University of Waterloo

ECE 452

SE2: Software Design and Architecture

Dress Me

Group No. 4

Tobi Burnett	taburnet
Thulasika Thiyaageswaran	tthiyaag
Matthew Tang	ml3tang
Matthew Lee	kh37lee
Marija Ivanovska	mivanovs
Jeffrey Yuen	jwy2yuen

https://github.com/Matt1504/ECE-452-Project.git

August 4th, 2021

Architecture

The main architecture of our app consists of data stored in two JSON files that store data for clothing items and outfits. These objects have specific properties and the activities/classes manipulate (insert/modify/delete) the data according to the user's inputs and display the data for the user to see. This architecture supports the functional and non-functional attributes described in the proposal. Below are the functional properties and how they were implemented by the architecture:

- **Able to add clothing articles to virtual wardrobe**
  - User fills in the properties of the clothing item (category, tags, etc.) in an activity and the inputs are made into a JSON object and put into the storage as a JSON file labelled "clothing-data.json".
- **Able to add a description to clothing such as "black champion hoodie"**
  - This is one of the attributes of a clothing item object from the class object AddClothingItem, which is an inherited class from ClothingItem.
- **Able to add tags to clothing such as "t-shirt" or "sweater" or "workout clothes"**
  - This is also one of the properties of a clothing item object, with the value as an enum value from the Tags class.
- **Able to select tags they would like to search by to generate a recommended outfit**
  - Done by searching through all the clothing items in the JSON file and retrieving the items with the selected tags through simple parsing and filtering.
- **Able to bookmark preferred clothing (favourite feature)**
  - This is also one of the properties of a clothing item object stored as a boolean value.
- **Able to combine clothes to create and save outfits under My Outfits**
  - The user chooses which clothing items they want to add and the unique identifiers of the item will be added as a property of an outfit item which is stored as a JSON object in the storage as a JSON file labelled "outfit-data.json".
- **Able to keep track of how often each piece of clothing is worn**
  - This is also one of the properties of a clothing item object as an integer value that we increment every time the clothing is chosen to be worn.

Below are the non-functional properties and how they are supported by our architecture:

Usability

The app supports usability as it is intuitive and simple to navigate through. All clickable buttons stand out as they are a different colour from the background or outlined with a border. They also have the text and sometimes an image that tells the user exactly what they expect if they click the button. For example, the button that says "Add Outfit" brings you to the Add Outfit page, the button that says "Add Clothing" brings you to the Add Clothing page, etc. The app is also user-friendly to navigate as all of the main features offered are accessible via the homepage and the sidebar menu that is accessible from anywhere in the app. Inside of

these features and in the header, common icons are used such as a cross to exit, pencil for editing, gear for settings, bullets for tags, dropdown menus, lists, etc. so that the user immediately understands their actions.

Evolvability

The app is designed in a way that is evolvable as our architecture style and methods are easily adaptable and modifiable in the event of future changes. Implementing new features and improving existing features is simple as it only involves creating new methods, using or editing existing methods (such as methods to read and write to the database), making changes to the data schema, or a combination of the above. Also, as the app grows, the data storage may be upgraded to a more complex database system. This transition is easy as the methods to read and write from the database will have to be modified slightly, but the methods that work with the data can remain unchanged.

Availability

All features of the app, except for retrieving weather are available for the user at all times. Because all of the components of the app deal with the manipulation of data and all the data is stored locally on the phone, the app can always be used and does not depend on WiFi access or database/server availability. The weather feature of the app only affects the recommendation algorithm and even when the weather is not available (due to internet or location issues), the recommendation algorithm can still be used, just without the filters introduced from the weather data.

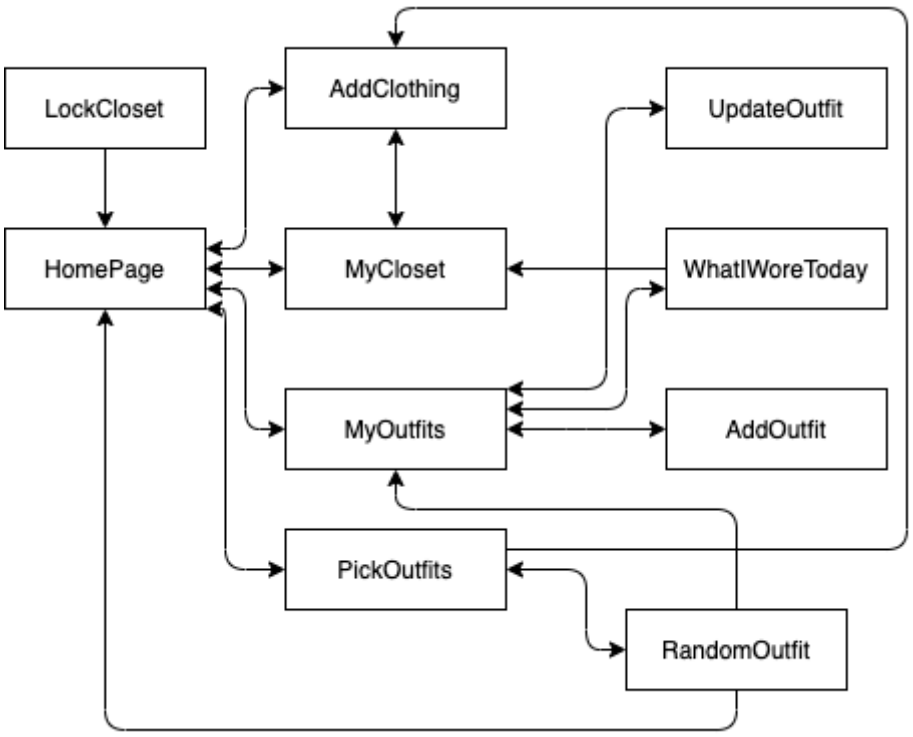

Figure 1.1: Component Diagram of DressMe Application

Architectural Style 1 - Data-Centered

The data-centered/repository style is used within our application. The application implements the data-centered architectural style because a central body of information is kept to help track the user's closet long-term. The application needs to store the clothing data in a central location so all functionalities can access this data and modify it.

When the user adds their first clothing, a JSON file is created called "clothing-data.json" which contains the data about the clothing articles that will be shared across the application to multiple pages (components). It is also similar when the user adds their first outfit, a JSON file is created called "outfits-data.json", which contains the data about the outfits that will also be shared across multiple pages. These two files act as the central data location which holds the current state of the application. The connectors would be the other class pages such as Add Clothing, Create Outfit, Closet Clean-up, Pick my Outfit, etc. These will all access this one central internal data storage. Any modifications that are made through one connector such as deleting an article of clothing or updating the tags of clothing will be made in these two files. When a different connector accesses the data/memory again, it will receive the new values. This style makes it easier for our application to keep track of any changes and update the display accordingly across all pages. Furthermore, this architectural style makes it easy to extend the application's features as any new feature would just be another component connected to the shared data.

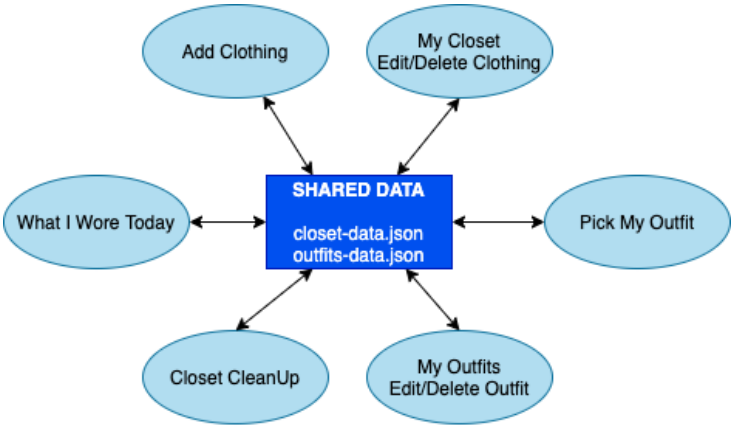

Figure 1.2: Data-Centered Architecture of the DressMe Application

Architectural Style 2 - Pipe and Filter

Another architecture style used was the pipe and filter. All functionalities included in the applications namely, Adding and Modifying Clothing, Displaying Clothing, Creating and Updating Outfits, Displaying Outfits, Picking an Outfit, Getting the Weather, and Choosing the outfit worn for the day, would all represent components or filters in the architecture style. Each of these filters independently processes the given information and provides useful output to be displayed to the user or saved to the device storage. Firstly, when using the "Add Clothing" filter, the user uploads an image from the device either via the photo gallery or through a new photo taken from the phone's camera and adds data input for the description,

category, secondary category, and tags for the outfit. This information is used to create an AddClothingItem object which is further saved to a JSON file. The same mechanism is used for "Update Clothing" or "Delete Clothing" filters; new ClothingItem objects are created, modified, and removed accordingly. When the "Displaying My Closet" filter is used, the information is read from the device storage and ClothingItem objects are created. These ClothingItem objects are then separated into categories through unique ArrayLists, which can be further filtered through tags to be displayed to the user. Likewise, user input is used to create OutfitItem objects when utilizing the "Add Outfit" or "Update Outfit" filters. These newly created objects are manipulated within other filters throughout the applications, such as PickMyOutfit which utilizes the pre-created ClothingItem objects to create a new randomly-defined OutfitItem object. Additionally, the "What I Wore Today" filter accepts an OutfitItem Object from the user to update values in the JSONArray containing all OutfitItem Objects in the device storage. Furthermore, for the "GetWeather" filter, an external API to OpenWeatherMap is called and values for the temperature, humidity, and location are displayed.

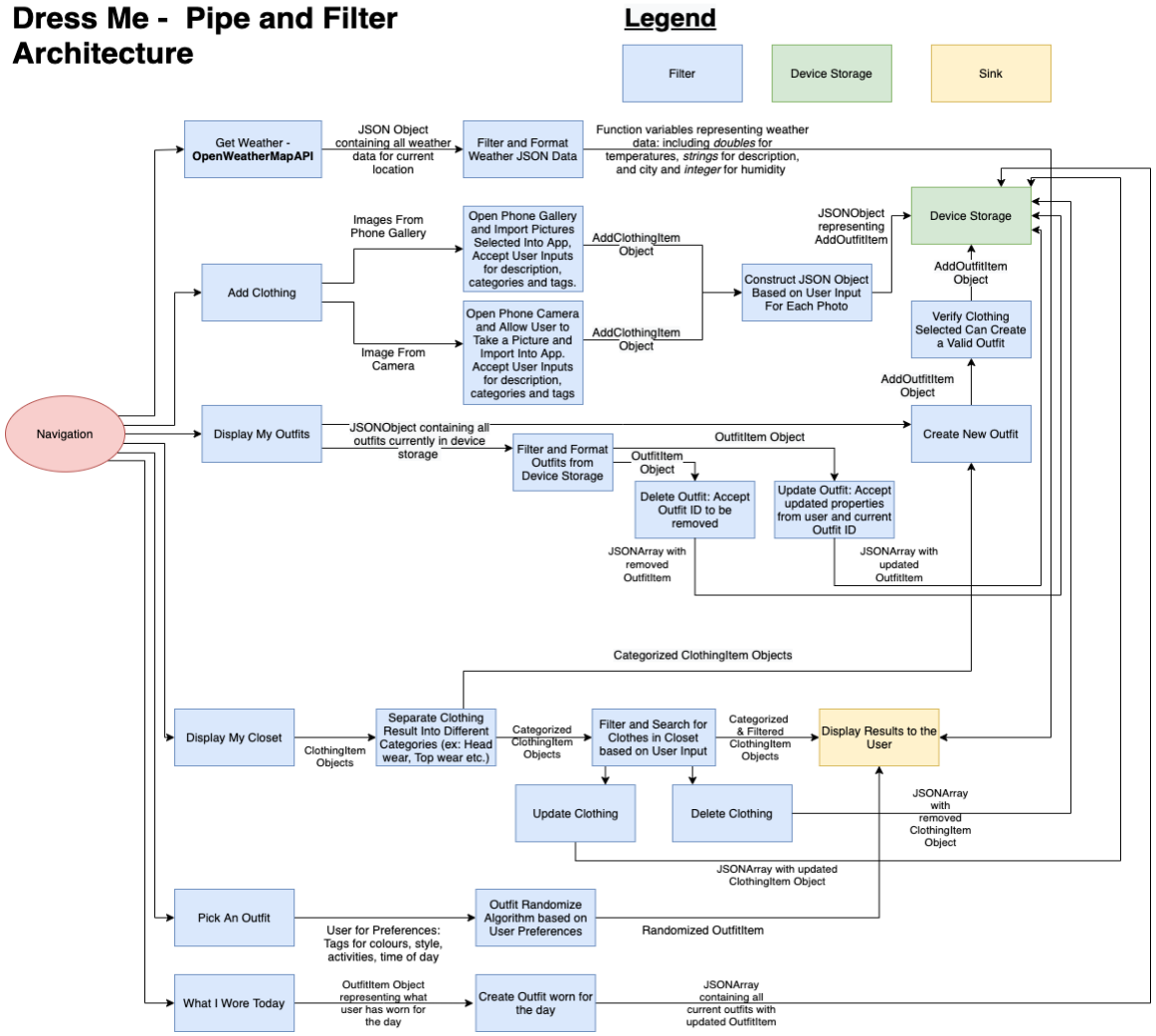

Figure 1.3: Pipe And Filter Architecture of the DressMe Application

Phone-Based/External Services

- **android.permission.INTERNET**
    - Used with the external API OpenWeatherMap to retrieve the current temperature details for the HomePage and PickMyOutfit components.
- **android.permission.READ_EXTERNAL_STORAGE**
    - Used to retrieve photos from the device photo library for the AddClothing component.
- **android.permission.WRITE_EXTERNAL_STORAGE**
    - Used to take photos within the application and store it in the phone photo library for the AddClothing Component.
- **android.permission.WRITE_INTERNAL_STORAGE**
    - Used to write to JSON files within the application: “clothing-data.json” and “outfit-data.json”. These files were created and modified within the AddClothing, AddOutfits, and UpdateOutfit components.
- **android.permission.READ_INTERNAL_STORAGE**
    - Used to read from JSON within the application: “clothing-data.json” and “outfit-data.json”. These files were accessed within the MyClothing, MyOutfits, and UpdateOutfit components.
- **android.permission.CAMERA**
    - Used to take photos within the application for the AddClothing component.
- **android.permission.ACCESS_FINE_LOCATION**
    - Used with external API OpenWeatherMap, to retrieve the correct weather details for the user’s current location for HomePage and PickMyOutfit components.
- **com.google.android.gms:play-services-location:17.0.0**
    - Allows the retrieval of user location. Used by OpenWeatherMap API for HomePage and PickMyOutfit components.

Design

Classes and UI Interfaces (XML)

Enum Classes:

Category, SecondaryCategory, Tag

Design Rationale: These three Enum classes are used to describe 3 important properties of the defined ClothingItem object and all of the options for each property. The Category enum broadly describes the type of clothing the item belongs to and the SecondaryCategory describes more specific categories within each Category enum value. This hierarchy is implemented by having methods in the class that returns all of the SecondaryCategory values for each Category value. The Tag category contains values that describe additional characteristics of the clothing item that is being added. Each value in all of the enums also has a "name" property which is the name of the value as a string. This is used for loading the value into our JSON data file. The AddClothing, MyCloset, MyOutfits, PickOutfits, and RandomOutfit classes use these enums to get the values to show to the user as options. Having the values as enums makes it easier to add or remove values as necessary in the future. Furthermore, each enum class has static methods that are called throughout the application to retrieve either the entire set or a subset of the enum class to help display appropriate data. For example, the Tag enum class has a method called `getSeasonal`, which returns an enum set of all the tags regarding the seasons.

Alternative: An alternative is to remove the use of enums and allow the programmer to place the values explicitly in the string.xml file for variable use or to hardcode the value programmatically. However, enums not only allow for us to store pre-set values for the users to input as their preferences when creating new items, but it also helps us to ensure that valid values for Categories, Secondary Categories, and Tags are being saved to the device storage. The programmer could easily make spelling errors when inserting values to the JSON files which would ultimately result in runtime errors as these values are used throughout the application. Additionally, as previously mentioned, allows for the easy addition and removal of values in the future. Having all of the values defined in a single location allows for consistency throughout the application.

"Item" Classes:

ClothingItem, OutfitItem, AddClothingItem, AddOutfitItem

Design Rationale: The first two classes are two objects that represent the two types of objects and their properties in the JSON data. These "Item" classes allow us to group properties associated with individual ClothingItems and OutfitItems for ease of display, deletion, creation, and modification. It also allows for consistency with each object created, as all properties are required to create the aforementioned Items. Furthermore, the AddClothingItem and AddOutfitItem classes are inherited from the ClothingItem and OutfitItem classes respectively. These subclasses contain added attributes that will help with

collecting and adding the data to the JSON files, but since they also use the existing attributes, making them subclasses is the most efficient option to reduce redundancy.

Alternative: Alternatively, instead of creating classes for each type of object in the application, another choice would have been to accept each property of the clothing or outfit items and pass each separately into different functions, much like the Data Clump antipattern. However, we chose to create classes such as OutfitItem and ClothingItem, to create objects which can easily be manipulated, passed into functions, and stored.

Adapter Classes:

AddOutfitGridAdapter, CategoriesAdapter, PopupTagsAdapter, TagsAdapter, SliderAdapter

Design Rationale: Adapter classes grant the ability to dynamically create views according to lists that are required for display, such as categories, CategoriesAdapter, PopupTagsAdapter, and TagsAdapter. This helps reduce code duplication in UI development and XML files. These adapters are attached to the RecyclerView layout which allows for an optimized user interface to enhance the usability of the application and this is also why most of these classes are subclasses of the RecyclerView Adapter class.

Alternative: An alternative design is to create UI elements (TextViews, CardViews, etc.) for each tag or category in the XML file. Not only would this create code duplication, as previously mentioned, but this would lengthen the code in a way that would make it more difficult to understand. Additionally, if tags or categories are removed from the enums previously mentioned, the elements in the UI would also have to be updated accordingly each time. Furthermore, using a single adapter for all situations was considered, but since each of the final adapters is used for different purposes and had different forms of functionality, it was best to separate the adapters into individual classes based on their use case.

Activities and Fragments:

Classes: MainActivity, MainFragment, AddClothing, AddOutfit, HomePage, MyCloset, MyOutfits, RandomOutfit, WhatIWoreToday, UpdateOutfit

Design Rationale: The design selected was chosen because we found that it was the most appropriate division of major functionalities within the application. The focus of the app is to allow users to add clothing, view the clothing, create outfits, view these outfits, and pick outfits worn or to be worn for the day. Therefore, the team chose to create a main class, MainFragment, to act as the base class that contains the code for the parts of the methods that are invariant. Then, the other classes will be subclasses of the base class to fill in the details of the implementation of the methods. Furthermore, these classes highlight and cover each of these functional specifications, and ensure that the functions within these classes are ones that only aid in achieving the specific specification. Each subclass highlighted above also holds independent functionalities that the user may access only within the said fragment. For example, the MyCloset fragment only holds functions used to manipulate ClothingItems in the device storage. There is also the MainActivity, which holds static methods that are helper

functions that can be used across all activities like reading and writing from a file and getting the weather details.

Alternative: An alternative design would be to place all modules into one class file which is ultimately accessed for each of the different functionalities in the application, much like the Blob antipattern. This would decrease the cohesion of modules, reduce organization, and make it more difficult for other developers on the team to quickly decipher the module.

Class Diagram

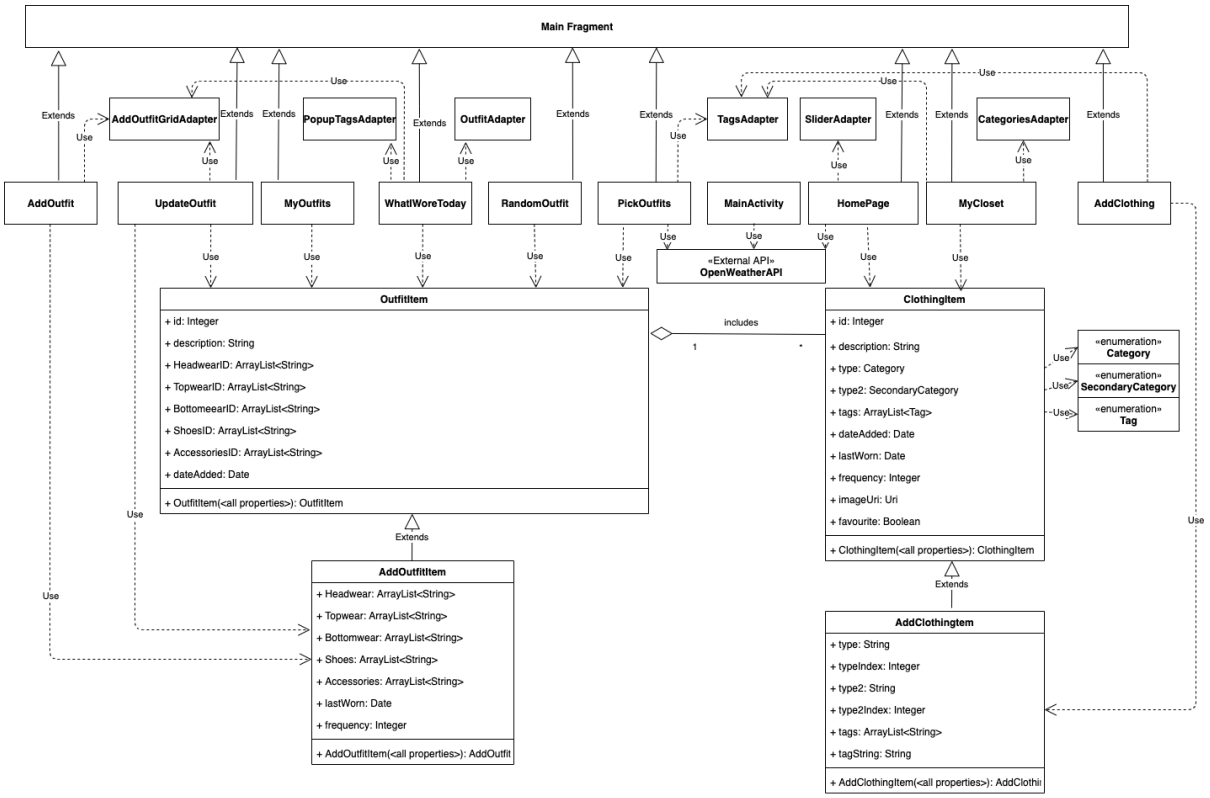

Figure 2.1: Class Diagram of DressMe application

Physical Location of Classes:

All classes reside in the `app/java/com.personalapp.dressme` directory on the Client, meaning all classes and code base is stored in the application itself.

XML Files: activity_main, activity_unlock_closet, add_clothing, add_outfit, app_bar_main, category_row, clothing_popup, content_main, grid_item, home_page, my_closet, my_outfits, nav_header_main, outfits_grid_item, outfits_popup, outfits_row, pick_outfits, random_outfit, random_outfit_add_popup, slider_item, tags_basic_chip, tags_row, update_outfit what_i_wore_today, nav_graph.xml

Physical Location of XML files:

All XML files reside in the `app/res/layout` directory, except for nav_graph.xml which resides in the `app/res/navigation` directory.

Design Pattern 1 - Adapter

The adapter design pattern has been used within the DressMe application to create a more compatible interface to work with our specific class objects. The adapter design pattern is used when we don't want to change an adapted class to cater to our specific requirements and object functionalities. In our application, several adapter classes were used to convert the recycler view interface to be compatible with our specific objects. One example of this is the Slider adapter that was created. This Recycler view was converted to accept a list of URIs and display them as images. "onCreateViewHolder", "onBindViewHolder", and "getItemCount" are all functions that were overridden within the interface so that it would be accepted by the client class that is using it. Having a single class would be difficult as some parts of the interface would not be compatible with the class, so following the adapter design allows each class to be modified to work with each specific use case.

Design Pattern 2 - Template

The template design pattern has also been used within the DressMe application. The template design pattern is used so that a skeleton of an algorithm can be defined and repeatedly used throughout the application with changes to the sub-classes depending on the functionality. This allows for similar implementations to share the same code with minor adjustments. This concept has been used in several places throughout the DressMe application as reusing common code has not only saved a lot of time but helped keep everything consistent within the app. One example of using the template design within the application is how the MyCloset page is loaded versus the MyOutfits page. They both maintain the same algorithm structure however certain subclasses have been changed to match the specific functionalities. Specifically, the populateCloset function differs for both subclasses. For the MyCloset class, the function will read from the respective JSON file and just add each clothing as a JSON object to the closet array list. However, for the MyOutfits class when the respective JSON file is read, only the clothing ID is included in the "outfits-data.json". Therefore, a hashmap needs to be created to map the clothing ID to its properties. In both classes, the file is parsed through and ClothingItem instances are created but different things are then done with those objects. Furthermore, there is also the general case where these classes are subclasses of the Fragment class. The Fragment class defines methods like onCreate, onCreateView, and onViewCreated, but the actual implementation details are left to the subclasses. These are some examples of how this pattern is used within the DressMe application.

Coupling and Future Changes

One of the main architectural styles chosen for this project is Data-Centered, because of this, each of the classes that exist within our design independently modifies the device storage and does not depend on each other for the desired functionality. This phenomenon is exemplified in several different instances, one of which is MyCloset and AddClothing. One may assume that the required sequence for displaying clothing items to the user is to access the objects created within the AddClothing class to display to the user, however, even if the overall clothing file is empty, the MyCloset Fragment still appears with a view indicating this to the

user and is completely independent of the AddClothing class. The same can be seen with the AddOutfit class and MyOutfits class. However, it is important to note that all functions that are unique to each ClothingItem or OutfitItem are kept within the module but still ensures that the Blob antipattern is not utilized as all functions within the classes are cohesive.

The current design readily facilitates future changes, as new functionalities are all created with new class files, which contain their independent operations. One way in which we envision the system being altered is to facilitate the integration of online stores, recommending new outfits to be purchased by users based on style and previously purchased items. Our design could support this change by creating a new class by the name of "UserStyle", which can store data related to all brand names and tags input by the user to be kept in the device storage. This data can be used to access external online stores and match brand names and tags to the site name and filter keywords for clothing items on the site respectively. This is a manageable addition due to the functionality's independent nature and the fact that our design is very modular.

Sequence Diagrams

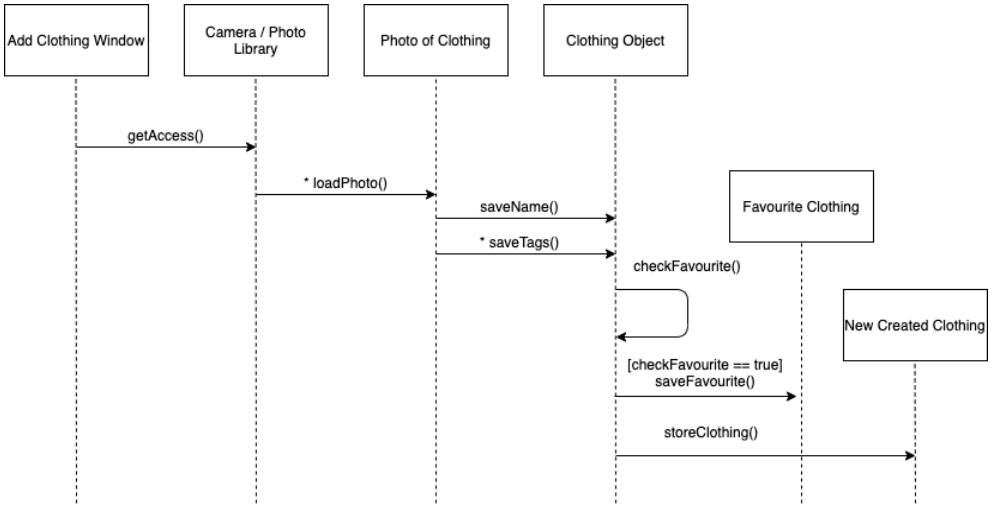

Figure 2.2: Sequence Diagram for Adding New Clothing

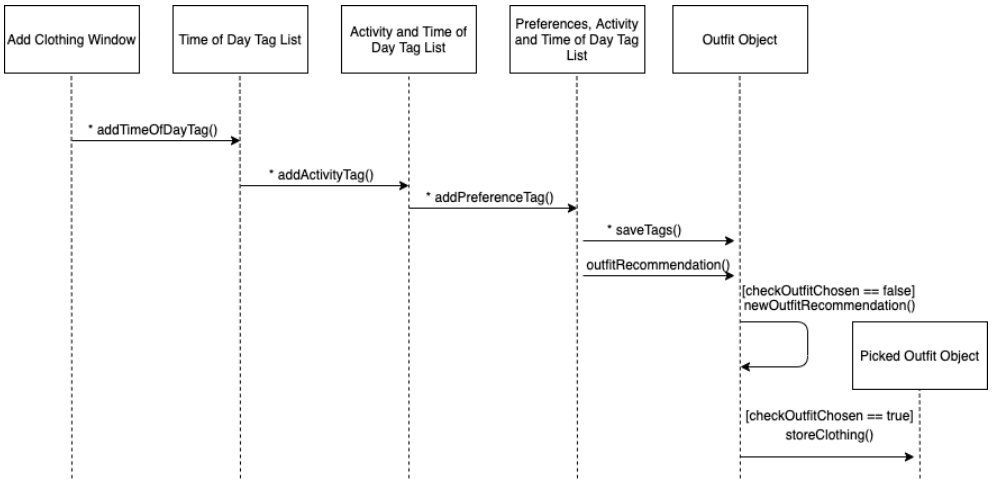

Figure 2.3: Sequence Diagram for Picking An Outfit

How we Mitigated the Harms

The buddy evaluation mentioned the general target population would be harmed as the app doesn't have restrictions on the type of image that is uploaded which could lead to compulsive shopping. The purpose of the app is to encourage users to utilize all the clothes in their closets. There is a frequency tracker within the application which makes recommendations based on clothes they haven't worn often. As a result, this helps reduce compulsive shopping habits and make better use of what they have.

Secondly, to reduce the harm towards pregnant people, a new pregnancy tag was added to the filtering algorithm. This allows users to add any pregnancy clothing and filter only for this clothing during their pregnancy months. In turn, this will prevent the recommendation algorithm to suggest any outfits that currently won't fit the user.

Furthermore, the buddy evaluation mentioned potential harm for users who identify themselves as certain gender identities. However, the application does not recommend clothes based on gender, rather it recommends clothes and generates outfits that the user owns in their closet. Assuming that they are comfortable with the clothing they upload, we can ensure our recommendations will not be harmful.

Next, to prevent harm against those who are temporarily disabled or sick, new tags were added. This will help the user pick outfits with which they can be comfortable when they have to stay home for a long period. If the user had to wear specific pieces of clothing during their time of recovery, the closet cleanup feature will allow the user to delete any clothes within a certain time that is no longer necessary.

Regarding the low-income user group that may be harmed by the app, the team decided not to collect information on a user's financial state as this could be an invasion of privacy. Currently, the app only suggests outfits using the clothes the users themselves had added. In the future, if we decide to partner with external clothing companies, the stores we display to the user will be similar to the clothes in the closet already with similar prices.

Additionally, the evaluation stated we would harm those who are image-conscious because they could wear the same clothes twice in one week and we don't consider color schemes that would result in "weird looks". Our team has implemented a "What I Wore Today" page that will send notifications and ensure the user inputs the clothes they have worn. This will ensure the frequency is correctly tracked and we will only suggest outfits that haven't been worn as much. We also have color tags and have taken into consideration the color wheel to ensure we will only pair colors of clothing that will look good together.

Finally, we received additional feedback to ensure we are more considerate of the security of our application. Since all the data for the closet is saved directly on the user's device storage, we did not have to worry about any external security attacks. However, we have added a pin to lock the app itself to prevent other users from looking into the closet.

Team Member Contributions

Matthew Lee
- AddClothing
- RandomOutfit
- WhatIWoreToday

Tobi Burnett
- MyOutfits
- UpdateOutfit
- MyCloset

Jeffrey Yuen
- Class Enums (Tag, Category, SecondaryCategory)
- AddOutfit
- MainActivity (Notifications)

Thulasika Thiyaageswaran
- HomePage
- MyCloset
- Adapters (AddOutfitGridAdapter, CategoriesAdapter, PopupTagsAdapter, TagsAdapter, SliderAdapter)

Matthew Tang
- MainActivity (OpenWeatherAPI)
- PickOutfits
- RandomOutfit

Marija Ivanovska
- Main Activity (Navigation menu)
- Lock/Unlock Closet
- AddOutfit