

```

sequenceDiagram
    participant Controller
    participant BaseFood
    participant Dinner

    Controller->>BaseFood: getRecipe(0, 1, 1)
    BaseFood-->>Controller: returns BaseFoods(0, 1, 1, 1)
    BaseFood->>Dinner: getRecipe(0, 1, 1)
    Dinner-->>BaseFood: returns Recipe(1)
    BaseFood-->>Controller: getRecipe(1, 1)
    Controller-->>BaseFood: returns Recipe(2)
    
```

Read in a food database consisting of three basic foods, a recipe that contains two of the basic foods and a recipe that contains the first recipe and the remaining food.

```
classDiagram
    class BasicFood
    class Recipe
    class Food
    class Log
    class DailyLog
    class Weight
    class FoodEntry
    class CalorieLimit
    class Model
    class Controller
    class View
    class Interface
    class LogDisplay
    class FoodDisplay
    class FoodController
    class LogController
    class ViewController

    BasicFood --> Recipe
    Recipe --> Food
    Food --> Log
    Log --> DailyLog
    DailyLog --> Weight
    DailyLog --> FoodEntry
    DailyLog --> CalorieLimit
    Model <--> Controller
    Controller <--> View
    Interface --> LogDisplay
    Interface --> FoodDisplay
    Controller --> FoodController
    Controller --> LogController
    Controller --> ViewController
    FoodController <--> LogController
    LogController <--> ViewController
```

```

classDiagram
    class Model {
        <<interface>>
        +getCalories() Double
        +getFat() Double
        +getCarbs() Double
        +getProtein() Double
        +getSodium() Double
    }
    class BaseFood {
        +calories: Double
        +fat: Double
        +carbs: Double
        +protein: Double
        +sodium: Double
    }
    class Recipe {
        +name: String
        +foods: List<Food>
        +servings: Integer
        +recipeFood: String
        +FILE: static final String
    }
    class DailyLog {
        +date: Date
        +weight: double
        +caloriesLimit: double
        +foodItems: List<FoodEntry>
        +entry: FoodEntry
    }
    class Exercise {
        +exerciseName: String
        +calories: Double
        +getCalories() Double
        +getFat() Double
        +getCarbs() Double
        +getProtein() Double
        +getSodium() Double
        +getCaloriesLimit() Double
        +getCaloriesIntake() Double
    }
    class FoodController {
        +FoodController()
        +addFood(FoodName: String, calories: Double, fat: Double, carbs: Double, protein: Double, sodium: Double): void
        +removeFood(FoodName: String): void
        +updateCaloriesLimit(FoodName: String, weight: Double, caloriesLimit: Double): void
        +getFoodName() String
        +removeRecipe(FoodName: String): void
        +getFoodName() String
    }
    class LogController {
        +LogController()
        +createDailyLog(FoodName: String, calories: Double, fat: Double, carbs: Double, protein: Double, sodium: Double): void
        +addFoodEntry(FoodName: String, calories: Double, fat: Double, carbs: Double, protein: Double, sodium: Double): void
        +removeFoodEntry(FoodName: String, calories: Double, fat: Double, carbs: Double, protein: Double, sodium: Double): void
        +updateCaloriesLimit(FoodName: String, weight: Double, caloriesLimit: Double): void
        +getFoodName() String
        +removeRecipe(FoodName: String): void
        +getFoodName() String
    }
    class ExerciseController {
        +ExerciseController()
        +addExercise(FoodName: String, calories: Double, fat: Double, carbs: Double, protein: Double, sodium: Double): void
        +removeExercise(FoodName: String): void
        +updateCaloriesLimit(FoodName: String, weight: Double, caloriesLimit: Double): void
        +getFoodName() String
        +removeRecipe(FoodName: String): void
        +getFoodName() String
    }
    class UserInterface {
        +addBasicFood(FoodName: String, calories: double, fat: double, carbs: double, protein: double, sodium: double): void
        +updateCaloriesLimit(FoodName: String, weight: double, caloriesLimit: double): void
        +changeWeight(FoodName: String, weight: double, caloriesLimit: double): void
        +addFoodEntry(FoodName: String, calories: double, fat: double, carbs: double, protein: double, sodium: double): void
        +removeFoodEntry(FoodName: String): void
    }
    class LogDisplay {
        +update()
        +highLog()
    }
    class FoodDisplay {
        +update()
        +drawPieChart()
        +drawGraph()
    }
    class ExerciseDisplay {
        +update()
        +drawPieChart()
        +drawGraph()
    }
    Model <|-- BaseFood
    Model <|-- Recipe
    Model <|-- DailyLog
    Model <|-- Exercise
    Model <|-- FoodController
    Model <|-- LogController
    Model <|-- ExerciseController
    Model <|-- UserInterface
    Model <|-- LogDisplay
    Model <|-- FoodDisplay
    Model <|-- ExerciseDisplay
    BaseFood --> Recipe
    BaseFood --> DailyLog
    BaseFood --> Exercise
    BaseFood --> FoodController
    BaseFood --> LogController
    BaseFood --> ExerciseController
    BaseFood --> UserInterface
    BaseFood --> LogDisplay
    BaseFood --> FoodDisplay
    BaseFood --> ExerciseDisplay
    Recipe --> DailyLog
    Recipe --> Exercise
    Recipe --> FoodController
    Recipe --> LogController
    Recipe --> ExerciseController
    Recipe --> UserInterface
    Recipe --> LogDisplay
    Recipe --> FoodDisplay
    Recipe --> ExerciseDisplay
    DailyLog --> FoodController
    DailyLog --> LogController
    DailyLog --> ExerciseController
    DailyLog --> UserInterface
    DailyLog --> LogDisplay
    DailyLog --> FoodDisplay
    DailyLog --> ExerciseDisplay
    Exercise --> FoodController
    Exercise --> LogController
    Exercise --> ExerciseController
    Exercise --> UserInterface
    Exercise --> LogDisplay
    Exercise --> FoodDisplay
    Exercise --> ExerciseDisplay
    FoodController --> LogController
    FoodController --> ExerciseController
    FoodController --> UserInterface
    FoodController --> LogDisplay
    FoodController --> FoodDisplay
    FoodController --> ExerciseDisplay
    LogController --> ExerciseController
    LogController --> UserInterface
    LogController --> LogDisplay
    LogController --> FoodDisplay
    LogController --> ExerciseDisplay
    ExerciseController --> UserInterface
    ExerciseController --> LogDisplay
    ExerciseController --> FoodDisplay
    ExerciseController --> ExerciseDisplay
    UserInterface --> LogDisplay
    UserInterface --> FoodDisplay
    UserInterface --> ExerciseDisplay
    
```

The diagram illustrates the architecture of a diet management application, organized into three main layers: Model, Controller, and View.

Model Layer:

- Food:** An abstract class defining methods for retrieving nutritional information: `getCalories()`, `getFat()`, `getCarbs()`, `getProtein()`, and `getSodium()`.
- BaseFood:** Implements the `Food` interface, storing attributes: `calories`, `fat`, `carbs`, `protein`, and `sodium`.
- Recipe:** Contains a list of `Food` objects (`foods`), the number of `servings`, a `recipeFood` string, and a static `FILE` path. It includes methods for adding/removing ingredients and updating the recipe.
- DailyLog:** Tracks daily intake with attributes for `date`, `weight`, `caloriesLimit`, a list of `FoodEntry` objects (`foodItems`), and a current `entry`. It has methods for logging and retrieving data.
- Exercise:** Represents physical activity with attributes for `exerciseName`, `calories`, and methods for retrieving and updating calorie limits and intake.

Controller Layer:

- FoodController:** Manages the `Food` database, providing methods for adding, removing, and updating food items, as well as retrieving food names and removing recipes.
- LogController:** Manages the `DailyLog`, including creating new logs, adding/removing food entries, and updating calorie limits.
- ExerciseController:** Manages the `Exercise` data, providing methods for adding, removing, and updating exercise records.

View Layer:

- UserInterface:** The main interface for user interaction, implementing methods for adding food, updating calorie limits, changing weight, adding food entries, removing food entries, and retrieving calorie information.
- LogDisplay:** Displays the daily log, with methods for `update()` and `highLog()`.
- FoodDisplay:** Displays food information, with methods for `update()`, `drawPieChart()`, and `drawGraph()`.
- ExerciseDisplay:** Displays exercise information, with methods for `update()`, `drawPieChart()`, and `drawGraph()`.

Relationships:

- The **Model** layer is the foundation, with `BaseFood`, `Recipe`, `DailyLog`, `Exercise`, `FoodController`, `LogController`, `ExerciseController`, `UserInterface`, `LogDisplay`, `FoodDisplay`, and `ExerciseDisplay` all inheriting from or implementing it.
- `BaseFood` has a directed association to `Recipe`.
- `Recipe` has directed associations to `DailyLog`, `Exercise`, `FoodController`, `LogController`, `ExerciseController`, `UserInterface`, `LogDisplay`, `FoodDisplay`, and `ExerciseDisplay`.
- `DailyLog` has directed associations to `FoodController`, `LogController`, `ExerciseController`, `UserInterface`, `LogDisplay`, `FoodDisplay`, and `ExerciseDisplay`.
- `Exercise` has directed associations to `FoodController`, `LogController`, `ExerciseController`, `UserInterface`, `LogDisplay`, `FoodDisplay`, and `ExerciseDisplay`.
- `FoodController` has directed associations to `LogController`, `ExerciseController`, `UserInterface`, `LogDisplay`, `FoodDisplay`, and `ExerciseDisplay`.
- `LogController` has directed associations to `ExerciseController`, `UserInterface`, `LogDisplay`, `FoodDisplay`, and `ExerciseDisplay`.
- `ExerciseController` has directed associations to `UserInterface`, `LogDisplay`, `FoodDisplay`, and `ExerciseDisplay`.
- `UserInterface` has directed associations to `LogDisplay`, `FoodDisplay`, and `ExerciseDisplay`.

```

sequenceDiagram
    participant User
    participant Interface
    participant Controller
    participant Model
    participant LogicData as Logic/Data
    participant Log

    User->>Interface: Add Food to Log
    activate Interface
    Interface->>Controller: AddFoodToLog(foodName, servings)
    activate Controller
    Controller->>Model: FindFood(foodName)
    activate Model
    Model->>Controller: GetCurrentDate()
    deactivate Model
    Controller->>LogicData: AddServingsToLog(food, servings, currentDate)
    activate LogicData
    LogicData->>Controller: ->>currentLog
    deactivate LogicData
    Controller->>Model: GetFoodInfo()
    activate Model
    Model->>LogicData: GetFoodInfo()
    activate LogicData
    LogicData->>Model: ->>currentLogInfo
    deactivate LogicData
    Controller->>LogicData: AddFoodToLog(foodName, servings, currentDate)
    activate LogicData
    LogicData->>Log: AddFoodToLog()
    activate Log
    Log->>LogicData: ->>currentLogInfo
    deactivate Log
    LogicData->>Controller: ->>currentLogInfo
    deactivate LogicData
    Controller->>Interface: ->>currentLogInfo
    deactivate Controller
    Interface->>User: ->>currentLogInfo
    deactivate Interface
  
```

Add two servings of a food to the log entry for the current date.

```

sequenceDiagram
    participant Controller
    participant View
    participant Logic
    participant Model
    participant Persistence
    participant Network

    Controller->>View: Request for current day's log
    Note over View: Retrieve log for current date
    View->>Logic: Log data for current date
    Note over Logic: Retrieve basic food data
    Logic->>Model: Basic food data
    Note over Model: Retrieve recipe data
    Model->>Persistence: Recipe data
    Note over Persistence: Calculate calories for basic food
    Persistence->>View: Calories calculated
    View->>Logic: Calculate calories for recipe
    Note over Logic: Get national information for ingredient 1
    Logic->>Model: National information for ingredient 1
    Note over Model: Get national information for ingredient 2
    Model->>Persistence: National information for ingredient 2
    Persistence->>View: National information
    View->>Logic: Calories calculated
    Note over Logic: Total calories for current date
    Logic->>Controller: Total calories for current date
  
```

Compute the total number of calories for the current date, assuming the log consists of a basic food and a recipe consisting of two basic foods.

Controller: Acts as the intermediary between the View (UI) and the Model (data). It processes user inputs from the View and translates them into actions to be performed by the Model. Responsibilities include fetching basic food items, generating recipes, and adding food to a log based on user requests.

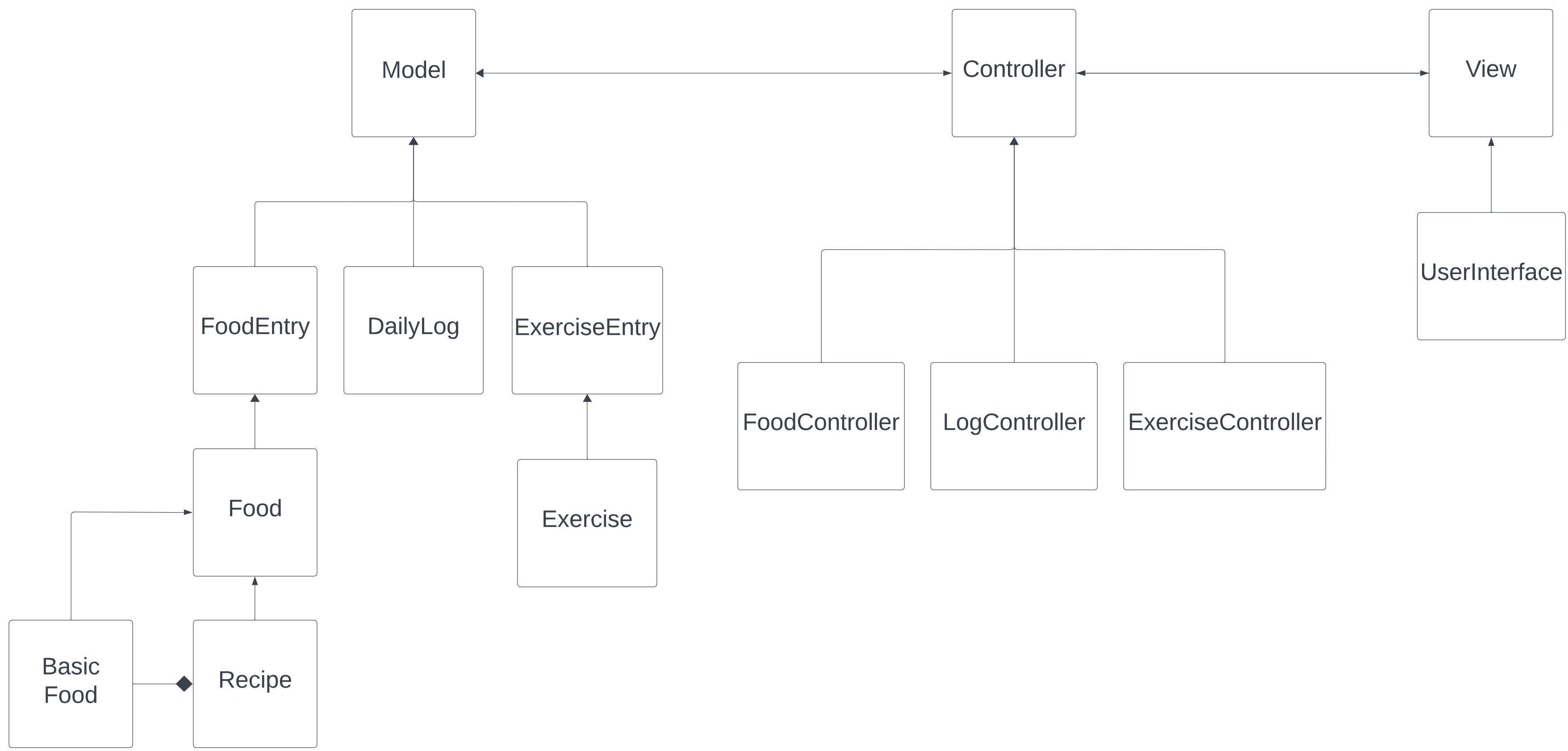
Model: Represents the system's data structure and logic for handling food items, recipes, and nutritional information. It is responsible for storing and manipulating the data, such as adding servings to a log, calculating calories, and retrieving nutritional information for both basic foods and recipes.

Log: Manages entries related to food consumption. It can add servings to a log, refresh the log display, and retrieve log entries for a specific date. Its primary responsibility is to track the user's food intake over time.

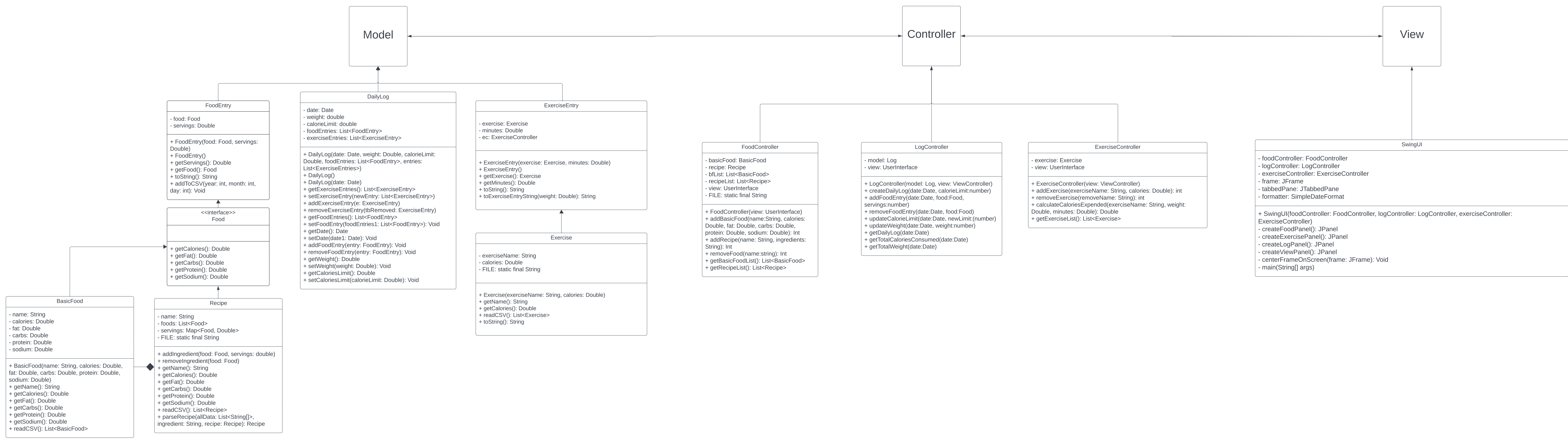
Basic Food and Recipe: These classes are part of the food database. Basic Food represents individual food items with their nutritional data, while Recipe represents a combination of these items into a meal, including the calculation of total nutritional values from its ingredients.

The organization of the system into separate classes for handling the UI, data processing, and data storage (Model-View-Controller or MVC architecture) allows for a clear separation of concerns, making the initial implementation more straightforward and future maintenance or upgrades easier to manage. This separation enhances the system's scalability and adaptability, as changes to one component (e.g., the user interface design) can be made independently of the others (e.g., the data model). However, this approach may introduce complexity in managing the interactions between components, especially for large-scale applications, and could require more effort in the initial design phase to ensure that all components communicate effectively.

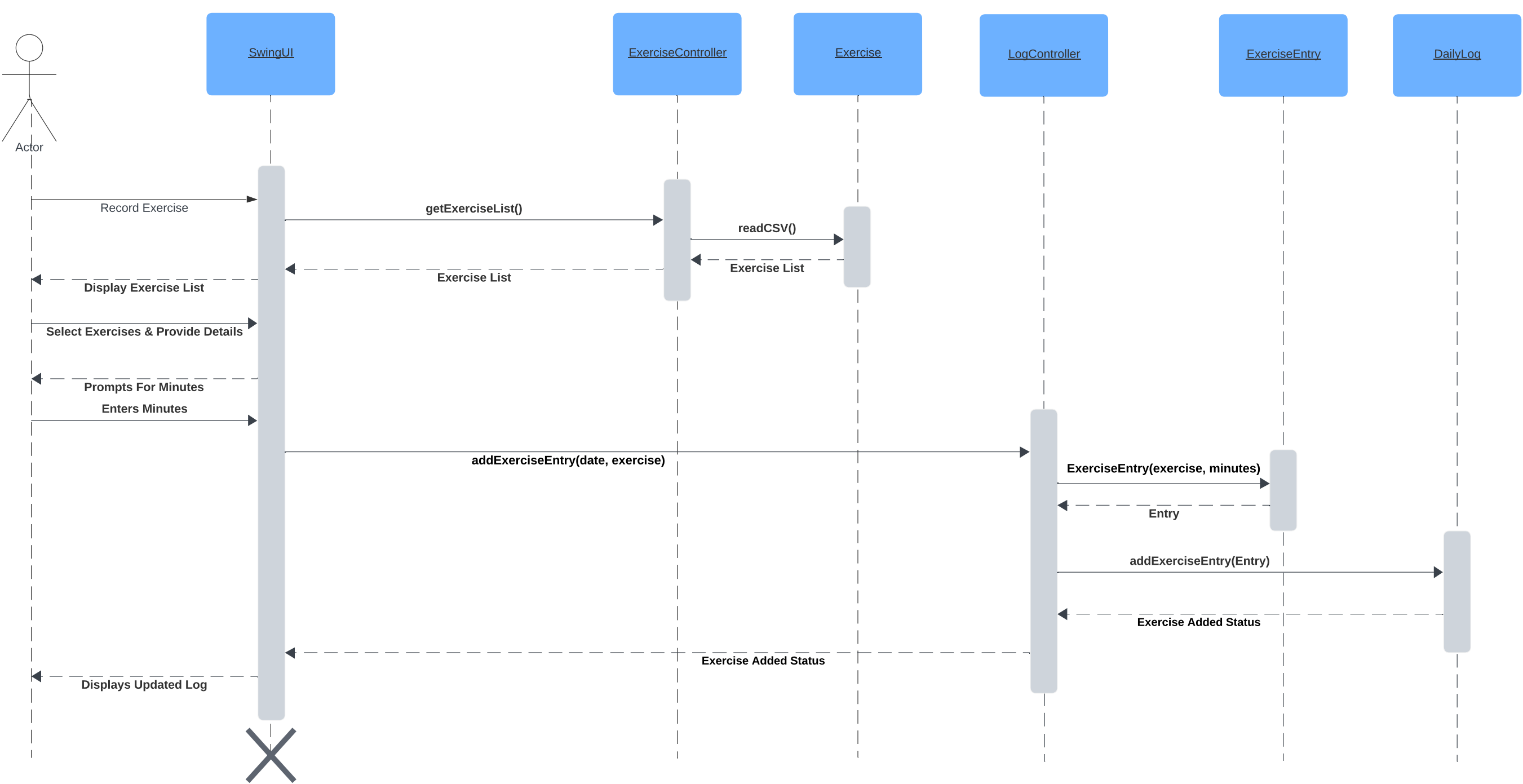
Domain Diagram



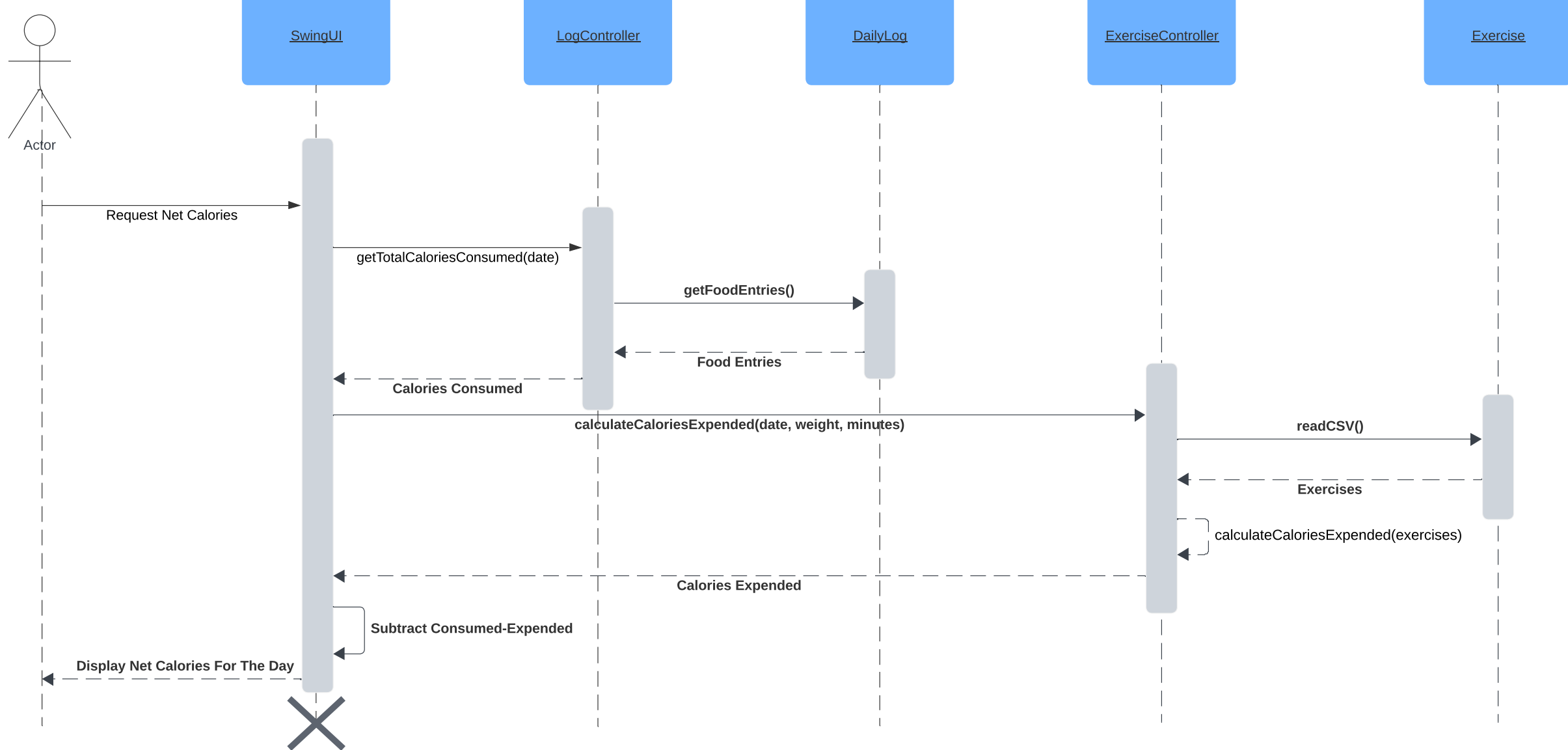
Class Diagram



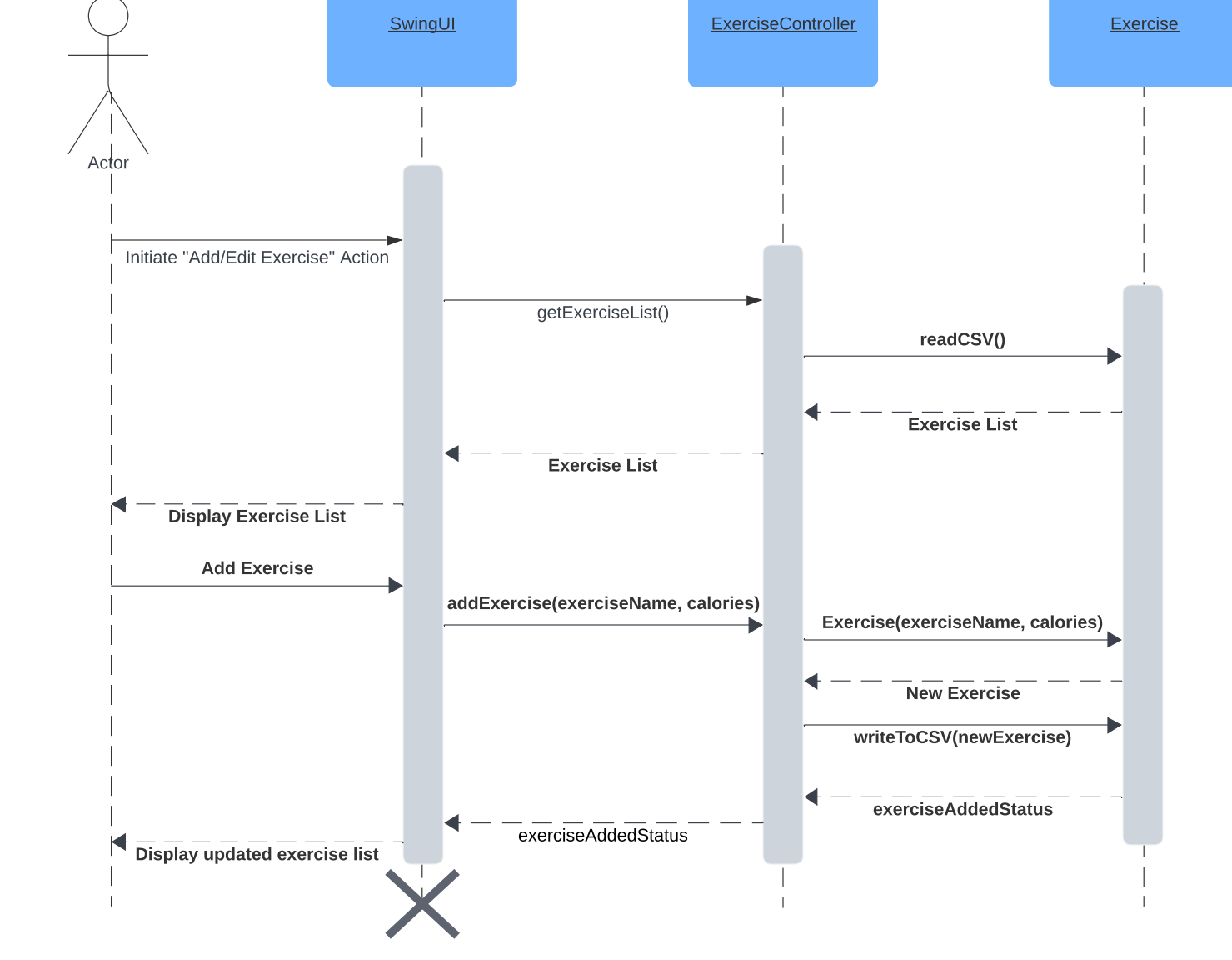
Recording Exercises



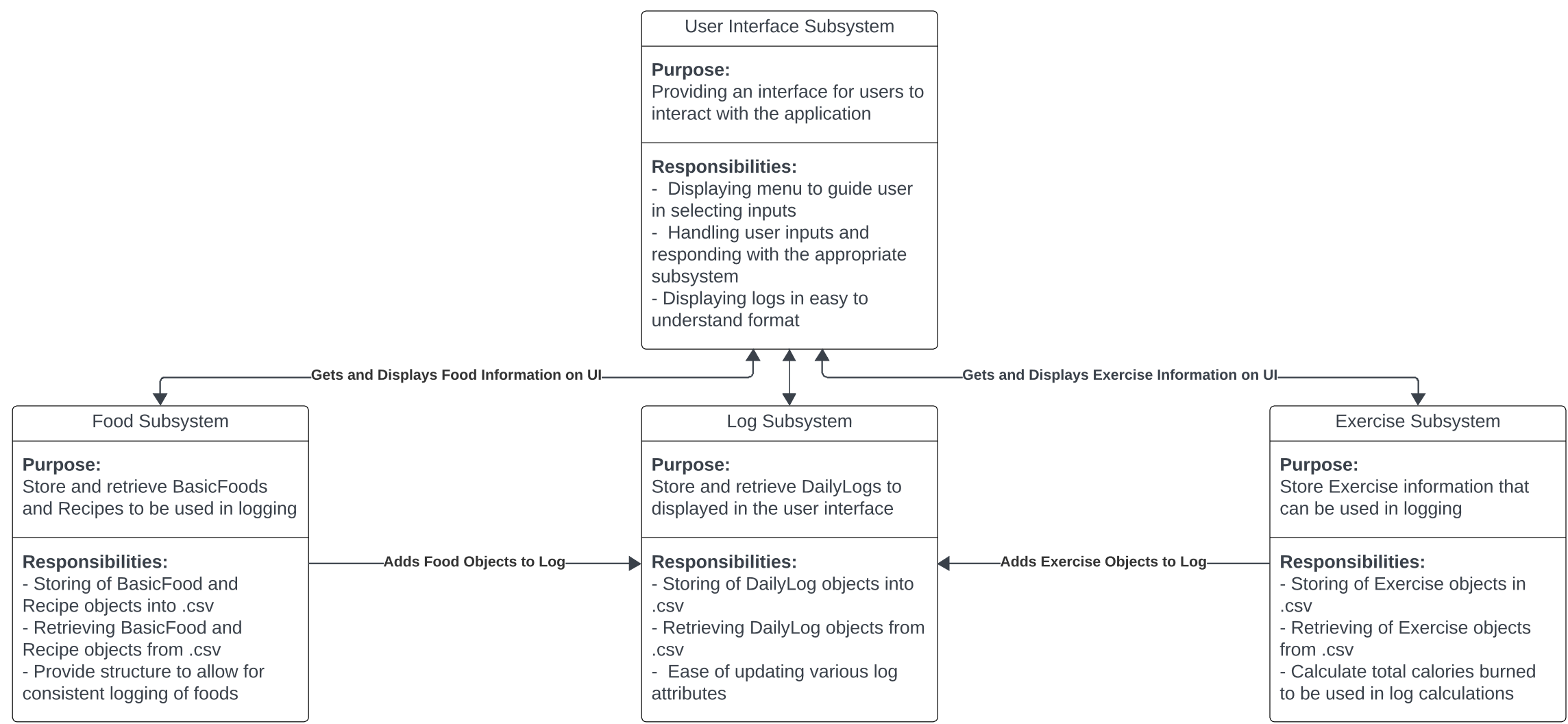
Subtracting calories depleted from calories consumed in order to compute net calories per day.



User adds to/edits the collection of exercised provided.



Subsystem Structure



Scrap this on the right

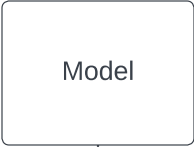
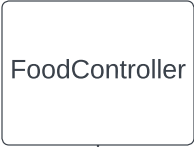
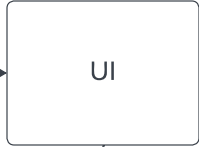
Must have something to
do with FoodEntry
objects

List<FoodEntry>

iterate through
FoodEntry list

for(food in FoodEntry)

- Person clicks add recipe
- UI prompts for recipe name
- UI displays options:
 - Create New Basic Food
 - Add Existing Basic Food
 - Add Existing Recipe
- Person selects "Create New Basic Food"
- UI opens Add Basic Food window
- Person enters basic food information (name, calories, protein ,etc)
- UI prompts for how many servings
- UI stores basic food name and servings in string
- UI ALSO sends basic food information to FoodController
- FoodController add basic food to food.csv
- UI displays add successful
- UI displays options again:
 - Create New Basic Food
 - Add Existing Basic Food
 - Add Existing Recipe
- Person selects "Add Existing Basic Food"
- UI calls FoodController to grab list of Basic Foods
- FoodController notifies Model (Basic Food) to update itself and return
- Basic Food reads in .csv and returns List<BasicFood>
- FoodController reads through List<BasicFood> and updates view with list of basic food
- UI displays list of basic foods for user to choose from and how many servings
- Person selects "Apple" from list and puts 2 servings
- "Apple" and servings are appended to ongoing recipe String
- UI displays options again:
 - Create New Basic Food
 - Add Existing Basic Food
 - Add Existing Recipe
- Person selects "Add Existing Recipe"
 - Repeat "Add Existing Basic Food" process again
- UI displays list of basic foods for user to choose from and how many servings
- Person selects "PB&J" from list and puts 3 servings
- "PB&J" and servings are appended to ongoing recipe String
- Person selects finish recipe
- Ongoing recipe String is then sent to FoodController



- Person opens program
- UI will call FoodController.java to return both Recipe.java and BasicFood.java from the foods.csv
- FoodController.java calls both ReadCSV methods of model objects

- find r
- if r, create recipe object
- if Pizza Slice, need to go through csv and look for pizza slice, if exists, create basic food out of pizza slice
- take next number next to pizza slice and use the addIngredient function
- Put recipe in array of recipes