Final Report

Team: Tiffany Christensen, Chia-Lo Hsu, Joseph Marylander

Title: Closet Bot

Project Summary:
Closet Bot is an application that allows users to add and tag different articles of clothing into their "closet". Based on what is in a given user's closet and specified user conditions (for instance, "winter"), the app will generate a collection of outfits for you that you can then save if desired to an outfit closet.

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

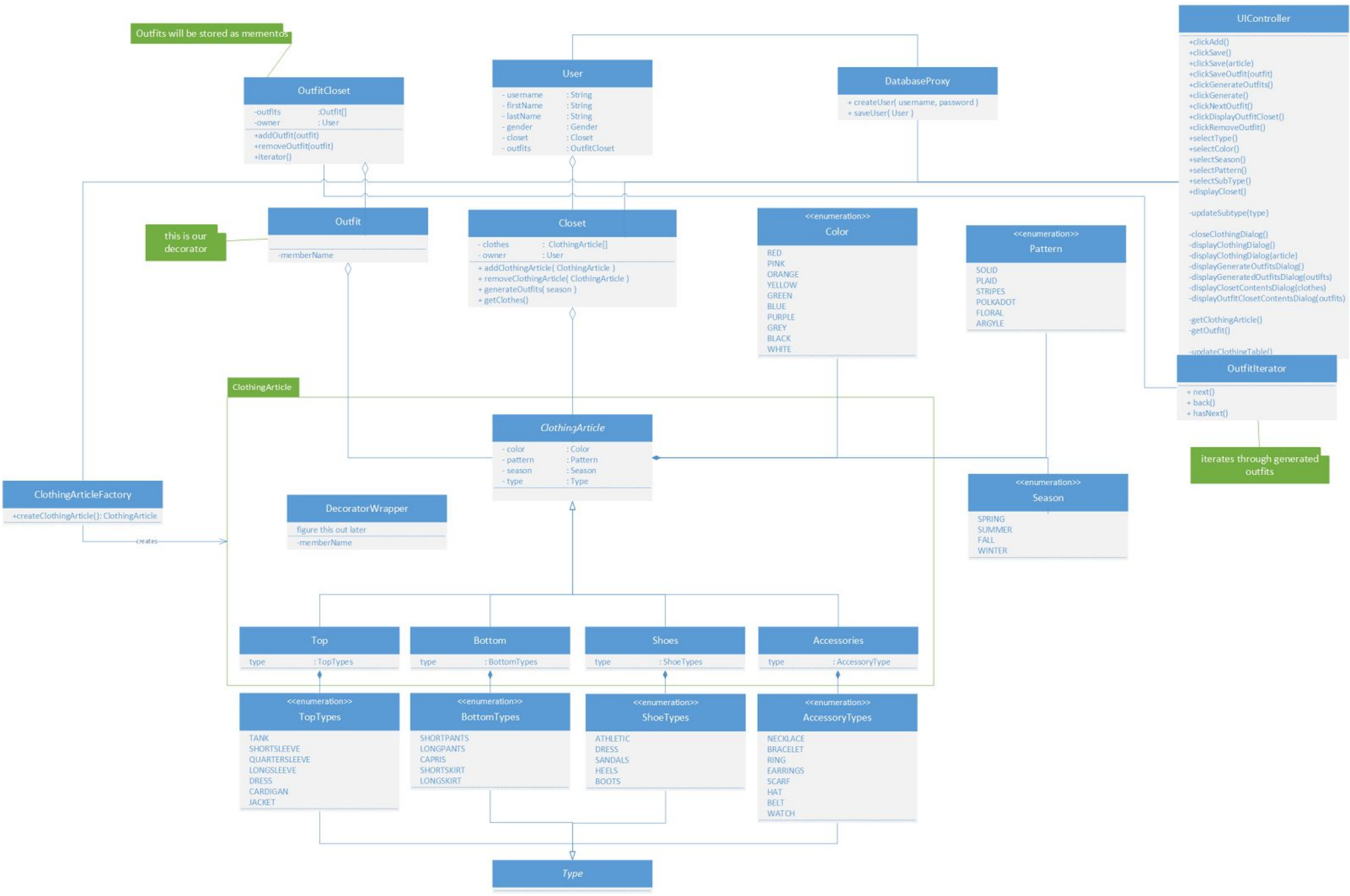
Implemented User Requirements	
ID	**Title**
UR_01	User can sign up
UR_02	User can log in
UR_04	User can add clothing to their "closet"
UR_05	User can edit existing clothing in their closet
UR_06	User can remove clothing from their "closet"
UR_07	User can view their own closet
UR_08	User can generate outfit based on season
UR_09	User can save generated outfit to outfit closet
UR_10	User can view saved outfit closet

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

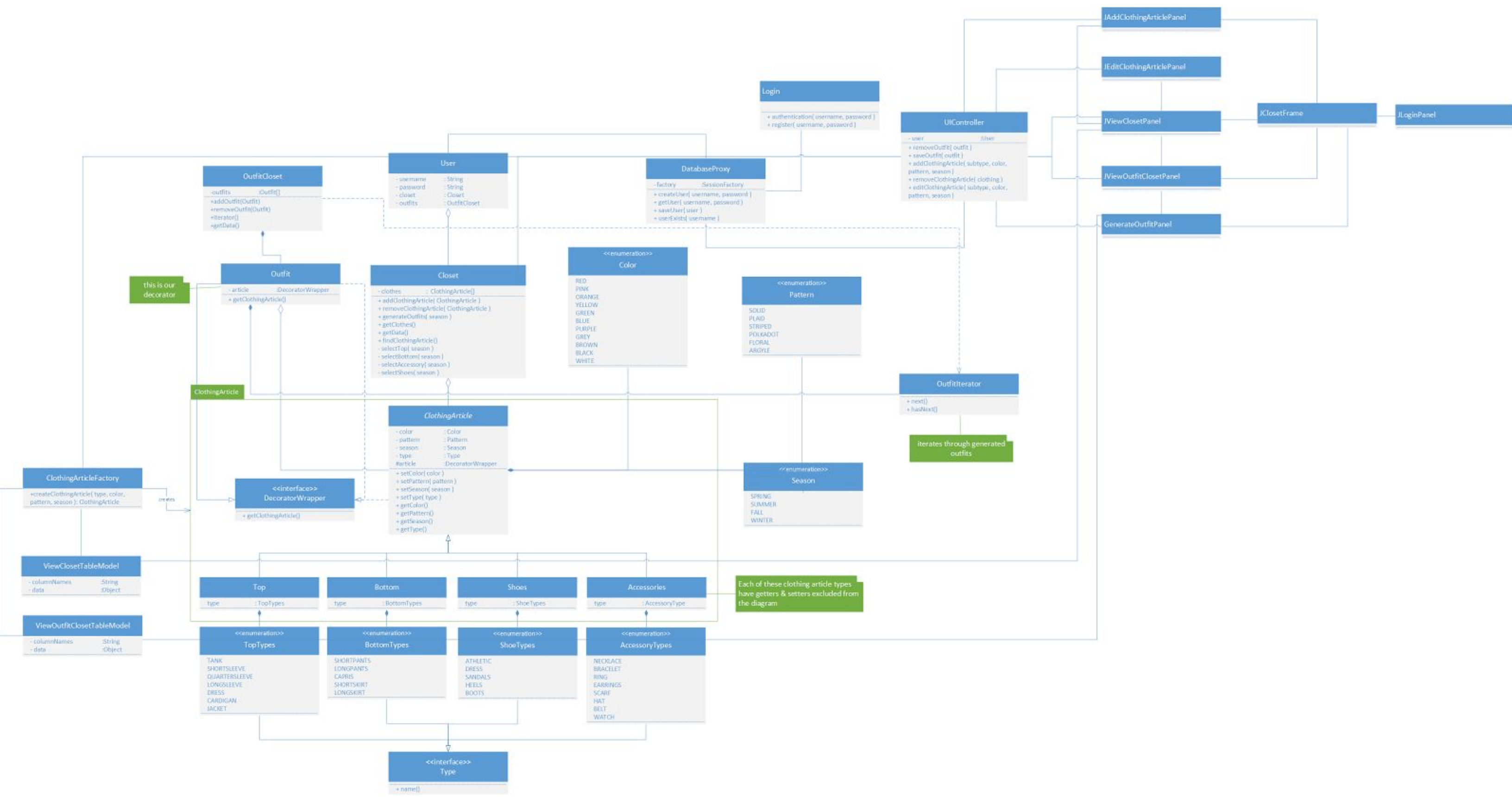
Not Implemented User Requirements	
ID	**Title**
UR_03	User can log out

3. **Show your Part II 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.**

Part II Class Diagram



Final Class Diagram



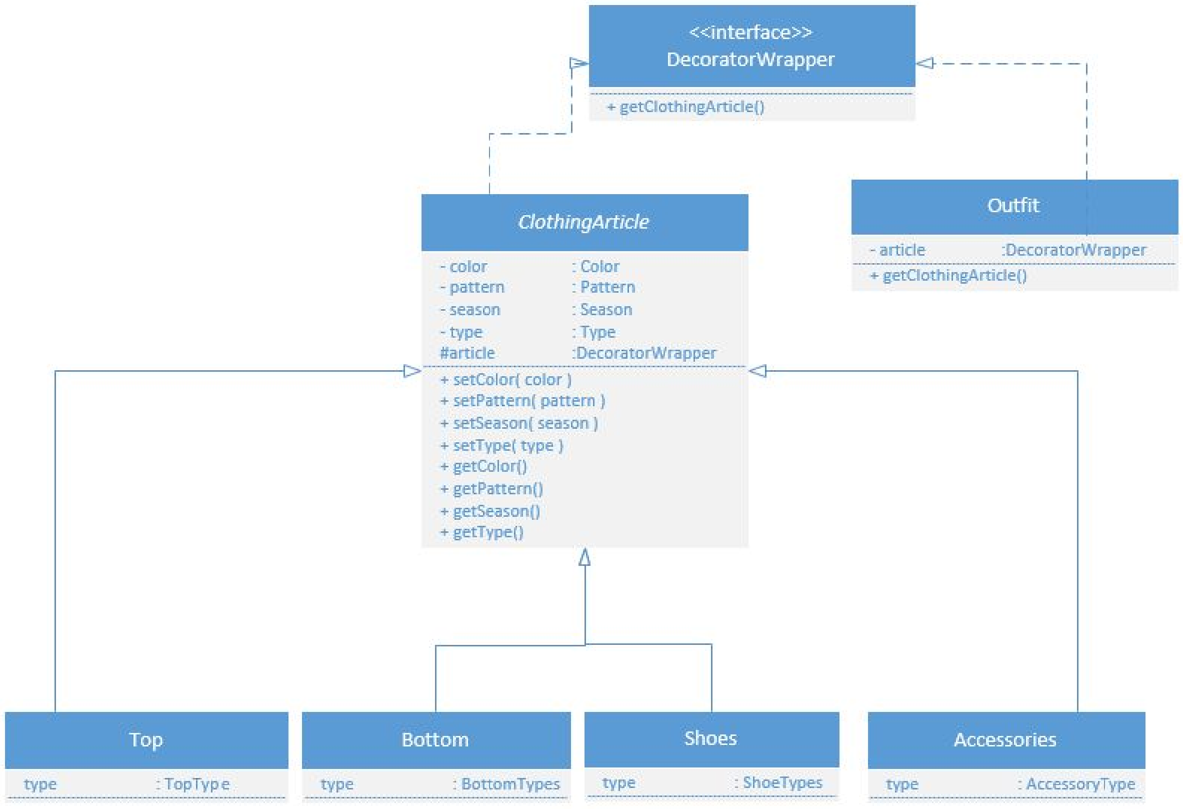
Discussion

Our class diagram did not change much apart from the addition of classes for our Java Swing frames, and more fleshed out classes for our design patterns once we figured out how to implement them efficiently. Designing the application up front allowed us to define the scope of the project early on, determine which classes could be implemented first, and which ones posed the greatest risk because of multiple dependencies. We were also able to determine earlier in the process where to best use design patterns before we ran into code smells. As a result, we were able to set up the structure of our application with dummy constructors very quickly and move forward with implementing actual functionality in a logical manner. Given the laid-out foundation, we were able to delegate tasks, estimate efforts, and track our progress throughout the project.

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.

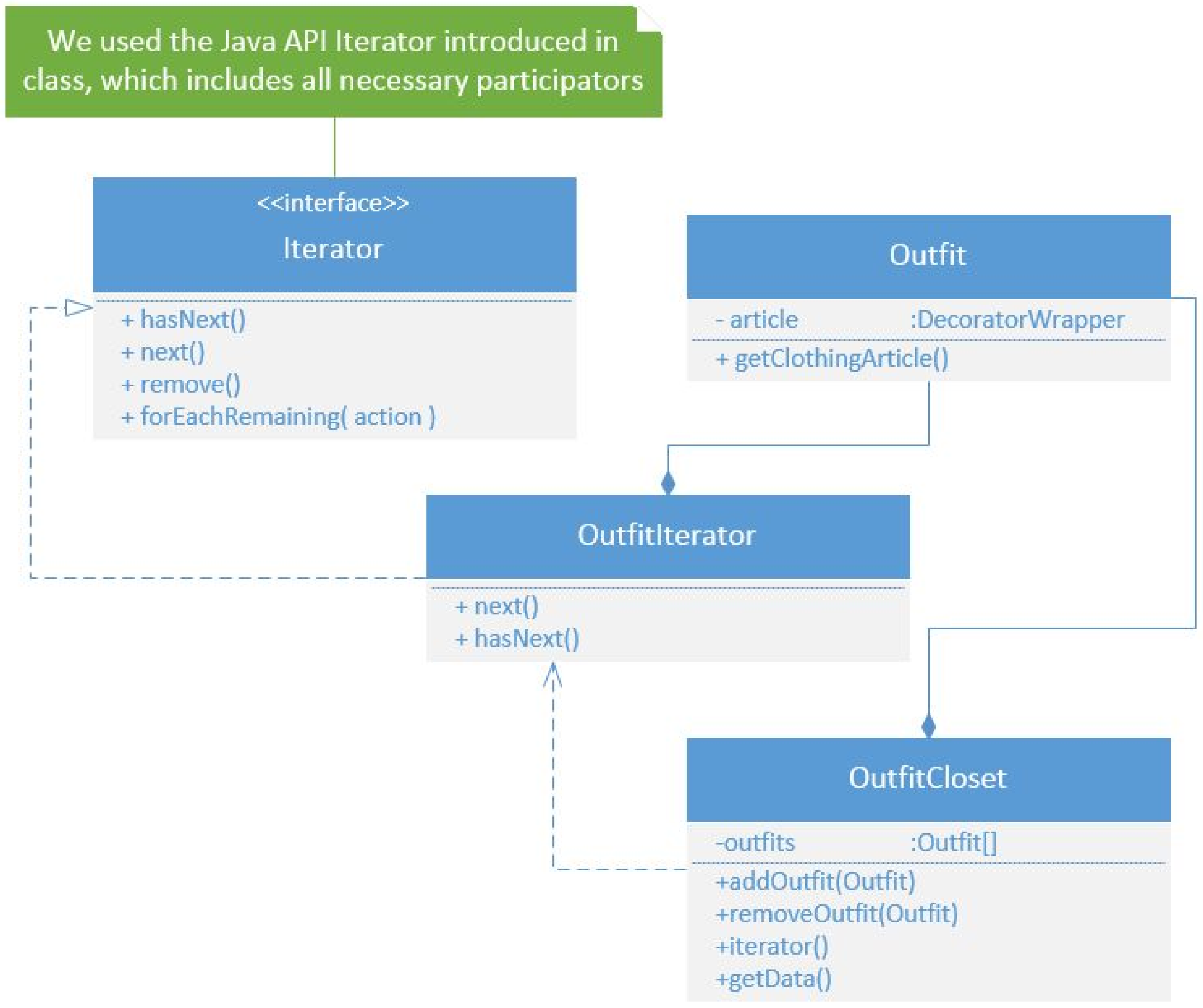
Decorator

In order to organize our generated outfits, we store them in a Decorator structure. Our "DecoratorWrapper" class is our Component, whereas "ClothingArticle" and its subclasses (Top, Bottom, Shoes, Accessories) are our concrete components that implement "DecoratorWrapper." Our "Outfit" class is our highest level wrapper for our outfits.



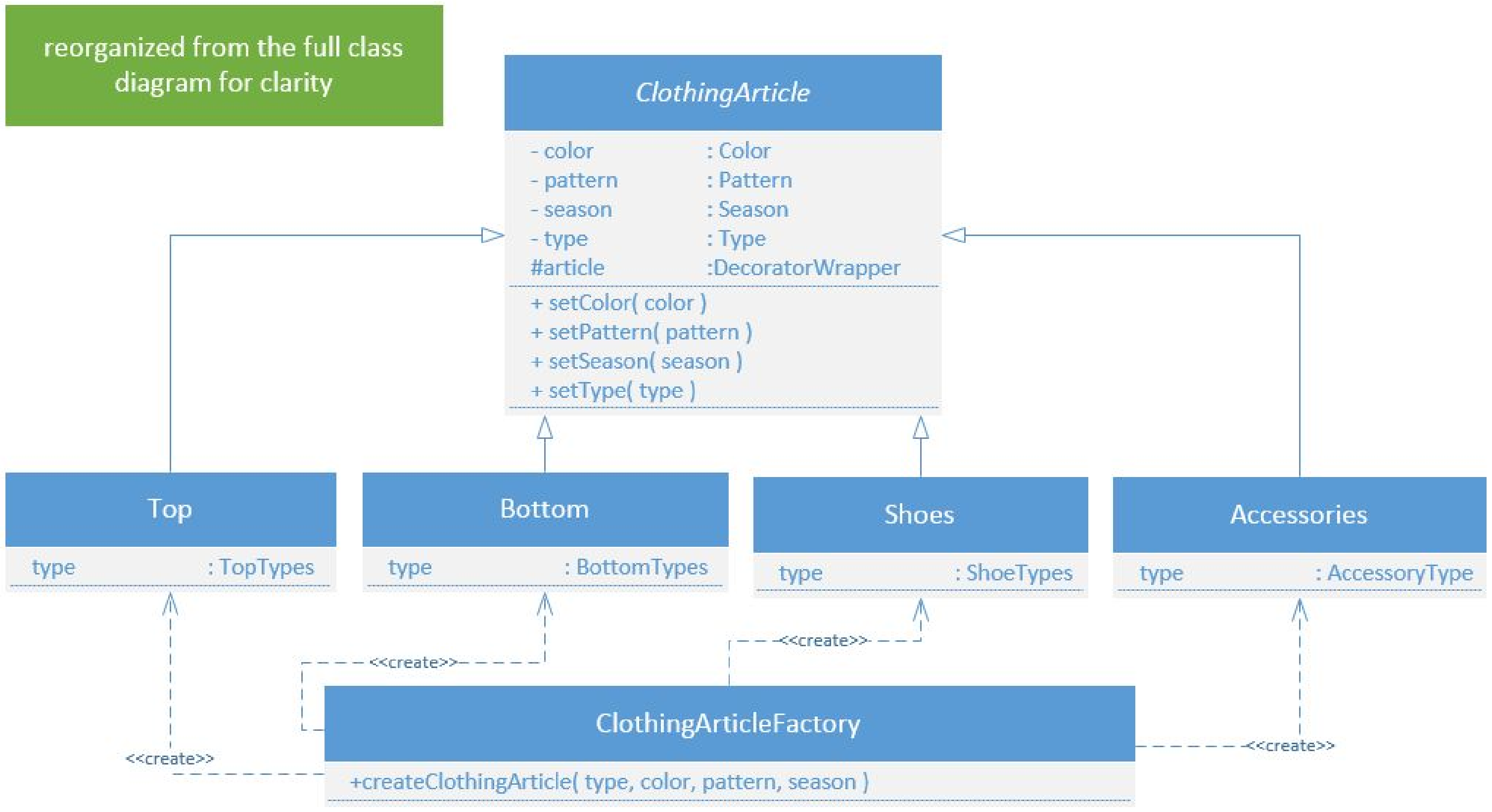
Iterator

Our iterator, “OutfitIterator”, is very simple because the given data structure is an array. We use this iterator whenever a user wants to look at their already generated “Outfit(s)”.



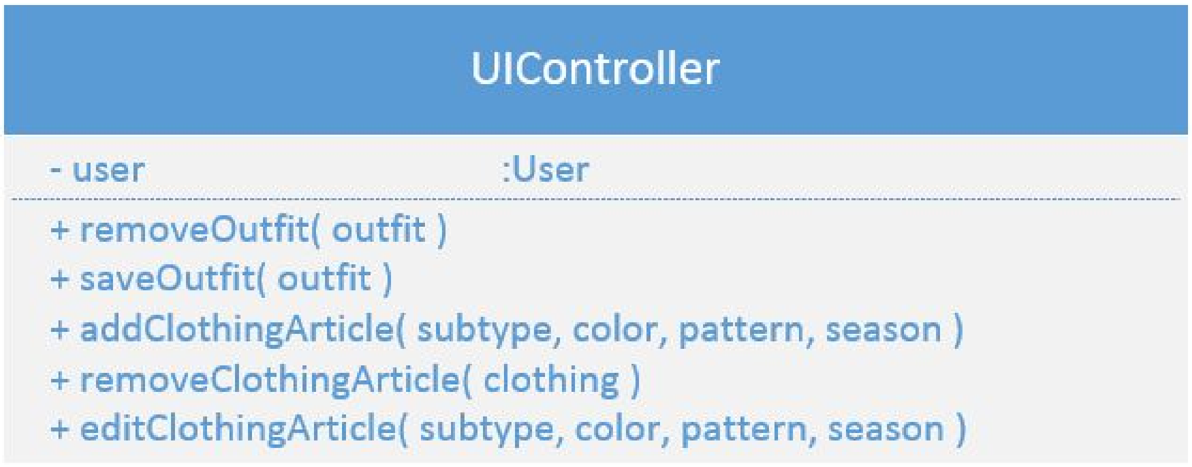
Factory

Our factory's purpose is to create new "ClothingArticles," hence its name "ClothingArticleFactory." "ClothingArticleFactory" will take in parameters: Type, Season, Color, Subtype; and return a new "ClothingArticle" of the correct subclass (Top, Bottom, Shoes, Accessories).



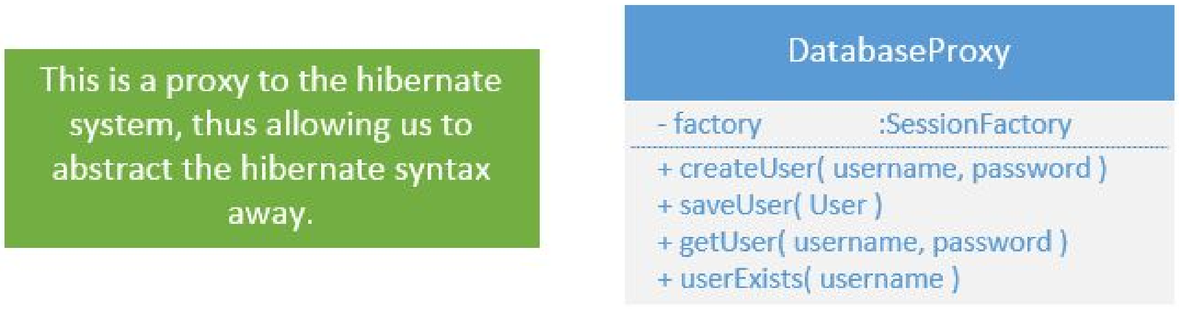
Singleton

The UIController is our singleton class, because we need to keep a single instance of it available to our View. Also, given that we only want a single instance of our User's data, we store it in our UIController to keep consistency.


UIController
- user :User
+ removeOutfit(outfit)
+ saveOutfit(outfit)
+ addClothingArticle(subtype, color, pattern, season)
+ removeClothingArticle(clothing)
+ editClothingArticle(subtype, color, pattern, season)

Proxy

Given that we want very specific operations done to our database, we needed to create a DatabaseProxy to interface with Hibernate to accomplish our user data saving/retrieval.


This is a proxy to the hibernate system, thus allowing us to abstract the hibernate syntax away.
DatabaseProxy
- factory :SessionFactory
+ createUser(username, password)
+ saveUser(User)
+ getUser(username, password)
+ userExists(username)

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?**

After stepping through the process of creating, designing, and implementing a system; we learned the value in planning, design patterns, and architectural patterns.

First off, through creating use cases, activity diagrams, sequence diagrams, and so on, we were able to define the scope of the project early on. We determined the best order to implement classes in light of dependencies through this planning, which resulted in less time taken to implement the actual code. We were able to set up the foundation of the project fairly quickly and then delegate tasks accordingly amongst the team. In doing so, we were able to efficiently estimate efforts and track our progress, while ensuring that all requirements were met. In a realistic environment, this is especially useful when delivering a product to a customer that directly traces their specified requirements. Planning for specific design patterns also mitigated the occurrence of code smells.

Prior to learning about design patterns, we tended to just hack together whatever data structure we thought would work in any given situation. This would then cause massive side effects in long-term maintainability, not to mention smelly code. However, now that we know how and in what situations we ought to use (or not use) a number of design patterns and anti-patterns, we know how to avoid commonly found problems. For instance, before we learned about the decorator pattern, we were going to store our outfits in a list. However, we learned that the decorator was a perfect fit for storing our outfits instead.

Lastly, by familiarizing ourselves with architectural patterns, we were forced to weigh the advantages and disadvantages of organizing our classes on the larger scale in different ways. Through the process of planning for and implementing an application with model-view separation (MVC), we were better able to determine how our different classes would relate. We were also able to independently develop both the UI and model, which resulted in fewer merge conflicts and more maintainable code in the long run. For example, we changed the implementation of a few classes in our model later on in the development process, but only needed to update the UI controller (rather than five more UI classes) because the controller handled the way data was fed to the view.