**Pocket Closet**

System Architecture and Design

Team  
**Pocket Devs**

Ibrahim Helmy - 100989032  
Marcio Paulo - 100953830  
Omar AlDib - 100986553  
Saif Al Gawas - 100839462

Submitted to: Dr. Olga Baysal & Lexi Brown  
COMP 3004 Object-Oriented Software Engineering  
School of Computer Science   
Carleton University

Table of Contents

[1. System Architecture 3](#_Toc498641938)

[1.1. Object Oriented Architecture 3](#_Toc498641939)

[1.2. Some other Architecture goes here 5](#_Toc498641940)

[1.3. Architecture Diagrams 6](#_Toc498641941)

[2. Design Patterns 7](#_Toc498641942)

[2.1 Facade Design 7](#_Toc498641943)

[2.2 Adapter Design 8](#_Toc498641944)

Table of Figures

[Figure 1 - High level UML Class Diagram (showing inheritance of clothing classes) 6](file:///C:\Users\Iby\Downloads\D3.docx#_Toc498641945)

# 1. System Architecture

## Object Oriented Architecture

The primary software structural design style we adopted was that of Object-Oriented Architecture. There are many characteristics that are prevalent in the Object-Oriented Architecture style that prove to be great assets for the creation of the pocket closet application, they would not only streamline operations we need to perform but also help us adequately satisfy all our functional requirements.

The first of these properties is Inheritance. This property can be best summarized as the ability to create a child class from a given parent class while inheriting its capabilities for expansion or refinement. This is key for the way we planned to implement our clothing items. Clothing items all are children of a larger clothing class. From this class we have many subclasses that form categories of clothing; headwear, tops, trousers, dresses, skirts, outerwear, footwear. With inheritance we can quickly create these subclasses of clothing without the need to rewrite all the code and just inherit the common properties we need. For example; skirts need a color and material in the same way that a shirt does, however, they may have a few more properties that play a role in our closet's filtering process. These additional properties are important in the filtering process because, for instance, skirts cannot be warn in cold weather whereas a sweater could. Inheritance here allowed us to create a distinction between these two clothing items and yet maintain the common properties they both share without the need for rewriting anything.

The second property is that of Polymorphism. With polymorphism we can override the characteristics and behaviors of a parent class and implementing new ones on top. This allows us to have objects that have a different internal properties work within a shared external interface. In the context of the closet, this would mean clothing items would have a shared external interface for manipulation despite being different classes. This is very useful in the cases where we needed to perform large operations on the whole closet. With polymorphism the operation could now be performed without the need to execute said operation on each individual category of clothing contained within the closet, but instead uniformly to all of them at once.

A third property that spawns out of the selection of an Object-Oriented Architecture is the ability for easy changes. We wanted to allow ourselves the ability to add new features or remove them as we progressed along the development of the app without the need to rewrite the structure of the code. With Object-Oriented Architecture, objects created within the code are self-sufficient. Because of this fact, we knew we'd have the flexibility to divide the workload amongst the team members and comfortably adjust individual features without worrying about the consequences the changes could impose on the already created content.

These properties helps us meet functional requirements 3/4/10/11, 6/15. Functional requirement 3/4/10/11 are all fulfilled because now the app can filter the closet based on weather loading in only the appropriate class types from the clothing collection available. Auto-generation of outfits will be done by matching different object types together appropriately which will satisfy functional requirement 6/15.

## Some other Architecture goes here

## Architecture Diagrams

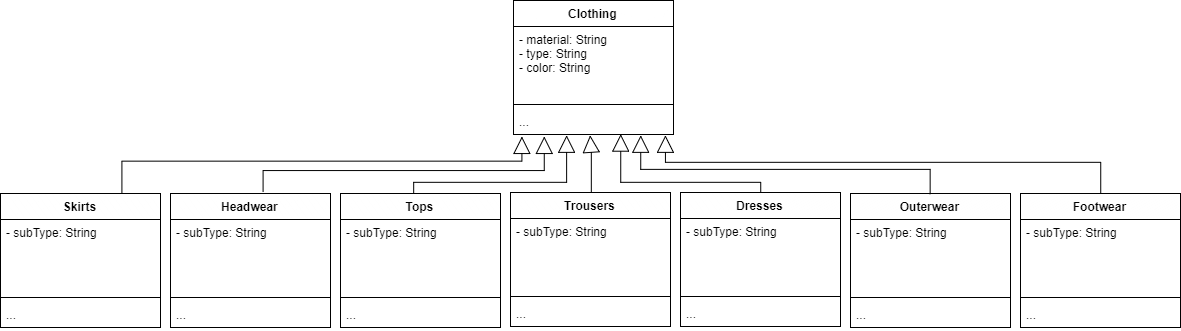


Figure 1 - High level UML Class Diagram (showing inheritance of clothing classes)

# 2. Design Patterns

## 2.1 Facade Design

The first design pattern that was implemented in the development of the pocket closet is facade. The facade design was applied to the filtering process in the app. The facade in this case is the filter button; a small button located at the top of the screen where the user selects the button and requests the closet to be filtered based on the weather. To the user this done immediately through the interaction with the graphical user interface of the app. Behind the scenes, however, the app must interact with the weather API and retrieve the needed weather information, then it is tasked with retrieving the appropriate clothing items from the database where the clothing is stored.

The facade here is the filter button and the subsystem classes are the weather API and the database. This shields the user from the complexity of the filtering process by allowing them to simply request it and see immediate results. By implementing this design pattern as well we've allowed ourselves to minimize the dependencies between subsystems and they can each be maintained individually. This stresses the goal of creating the application with a collection of self-sufficient subsystems as discussed previously in section 1.1. The user interface can also now be adjusted without needing to adjust the subsystems in the background and vice versa.

This pattern helped us fulfill our usability non-functional requirement. Because of the abstracted interface the user does not have to be exposed to the complexity of the filtering process. This makes the app significantly easier to use and browse, which was part of our product usability goals.

## 2.2 Adapter Design

The second design pattern we applied in development was the adapter pattern. We applied this pattern in the pack my bag feature. In the pack my bag section of the app the user adds clothes to a virtual bag based on the weather in an foreign travel destination. Similarly to how a user can add and view clothing items within their closet, the user is able to do the same within their travel bag. Since these two separate features have a lot of similar functionality, it would be very efficient to use the pre-existing closet code for the pack my bag. However, the problem was that the interface of the close would not work with the pack my bag and so we decided to apply the adapter design pattern to help solve that problem.

The target here is the UI of the pack my bag that will presented to the user at the time the feature is accessed. The adaptee in this case is the UI elements of the closet; add item interface, filter interface, remove interface, etc. The adapter is those same UI elements from the closet adapted to fit the travel bag context.

This helps us achieve all of functional requirement number 13, the pack my bag feature. With the adapter pattern we were able to create a brand new interface and functionality based on pre-existing ones while still maintaining our reoccurring goal of self-sufficient subsystems that for our application.