The A Team

Diet Manager V2.0 Presentation

Members

Brianna Buttaccio

Caitlyn Daly

Tiffany Ellis

Roy Tran

Fu Quan Li

Kevin Reynolds

Summary

- Allows a user to keep track of personal health information
- Track exercise, food intake, weight, and calorie intake goal
- Update their information on a daily basis

Design Patterns

Design Pattern Summary

- MVC
  - Why: There is a clearly defined separation between components of a program
  - Why: Leads to lower coupling and higher cohesion
- Observer / Observable Pattern
  - Why: Allows us to have data updated as the user is inputting data
- Composite Pattern
  - Why: Allows us to compose our objects into a tree structure to create our food, basic food and recipe objects

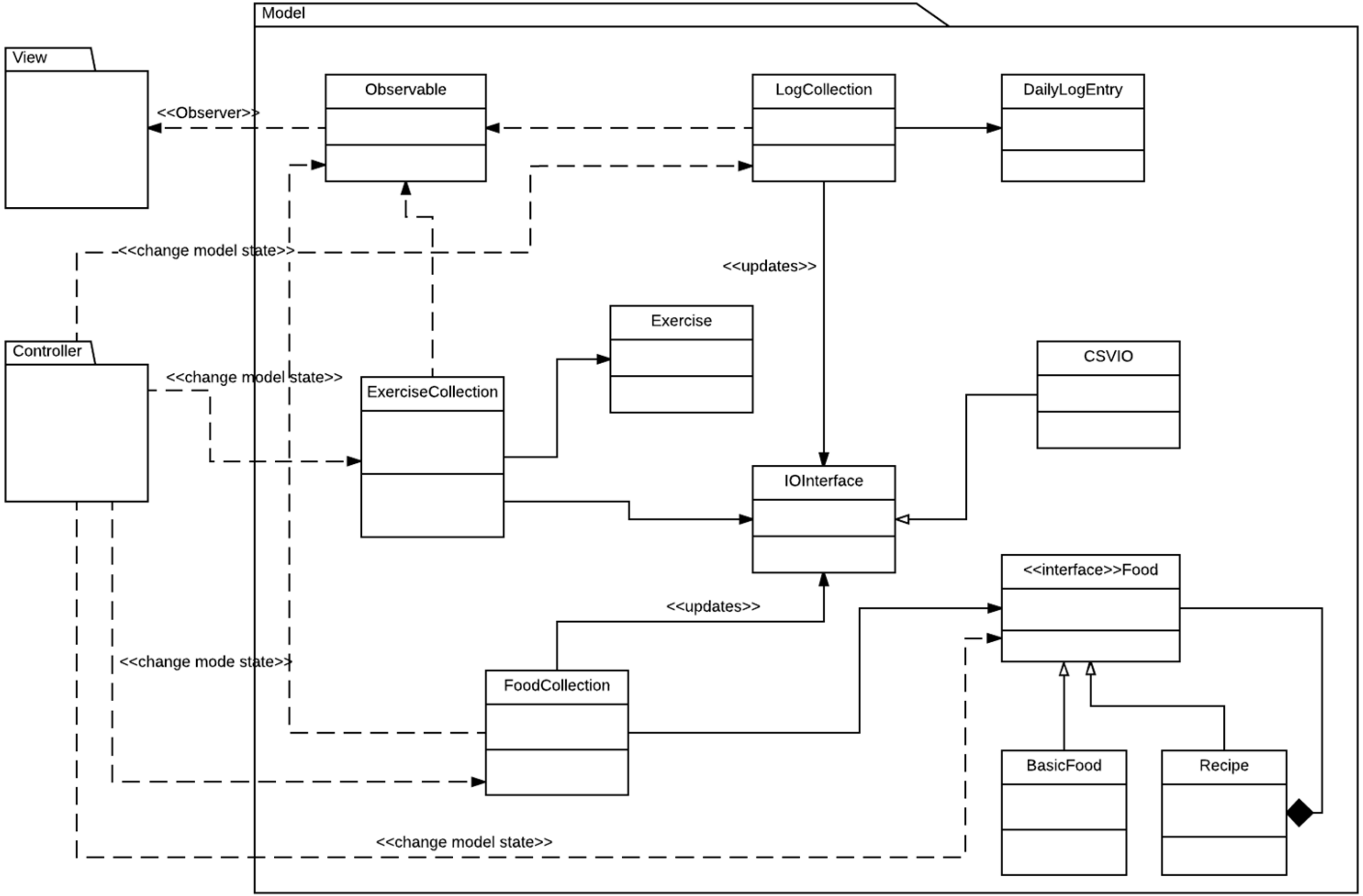
Design Pattern Summary

- Programming to Interface
  - Why: Allows us to increase the maintainability of the code
  - Why: When classes are added in the future, a template for their behavior is already defined and the rest of the code does not need to be updated
- Dependency Injection
  - Why: Reduce coupling - high level code can receive lower level code that it can call down to

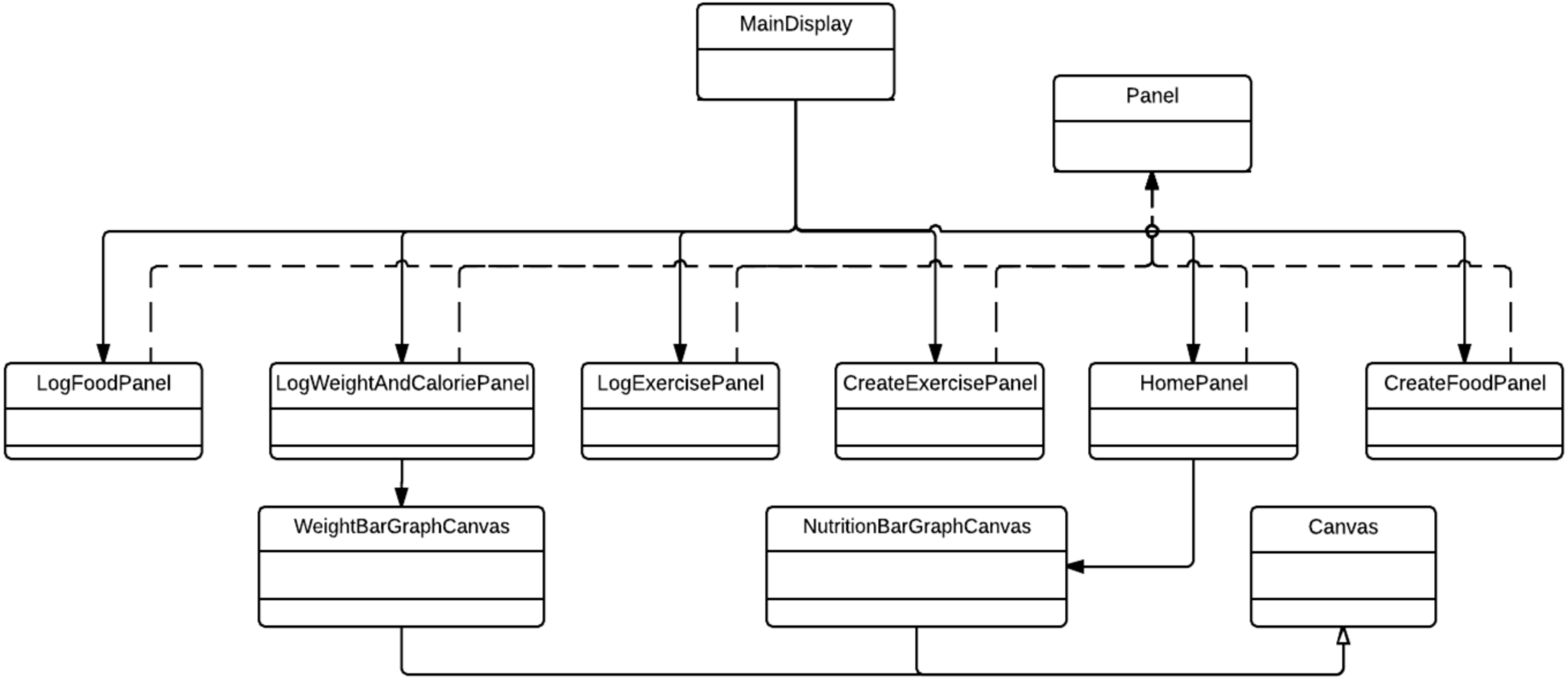
Approach - MVC Pattern

- **Model** - Each class is in charge of recording their respective data. The model uses the Observable pattern so that the view can update the display when data changes in the model. All of these files are use the controller to handle the logic and business rules of changing data.
- **View** - It will know about and be able to call on other classes that will act as widgets that can be reused in different parts of the program. Everything is split into tabs for easy navigation, and all panels are programmed to our Interface panel.
- **Controller** - The LogCollectionManager, FoodCollectionManager, ExerciseCollectionManager are classes that act as our controllers. They will handle all of the logical functionality that comes with the user's nutritional, daily log data and food and recipe data respectively.

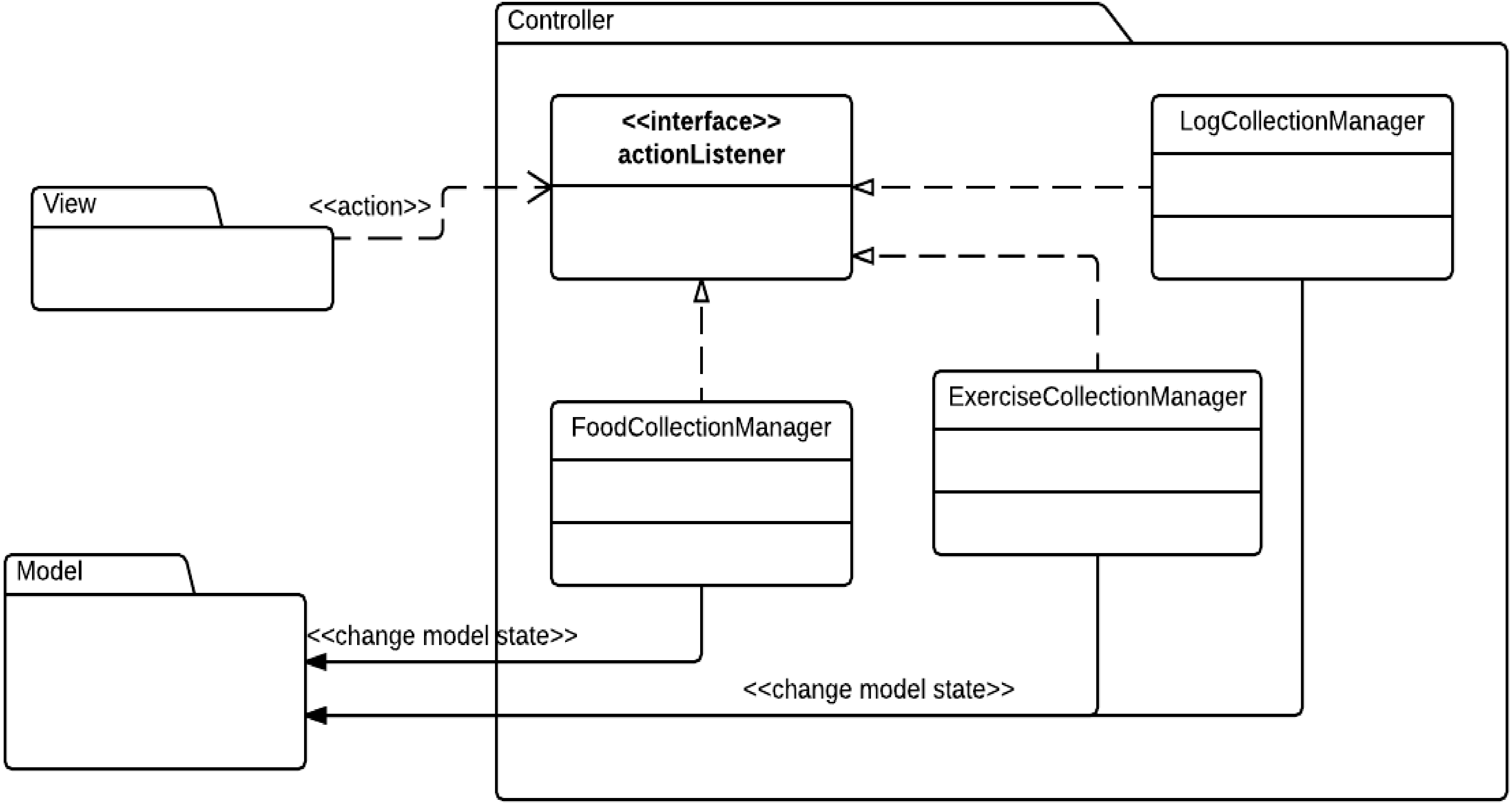
Model


Model
View
Observable
LogCollection
DailyLogEntry
<<Observer>>
<<change model state>>
<<updates>>
Exercise
CSVIO
Controller
<<change model state>>
ExerciseCollection
IOInterface
<<interface>>Food
<<updates>>
<<change mode state>>
FoodCollection
BasicFood
Recipe
<<change model state>>

View


MainDisplay
Panel
LogFoodPanel
LogWeightAndCaloriePanel
LogExercisePanel
CreateExercisePanel
HomePanel
CreateFoodPanel
WeightBarGraphCanvas
NutritionBarGraphCanvas
Canvas

Controller


Controller
View
<<action>>
<<interface>>
actionListener
LogCollectionManager
FoodCollectionManager
ExerciseCollectionManager
Model
<<change model state>>
<<change model state>>

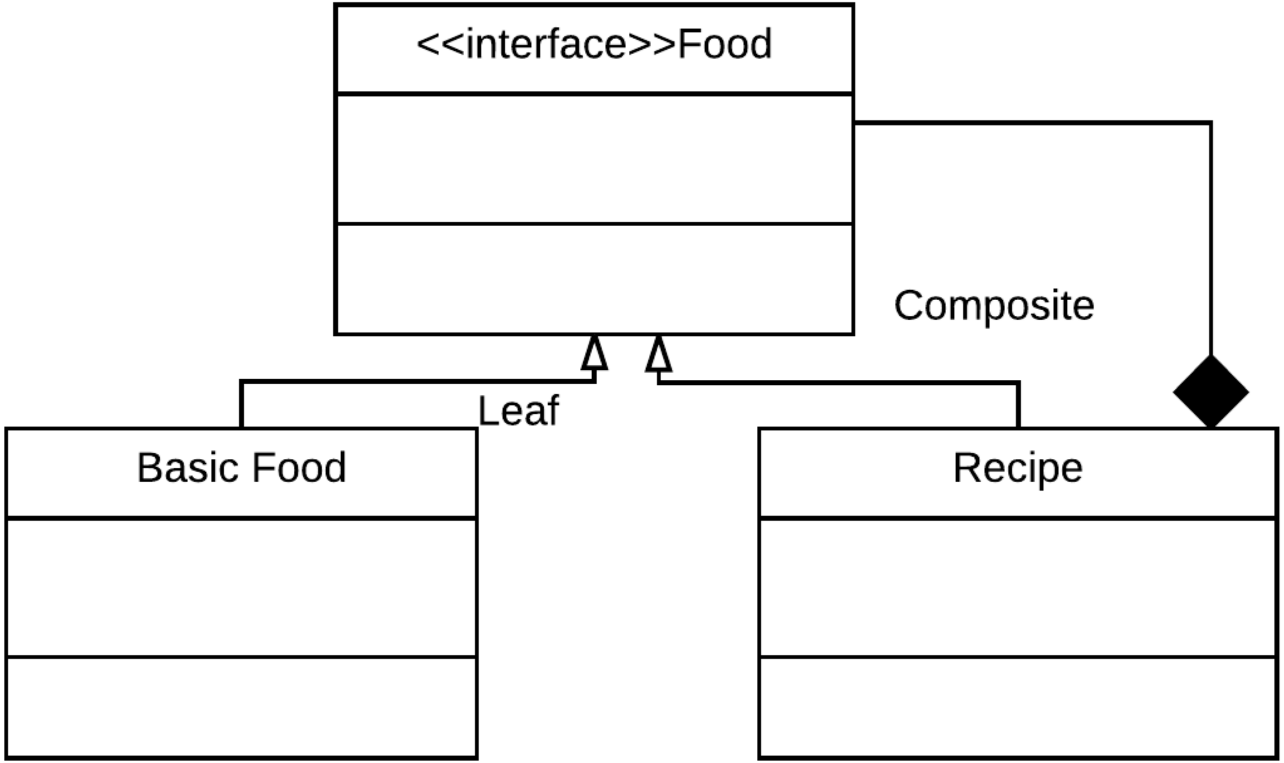
Design Patterns: Observer / Observable

- Observer
    - Any object that wishes to be notified when the state of another object changes
    - View - Observes the data and wants to know when the data changes
- Observable
    - Any object whose state may be of interest, and in whom another object may register an interest
    - Model - Will notify the observer when any of the data changes

Design Patterns: Composite Pattern

Composites that contain Components, each of which could be a Composite.

- Component
    - declares the interface for objects in the composition
    - Food Class
- Composite
    - Has children and implements methods to manipulate children
    - Recipe Class
- Leaf
    - implements all Component methods
    - Basic Foods class


<<interface>>Food
Composite
Leaf
Basic Food
Recipe

Dependency Injection

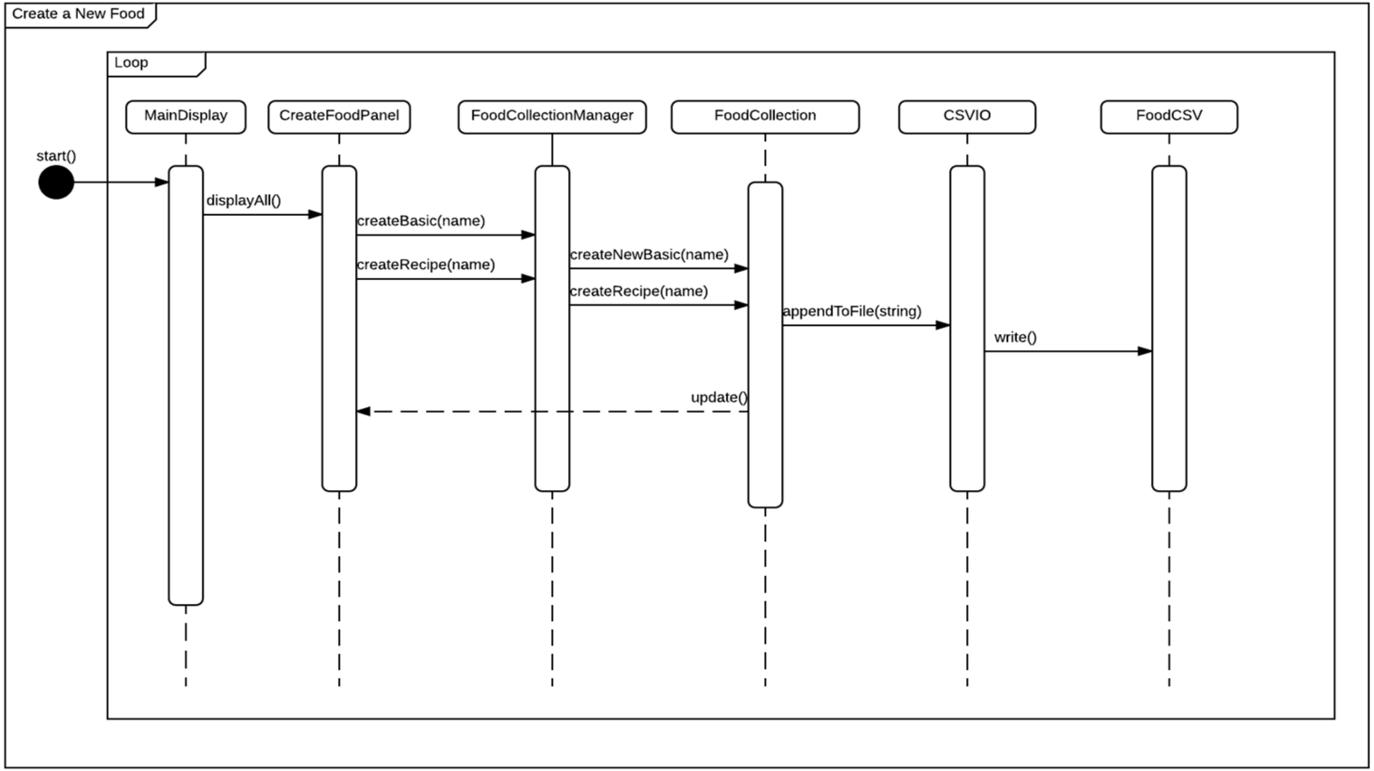
```
public UserMain(){
    foodCollectionObj = new FoodCollection(foodFileName);
    excerciseCollectionObj= new ExerciseCollection(exerciseFileName);
    logCollectionObj= new LogCollection(userFileName, foodCollection,
exerciseCollection);
        display = new MainDisplay(foodCollection, exerciseCollection,
logCollection);
}
```

Design Patterns: Programming to Interface

When possible, one should refer to a more abstract level of a class instead of referring to a concrete implementation.

- IOInterface
    - CSVIO programmed to IOInterface
    - Allows us to change the way the data is stored easily
    - CSVIO - Takes care of reading in and writing to the csv
- Panel
    - Interface for our view
    - Interfaces for all of the separate panels that display information to make it easy to create a JButton, JLabel, and JTextField

Sequence Diagram - adding recipe to Food Collection



Our Experience

Some of our Struggles

- Not following the Design when Coding
  - Forgot to use the controllers
- Keeping Track of tasks and dividing them well
  - People were working on the same tasks
- We did not always complete tasks in a logical order
  - We often needed classes in the model to get the data in the view
- Had a hard time getting the date function to work
  - It was difficult to change the date and maintain the selector
- Recursion in recipe
  - Calculating calories with servings for recipes inside of recipes inside of recipes

Design Decisions : The Good

- Implementation of the Observer Pattern
  - Made it easy to get most recent data in the view
- Programming to Interface for CSV
  - Makes it easy to change data storage method (CSV, JSON, or XML)
- Programming to Interface on our View
  - Created reusable functions for all view classes
- Use of the Hashmap to store data by date
  - Made it efficient to get the daily log for a specific day which we used frequently

Design Decisions: The Bad

- Originally were not planning on using the MVC pattern
  - We needed a way to update the view (observer / observable)
  - Has the kind of separation we needed to implement observer/observable
- Originally we had just an IO class that took care of everything having to do with the csv and io, however this ended up turning into a god class that could see everything, so we separated it and designed to an IO Interface.
- Naming Conventions
- Separation of concerns was so high it was sometimes difficult to keep track of where things were happening

Demo Time!