BoozeTracker Final Report - Project #20

Nathan Bellowe
Nathan Mckenna
Yash Parekh

Demo available at nbellowe.com/BoozeTracker/ionic-lab/

Project Summary:

What is the high-level overview of your semester project. What are you trying to accomplish? What will your system do when you are done?

Our application serves to track a user's alcohol consumption and estimate levels of intoxication. A user will click a button every time they consume a drink, and be notified of the amount of drinks that they've had, with a cool graphic. The user will also be able to view previous drinking history. This will be useful to allow for proper analysis and planning around drinking habits.

Implementation Note:

This was written utilizing several tools, including Typescript as the programming language, AngularJS for data binding, Ionic to build and style a hybrid application, and SASS to write CSS.

1. List the features that were implemented (table with ID and title).

Business Requirements	
BR-00	The app must prominently display that this holds no legal guarantee of accuracy.
→	This is accomplished through two user experience decisions. First, the user is prompted with a display on opening the app to confirm their age, if they confirm that they are over 21, then the user is prompted with a notification window that notifies them of the inaccuracy of the app, and to drink responsibly.
BR-01	The app must require users to be at least 21 years old
→	On opening the app the user is prompted with a display to confirm their age. If they select no, the app will close.

Non-Functional Requirements	
NF-00	Performance: Upon having a drink, it should be recorded in the database in order to use persist the data.
→	An IndexedDB database connected through a library stores drink history, and user configuration.
NF-02	The user interface must be understandable, and simple to use.
→	We utilized Ionic in order to make a hybrid application. This allowed us to easily craft a neat interface in HTML.
NF-03	The application must handle, catch, and hide, exceptions without compromising the user's experience.
→	The application has error handling screens when deployed.
NF-04	The drink filling animation must be smooth.
→	The drink filling animation uses hardware-accelerated CSS3 animations, and performs great.

User Requirements	
UR-00	As a user, I want my estimated BAC to be accurately calculated based on my height, weight, and gender.
→	The settings tab allows a user to input their personal characteristics. These are persisted to the database, and used to calculate the BAC.
UR-01	As a user, I want to be able to click the “I had a drink” button so that my stored BAC estimate is updated accordingly.
→	The home tab has an “Add Drink” button which can be clicked. This presents the user with a dialog to specify the type of drink that was consumed. Upon confirmation of the drink information, the drink is stored in the database. When the BAC is recalculated, which happens once per second, the BAC algorithm uses the most recent drinks to estimate the BAC using ??? formula...
UR-02	As a user, I need to be able to view the number of drinks I’ve had on a previous night, so I can know how many drinks I had on a given night
→	The history tab has a plot with the number of drinks consumed per day for up to the last 31 days. Each datapoint can be selected to display that date and number of drinks. In the future, this requirement can be expanded by adding a variable date range, along with different data, such as number of ounces of alcohol per day, or average drink strength.
UR-03	As a user, I need to be able to see the estimate of my current BAC both visually and numerically so that I can track intoxication level.
→	The home page has both a BAC estimate as a number, and a “beer glass” that is filled to indicate a similar metric that is optimized for animation quality.
UR-04	As a user, I need to be able to enter information about my size and gender to better approximate BAC.
→	As described above, the settings tab allows a user to input their personal characteristics.
UR-05	As a user, I want to be able to view my drinking history through a graph depicting number of drinks vs. days.
→	As described in detail above, the history tab has a plot with the number of drinks consumed per day for up to the last 31 days.
UR-06	As a user I expect to be notified when I have had too much alcohol
→	After a user has pressed the add drink button, the drink is added to the database and the BAC estimate is recalculated. If the user’s current BAC estimate is

	greater than 0.1% then an alert is displayed informing the user that they have had too much to drink.
UR-07	As a user, I want to be able to clear the history of how many drinks I've had, so I can remove drinking history
→	On the settings tab there is a button that allows the user's drinking history to be erased. Once the button is pressed the user's history is deleted from the database and is no longer referenceable from the history tab. The clear button also resets the user characteristics to defaults.

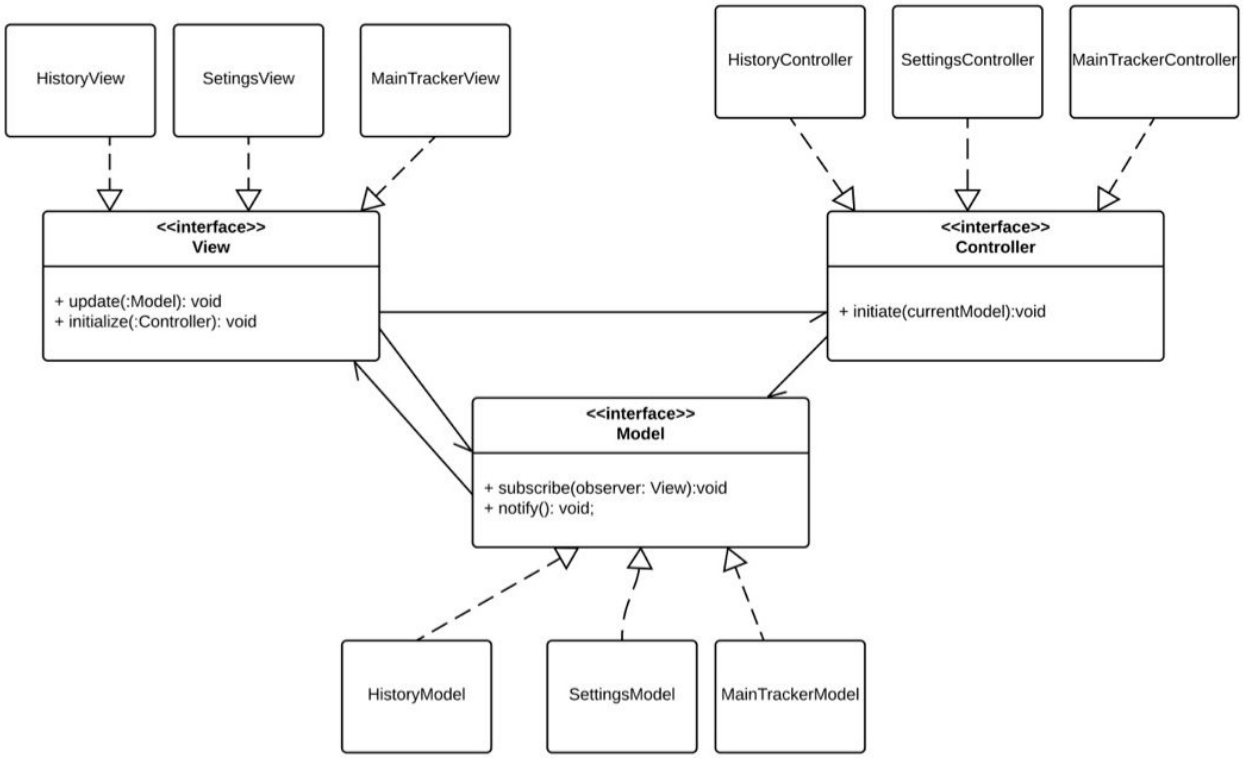
2. List the features were not implemented from Part 2 (table with ID and title).

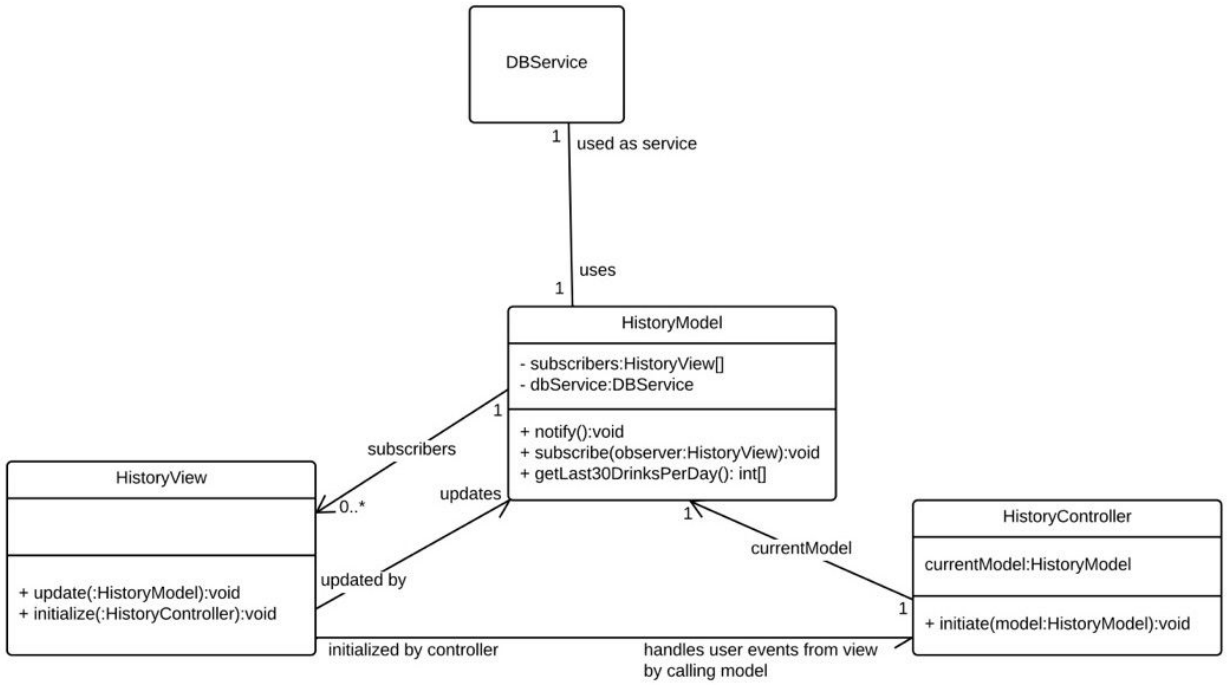
Non-Functional Requirements	
NF-01	Works on different platforms the same. (IOS/Android)
→	While the program is built for all platforms, the user interfaces have slight differences, particularly in the settings screen. In order to finish this requirement, time needs to be put into styling utilizing a combination of Ionic HTML component changes, and CSS styling.

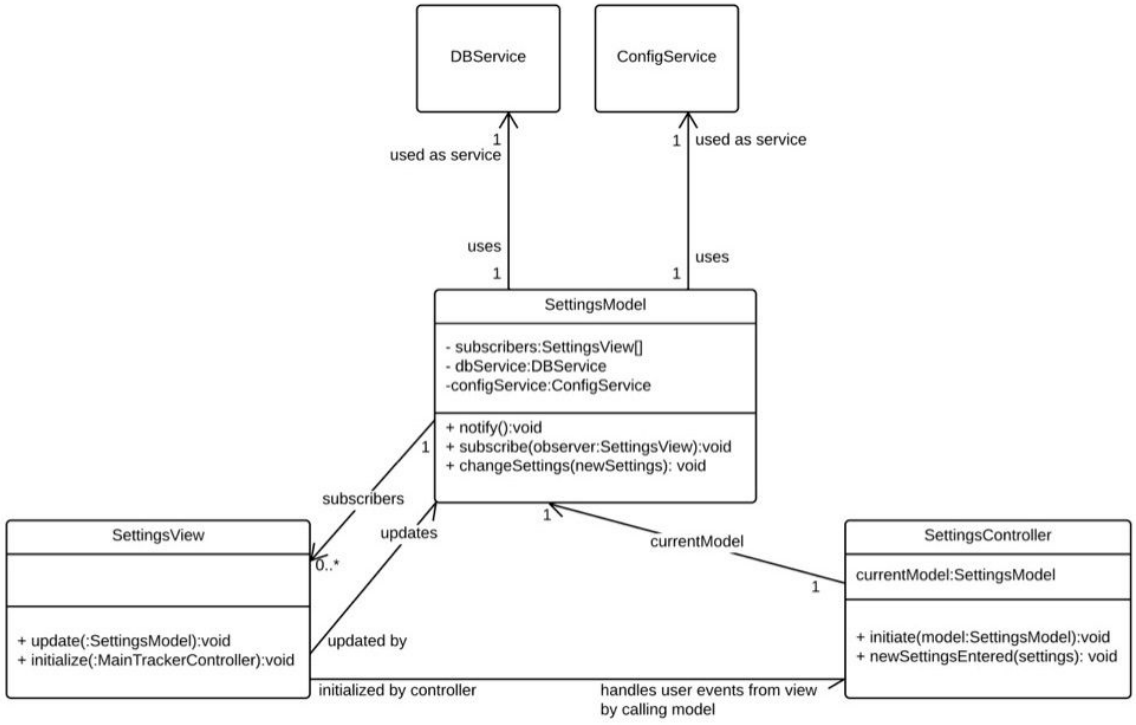
3. Show your Part 2 class diagram and your final class diagram. What changed? Why? If it did not change much, then discuss how doing the design up front helped in the development.

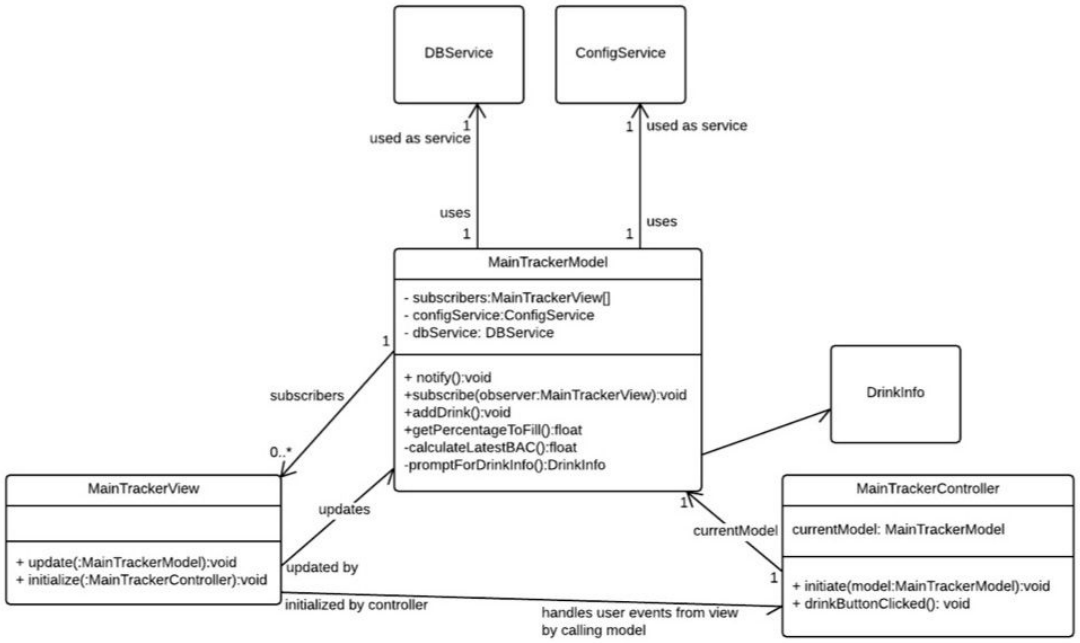
Our initial class diagram was to use three instances of the MVC pattern, one for each of the three tabs in the program: Home, History, and Settings.

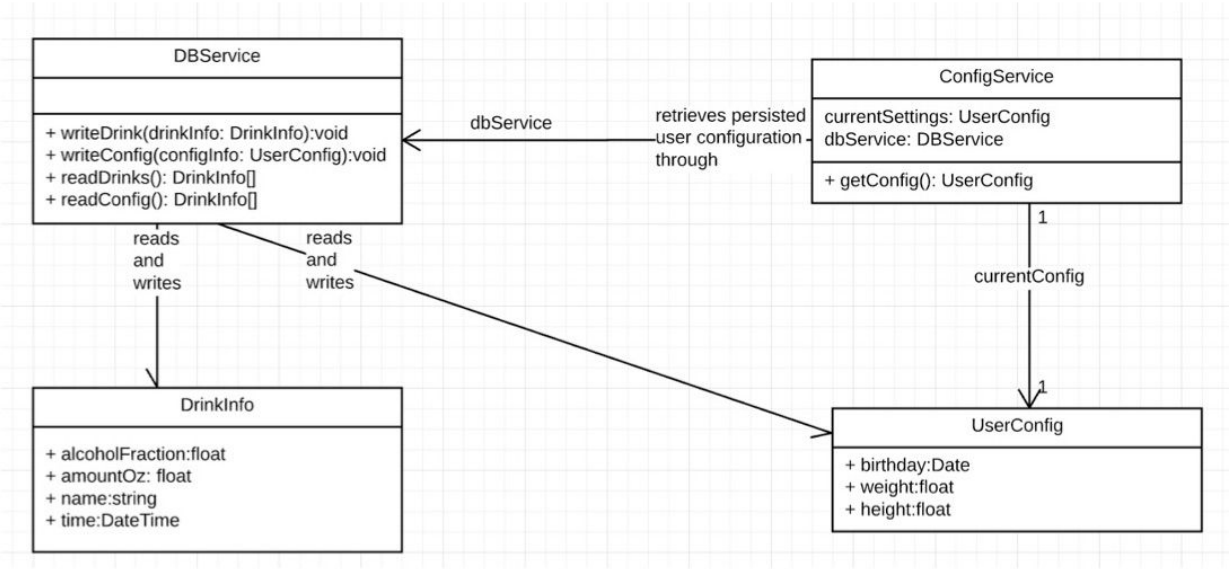
Old Class Diagrams:




DBService
1
used as service
1
uses
HistoryModel
- subscribers:HistoryView[]
- dbService:DBService
+ notify():void
+ subscribe(observer:HistoryView):void
+ getLast30DrinksPerDay(): int[]
1
subscribers
0..*
updates
HistoryView
+ update(:HistoryModel):void
+ initialize(:HistoryController):void
updated by
1
currentModel
HistoryController
currentModel:HistoryModel
+ initiate(model:HistoryModel):void
1
initialized by controller
handles user events from view
by calling model

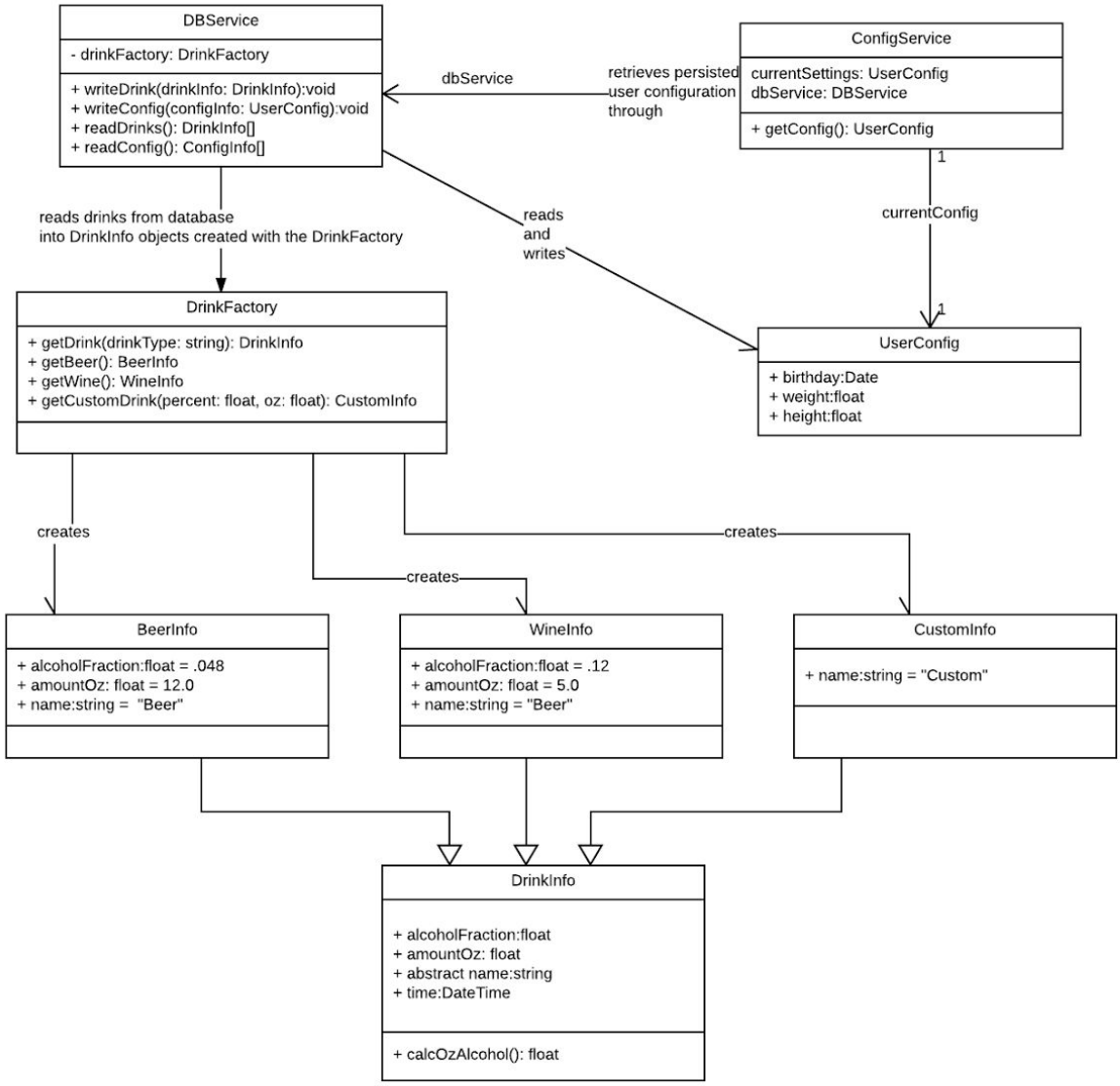

DBService
ConfigService
1
used as service
1
used as service
uses
1
1
uses
SettingsModel
- subscribers:SettingsView[]
- dbService:DBService
-configService:ConfigService
+ notify():void
+ subscribe(observer:SettingsView):void
+ changeSettings(newSettings): void
1
subscribers
0..*
updates
SettingsView
+ update(:SettingsModel):void
+ initialize(:MainTrackerController):void
updated by
1
currentModel
SettingsController
currentModel:SettingsModel
+ initiate(model:SettingsModel):void
+ newSettingsEntered(settings): void
1
initialized by controller
handles user events from view
by calling model

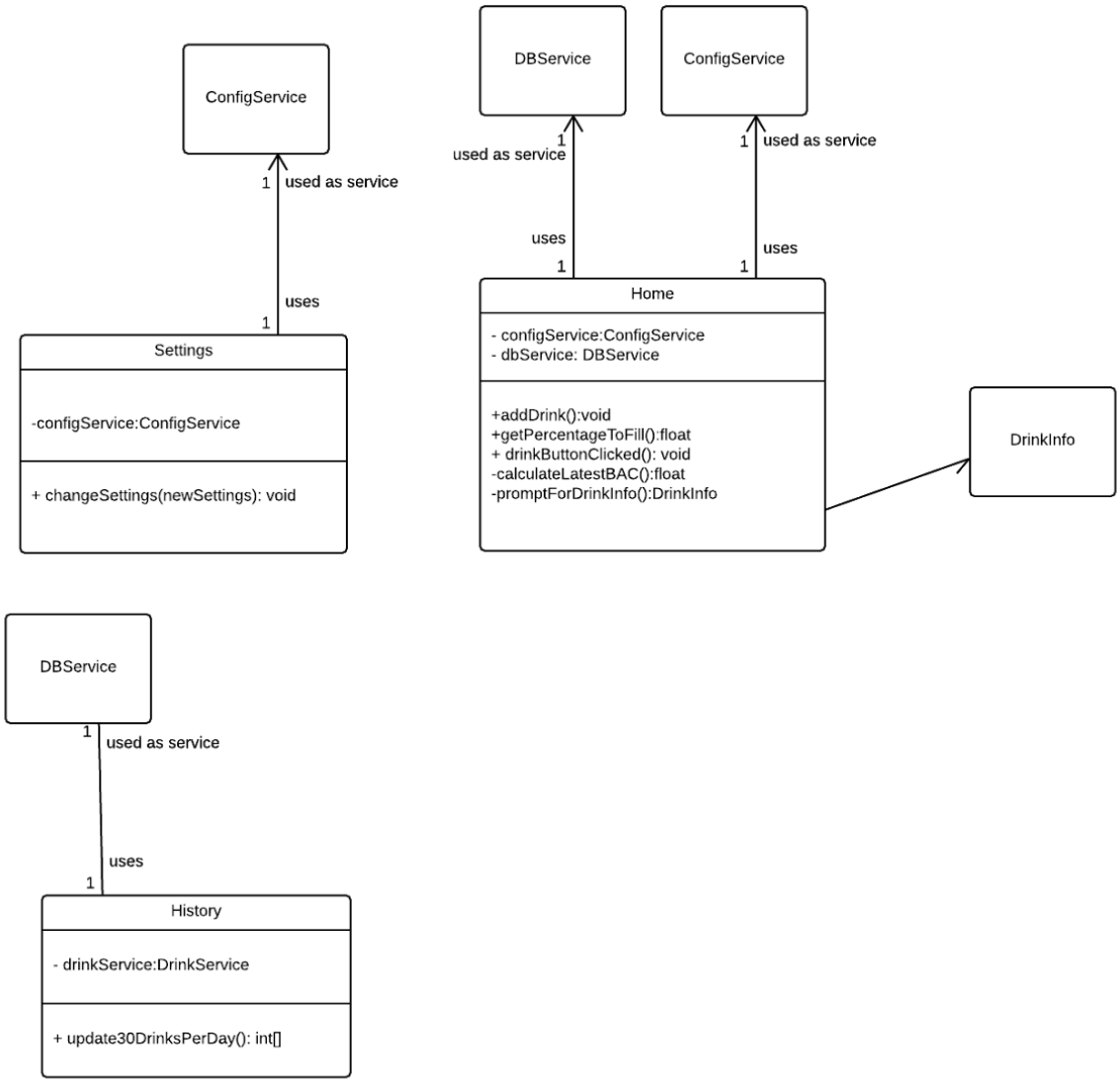

DBService
ConfigService
1
used as service
1
used as service
uses
1
1
uses
MainTrackerModel
- subscribers:MainTrackerView[]
- configService:ConfigService
- dbService: DBService
+ notify():void
+subscribe(observer:MainTrackerView):void
+addDrink():void
+getPercentageToFill():float
-calculateLatestBAC():float
-promptForDrinkInfo():DrinkInfo
1
subscribers
0..*
DrinkInfo
MainTrackerView
+ update(:MainTrackerModel):void
+ initialize(:MainTrackerController):void
updates
updated by
initialized by controller
1
currentModel
1
MainTrackerController
currentModel: MainTrackerModel
+ initiate(model:MainTrackerModel):void
+ drinkButtonClicked(): void
handles user events from view
by calling model


DBService
+ writeDrink(drinkInfo: DrinkInfo):void
+ writeConfig(configInfo: UserConfig):void
+ readDrinks(): DrinkInfo[]
+ readConfig(): DrinkInfo[]
dbService
retrieves persisted
user configuration
through
ConfigService
currentSettings: UserConfig
dbService: DBService
+ getConfig(): UserConfig
1
currentConfig
1
reads
and
writes
reads
and
writes
DrinkInfo
+ alcoholFraction:float
+ amountOz: float
+ name:string
+ time:DateTime
UserConfig
+ birthday:Date
+ weight:float
+ height:float

New Class Diagrams:

The only major changes to our class diagrams were removing the MVC architecture, and adding a DrinkFactory:




ConfigService
1
used as service
uses
1
Settings
-configService:ConfigService
+ changeSettings(newSettings): void
DBService
ConfigService
1
used as service
1
used as service
uses
1
uses
1
Home
- configService:ConfigService
- dbService: DBService
+addDrink():void
+getPercentageToFill():float
+ drinkButtonClicked(): void
-calculateLatestBAC():float
-promptForDrinkInfo():DrinkInfo
DrinkInfo
DBService
1
used as service
uses
1
History
- drinkService:DrinkService
+ update30DrinksPerDay(): int[]

4. Did you make use of any design patterns in the implementation of your final prototype? If so, how? Show the classes from your class diagram that implement each design pattern (each design pattern as a separate image in the .PDF). If not, where could you make use of design patterns in your system? Show a class diagram of how you could implement each design pattern and compare how it would change from your current class diagram.

We used many design patterns in our system. These included Singletons, Services, and Factories.

Many of our classes are singletons. These include the Tab controllers “Home”, “History”, and “Settings”, and the Modal window controller “AddDrinkModal”. The services “DatabaseService”, “ConfigService”, and “DrinkService”, are also singletons, which are created during the bootstrapping process, and exposed through dependency injection to any class when needed during construction.

The services followed a simplistic service pattern where each service provides specific functionality to other classes, stressing separation of concerns by using the service to handle something. These services are listed in an inventory, and requested by other classes when needed. The services allow for each controller to share data, such as the user configuration, and read and write data from the database.

We also implemented the Factory pattern to create Drink objects. Drinks are all created through the DrinkFactory class which handles constructing the proper subclass of drink. This allows us to create Drink objects without exposing instantiation logic, and refer to the newly created object through a parent type. Also, it is good practice for extensible code to use creational patterns in a situation where further modification of subclass instantiation logic may be needed.

Our data-binding framework Angular encouraged us off of our original plan to use a pure MVC pattern for each tab in the application. Instead we used an HTML file as a view, attached to a class called a “Component” which handles modifying the data the view uses, and user interactions. We found this to be very effective, and although we could have added an additional layer to separate the “controller” and “model” functionality out of the component into their own separate classes, we found resources recommended using Angular in this way, and were happy in our decision to modify our original plan.

5. What have you learned about the process of analysis and design now that you have stepped through the process to create, design and implement a system?

Throughout the development of the BoozeTracker app our team learned many important lessons about analysis and design that are sure to be useful in future projects and our careers.

By far the most important lesson we learned is just how crucial the planning and design stages are for successfully completing a group project. Prior to this class, the members of our team had approached similar coding projects head on with little thought to the overall design and how the pieces of the application would fit together. For this project we initially thought the extensive planning we did in project parts 2 & 3 would be overkill. However once it came time to actually create the application, having done all of the planning beforehand made writing the code much easier than it would have been. This experience gave us a new appreciation for the analysis and design steps of a project.

Another lesson we learned while working on this project is the benefit of design patterns and how to apply them to a particular project. As a team, we decided we should be using design patterns that helped us better communicate our ideas to each other. It allowed us to not just point to code to try to explain an idea, but explain conceptually what we wanted to accomplish, so that everyone has an easier time understanding.

The last major lesson we learned was the importance of flexibility. Plans are vital, but being willing to modify your plan when a better alternative exists is important to a project's efficiency. On one particular occasion, we ran into reasons to deviate from our original architecture. By deviating and applying a new strategy, we both saved time and wrote cleaner, less complex code.