitamin Personalizer's functionality, code, and structure.

1.4 References
IEEE. (1998). IEEE Std 830-1998, IEEE recommended practice for software requirements specifications.
IEEE. (2009). IEEE Standard for Information Technology--Systems Design--Software Design Descriptions (IEEE Std 1016-2009). DOI: 10.1109/IEEESTD.2009.5167255.

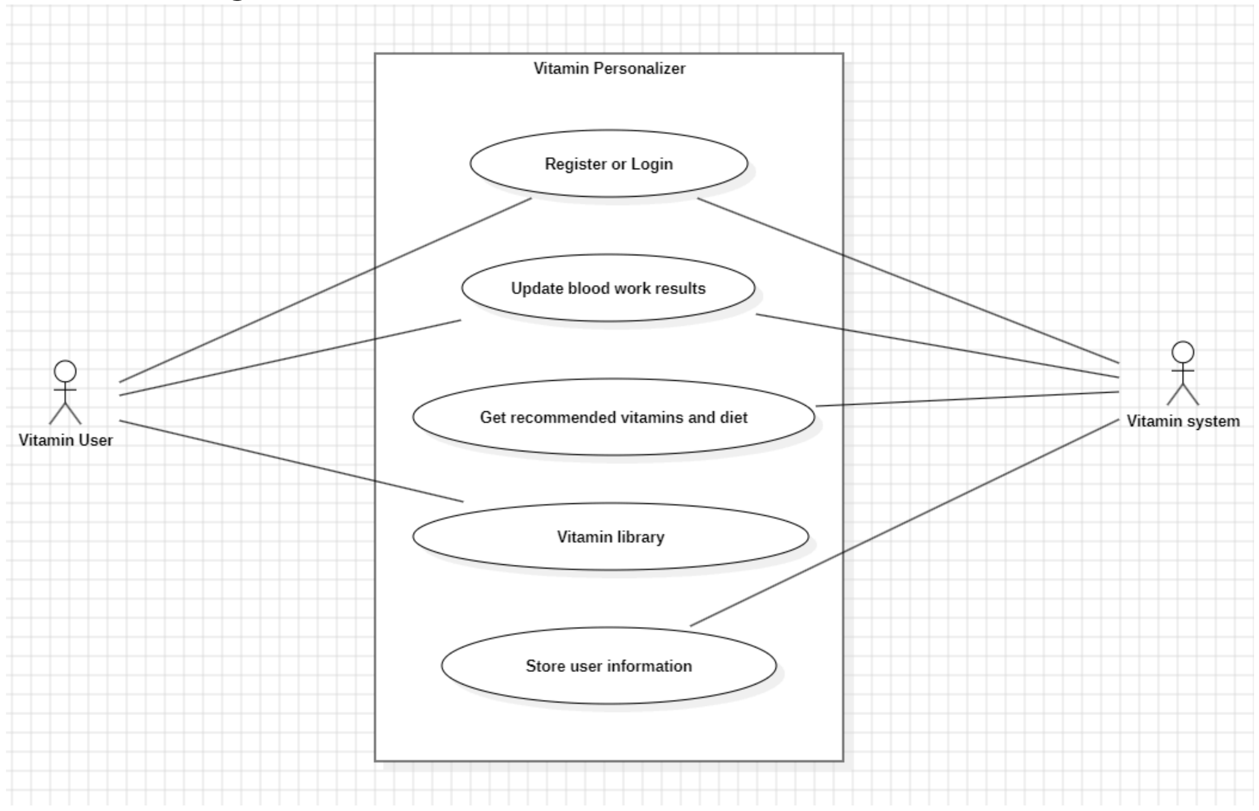
1.5 Overview
The following content in the Vitamin Personalizer SDD document will contain in-depth information about the software's architecture design, structure diagram, and components. The document will also include security measures and constraints to consider for the developers.

2. System Overview
System Architecture: The Vitamin Personalizer employs a three-tier layered architecture approach. The system uses a desktop application model with a clear separation between the presentation layer (user interface), business logic layer (analysis and processing), and data layer (local database storage). This layered architecture ensures modularity and maintainability and supports the offline operation requirement specified in the SRS.

2.1 UML Diagram for Vitamin Personalizer:



2.2 Use Case Diagram for Vitamin Personalizer:



2.3 Key Components

The system contains seven major components that handle the core functionality. The main.py component serves as the central user interface and application controller. The lab_analyzer component processes blood work analysis and generates recommendations. The lab_data component manages blood test data structures and validation. The lab_info component handles laboratory information and reference ranges. The lab_results component displays analysis outcomes and recommendations. The user.py component manages user authentication, registration, and profile data. The vitamin vocabulary component provides vitamin definitions, optimal levels, and health benefit information. These components interact through defined interfaces, with main.py coordinating user interactions while lab components handle data processing and analysis.

3. System Architecture

3.1 Architectural Design

The Vitamin Personalizer has a simple layered design. The main.py file shows menus and gets user input. The lab_info.py file handles the main work like adding users and processing lab results. The lab_data.py file saves and gets data from the database. When users enter information, it flows from main.py to lab_info.py, then to lab_data.py for storage, and back up to show results.

3.2 Module Decomposition

The system has six main parts:

- **main.py**: Shows menus, gets user input, and displays results.
- **lab_info.py**: Controls user registration, login, and lab processing.
- **lab_data.py**: Saves and loads data from the SQLite database.
- **lab_result.py**: Stores all the blood test numbers.

- **lab_analyzer.py**: Checks if lab values are normal, high, or low.
- **user.py**: Holds user information like name, age, and weight.

3.3 Design Patterns
The system uses two design patterns: Model-View-Controller (MVC) and Factory Pattern. The MVC pattern separates main.py as the View (user interface), lab_info.py as the Controller (coordinates operations), and lab_data.py as the Model (data storage). The Factory Pattern creates User and LabResult objects in lab_info.py. Each file has its own purpose for interface, coordination, data storage, and recommendations.

4. Detailed Design
4.1 main.py
- **Purpose:** The purpose of main.py is to be the main interface for all seven components combined. It allows users to navigate through menus, register themselves with unique IDs, input their blood work results, and gain vitamin and diet recommendations based on their results. Additionally, main.py allows users to re-enter or update their data.

- **Inputs and Outputs:** Main.py has many inputs and outputs. The many inputs include name, age, sex, height, weight, and BMI for user registration, number inputs to help navigate through menus, and the values for each blood work component of the user.

- **Data Structures:** Main.py works with the Lab_info.py component to help store imputed values by the user and into a class object by assigning the variable “info” to the LabInfo() class. Main.py also assigns the variables “success” and “message” from the lab_info.py component after the successful registration of the user for future use.

- **Algorithms:** Main.py has many algorithms within it that help the user navigate through the menus. For user registration, variables such as name, if the user has any characters, such as numbers, for example, it will display an error. If the user inputs an age that is below 1 or above 120, it will tell the user to enter a valid age. This is the same algorithm for height and weight, where the user must enter a valid number or it will display an error to help with accuracy and integrity. If a user has already registered, the algorithm should check for existing user IDs. If there exists one that the user has entered, it should allow the user to access the menus. Otherwise, it will give the user an error. The algorithm also checks whether the user is logged in or not by scanning for the variable current_user. If it does not exist, the algorithm should deny the user access until they are logged in. Main.py also has an algorithm for one of its functions, display_analysis(), where it will separately display normal and abnormal results acquired from the lab_analyzer.py component.

4.2 Lab_info.py
- **Purpose:** The purpose of the lab_info.py component is to process all of the information and data from the user interacting with the main.py component interface. Lab_info is also important for the functionality of the login system, and adding/removing lab data results.

- **Inputs and Outputs:** There are no inputs and outputs, as they are mainly done in main.py.

- **Data Structures:** The Lab_info.py has the most modules and components imported for its data structures, with them being user, lab_data, lab_result, and lab_analyzer. The component has various functions to communicate the user’s interaction with the system. Examples are add_user(), which obtains the information from the user and imports it to the database in the lab_data.py component; add_lab_results() and update_lab_results(), which assign the variable lab_results with the user's blood work input from the main.py interaction.

- **Algorithms:** The algorithm mainly pertains to how lab_info.py interacts with the user's blood work results inputs. The algorithm will use the class object LabInfo to obtain the user's ID and input data for each blood work component. If the user or input data does not exist, the algorithm will return none with the message that it does not exist. If there are any inputs left empty or if there are any special characters after entering the results, the algorithm will return none and give the user an error message. If the algorithm scans that the values are not None, it will acquire data from the lab_analyzer.py component and assign it to variables such as classification, recommendations, unit, and source for each input data successfully imputed. The variables are then returned.

4.3 Lab_data.py

- **Purpose:** The purpose of the lab_data.py component is to initialize the databases for both the medical database and users who have registered, and store the data after a user has completed their vitamin recommendation. Each blood work component and user should be uniquely identified within the created databases.

- **Inputs and Outputs:** There is no input or output for this component as it is mainly done in main.py. The inputs for lab_data.py should be the user's inputted data, updated data, and their existing information and ID.

- **Data Structures:** Lab_data.py uses various functions such as get_connection(), reset_table(), create_tables(), add_user(), get_user(), add_lab_results(), and more to create a database and its tables to store and update user data and lab results. Whenever a user has to update their blood work results, the old row tied to the user_id will be deleted, and the component will use add_lab_results() to input the updated data.

- **Algorithms:** There is no algorithm present.

4.4 Lab_result.py

- **Purpose:** The purpose of the lab_result.py component is to store the user's input into a class object variable to either be compared to get the vitamin and diet results or to store alongside other recorded results for progress check.

- **Inputs and Outputs:** There is no input and output for Lab_result.py as it is mainly a class object to store the user's inputted blood work results.

- **Data Structures:** Lab_result.py includes a class object named LabResult, which is used to store the user's inputted blood work result. The user can update and re-enter their results, which is why the class object is warranted.

- **Algorithms:** No algorithm is present.

4.5 Lab_analyzer.py

- **Purpose:** The purpose of the Lab_analyzer.py component is to provide the user with the correct vitamin recommendation and diet based on the values given by the user's blood work results. Each input has a unique range and the component should give the correct result depending on the value.

- **Inputs and Outputs:** The inputs are done in the main.py component. The inputs are the user's blood work results, and the output should be the vitamin and diet recommendations based on the inputs.

- **Data Structures:** The lab_analyzer.py component is created with a class object that has a list of blood work components with each having low and high ranges, and a recommendation list that provides the vitamins and diet recommendations for low values and high values.

- **Algorithms:** The component uses a classify_value() function that reads through every input for each blood work component and gets the low_threshold and high_threshold for each. The function will then loop through every blood work component with its unique thresholds and compare the results. If the inputted value is below the low_threshold variable, it will be labeled and returned as low and thus will be given the recommended vitamins and diet that is labeled that. This is the same algorithm for values above the high threshold.

4.6 user.py

- **Purpose:** The purpose of user.py is to create a class object for each unique user who has registered for the vitamin personalizer. This component is necessary for main.py to function.

- **Inputs and Outputs:** The input should be the user's input for the class variables. The output is the initialization of the object.

- **Data Structures:** Each user that is registered has a variable created under a class object named "User." Under the class definition, variables such as user_id, name, age, sex, height, weight, and BMI are stored. The class also allows each object to return its defined variables when used. Each variable under the class object will have a unique ID.

- **Algorithms:** No algorithm is present.

4.7 Vitamin_vocabulary

- **Purpose:** The purpose of the vitamin vocabulary component is to provide vitamin definitions, optimal levels, and health benefit information to the user.

- **Inputs and Outputs:** There is no input or output present.

- **Data Structures:** Each vitamin should have a unique identifier alongside a unique description, optimal level, and health benefit description.

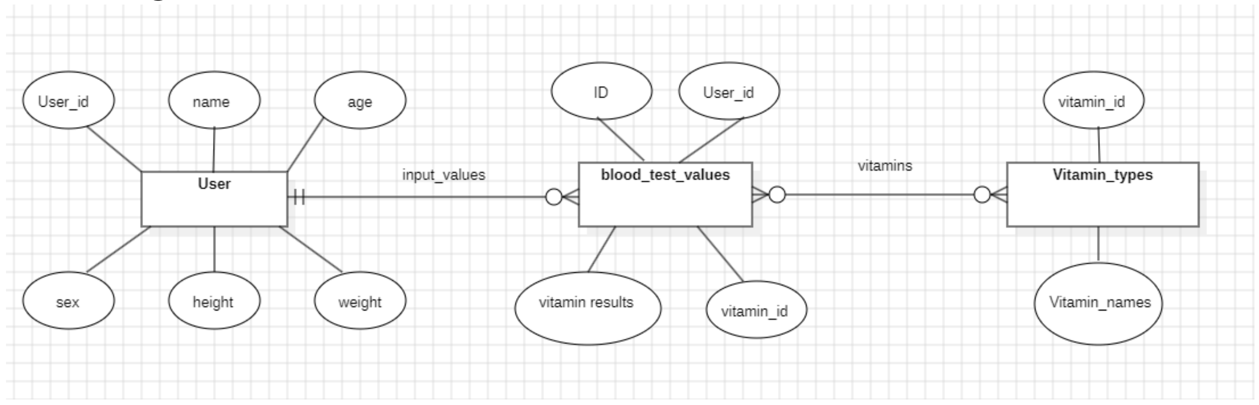
- **Algorithms:** No algorithm is present.

4.8 Interface Design:

The main.py component works with the lab_info.py component to communicate the user's interaction with the system interface, whether it's inputting account information and ID, or inputting their blood work results to be analyzed. Lab_info.py, on the other hand, works together with 4 other components with them being lab_data.py, lab_analyzer.py, lab_result.py, and user.py, to analyze and scan the user's imputed information. All 4 communicate by processing, adding, and generating the user's information, ID, and blood work results in a database, which can be updated if the user chooses to. Lab_results.py and lab_analyzer.py both contain descriptions, value ranges, and information unique to each blood work component and lab_info.py communicates by sending the inputs to both components to process and obtain the corresponding result. User.py acts as a separate list and an important component for lab_info.py and to scan, add, or remove any existing users in the database created by the lab_data.py component. Lab_data.py, lab_analyzer.py, lab_result.py, and user.py all communicate with main.py through lab_info.py as the 4 components are important functionalities, which in turn communicate indirectly with main.py. The vitamin_vocabulary component is a separate component from lab_info.py, as it contains descriptions separate from the lab_results.py and lab_analyzer.py components. It communicates through main.py, where the user can navigate through menus and select a specific vitamin or diet they want to learn about.

5. Data Design

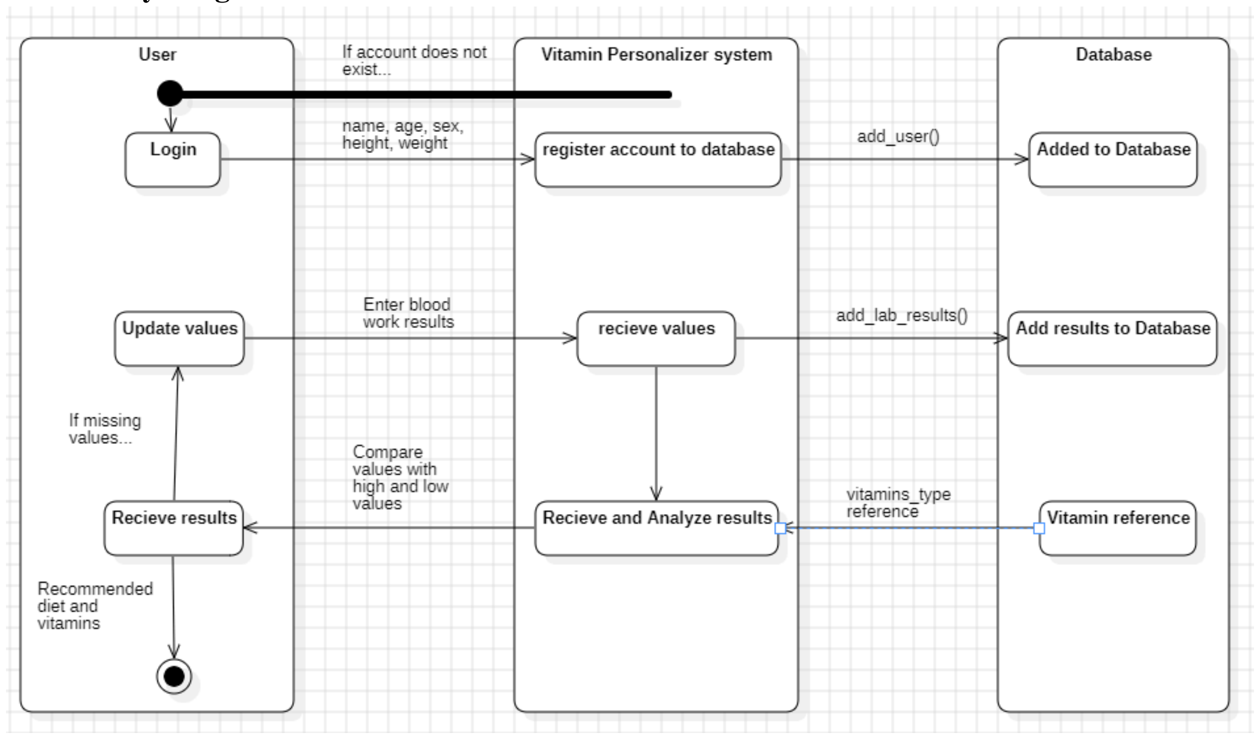
5.1 ER Diagram for Vitamin Personalizer:



5.2 Data Description:

The data design for the Vitamin Personalizer shows 3 tables used to transfer data within the system in the SQL database. The user table stores the account information of the user and uses a unique ID for each user in the table. The blood_test_values table stores all of the blood work information from all users who've entered their blood work results. Its attributes include a unique ID, the user_id as a foreign key, vitamin_results/values for the corresponding vitamin, and vitamin_id for the types of vitamins used for the value. It acts as a foreign key to the vitamin_types table. Since there will be multiple inputs and different values, the table uses a foreign key "User_id" to not only show the relation with the user table but also to have another layer of differentiation apart from each input having a unique ID. The vitamin_types table stores all of the unique vitamins separately such as red blood cell count, white blood cell count, hemoglobin, and more. The table is separate for better organization and normalization throughout the tables.

5.3 Activity Diagram for Vitamin Personalizer:



5.4 Data Flow:

Data flows from the user to the blood_test_values table. The user should also have access to their records through the progress check option. The data also flows from the vitamin_types table to the blood_test_values table to act as a reference for the vitamins used. The user table and blood_test_values table have a one-to-many relationship, as each user has many values unique to them in the blood_test_values table and the vitamin_types have a many-to-many relationship with said table due to assigning multiple vitamins to multiple blood test value inputs.

6. Security and Safety Considerations

- The vitamin personalizer system uses generated codes for user authentication, where each user receives a unique access code to log in and view their lab results. These codes are stored in SQL database columns to enable account retrieval and management. However, this approach has vulnerabilities since malicious programs could potentially break these codes through brute force attacks, trying every possible combination until they find the right one. If successful, this could lead to information leakage and compromise user data privacy.

- To address the risk of users losing their accounts permanently, the system implements two key safety measures. First, users create a security question during account setup that they must answer every time they access their lab results through the vitamin personalizer system. This provides an additional layer of protection against unauthorized access, but it doesn't help recover forgotten codes. Second, the system offers a file download feature that automatically saves the user's passcode to their device's file location as notes, preventing users from losing their access codes permanently.

- The security approach focuses on preventing complete account lockout while maintaining data protection. The combination of generated codes and security questions creates multiple protection layers, so even if hackers break the access code, they still need to answer the security question to gain access. The automatic code-saving feature addresses user error by reducing reliance on memory and manual code storage. However, if users lose both their generated code and the downloaded file, they would still face permanent account lockout since the security question alone cannot recover the account. The SQL database storage system ensures that account information is maintained, but without the proper authentication credentials, users cannot regain access to their accounts and lab results, making the code backup file download feature critical for account recovery

7. Design Decisions

- The application is built as a Python terminal program that loads in 3-5 seconds maximum per session, depending on the user's device. Users enter basic data when the application prompts for values, eliminating complex file processing that could impact performance. The system operates with one user at a time on each device through the terminal, maintaining fast response times and simple operation. Storage depends on each user's device since all data is saved to their local hard disk, requiring Windows 10 operating system, 4GB of RAM, and at least 10GB of free storage space

- The current system architecture supports one user per device instance where users must open their terminal and execute main.py to start the program. This design limits concurrent user access unlike web-based applications, but provides optimal performance for individual use cases. Future scalability would require migration to AWS cloud servers with HTTPS implementation, though this would increase operational costs compared to the current local storage model.

- The system requires Python programming language and SQL database integration for optimal functionality. The database operates on SQL to manage user information storage and retrieval. All program components must be downloaded to the user's device for terminal execution. Cloud server deployment remains a future option, though the current local version maintains cost efficiency through user-managed storage on personal devices.

- The unique ID code system was selected instead of traditional username and password authentication to provide functionality similar to institutional identification systems. This approach requires less storage space, enables faster login processing, and provides enhanced security since codes are not derived from personal information or previous passwords, making them difficult to predict or compromise.

- Security questions serve as the secondary authentication method because only legitimate users possess the knowledge to answer correctly. This solution offers cost advantages over external authenticator systems while requiring minimal storage resources and maintaining user-friendly operation.

- Python and SQL were chosen for their superior data analysis and scientific computing capabilities, directly supporting the application's primary function of vitamin analysis and health data processing. SQL integration with Python provides efficient database management for the application's data storage requirements.

- The PDF format was selected as the export standard because it ensures universal compatibility for printing across different devices and printer systems. This format maintains document integrity and provides the most reliable printable output for users.

- Alternative authentication methods including Microsoft authentication systems and email-based verification codes were evaluated against the unique ID system. The current approach was chosen for its simplicity and independence from external services. Local file storage was selected over cloud storage solutions primarily for cost reduction benefits. The decision was made to exclude automatic storage of historical test results to prevent excessive local storage consumption, though users retain the option to manually track progress through printed monthly results.