1.project

Team:

Si Shen

Yang Jiang

Albert Dayn

Title:

CampusRecycle

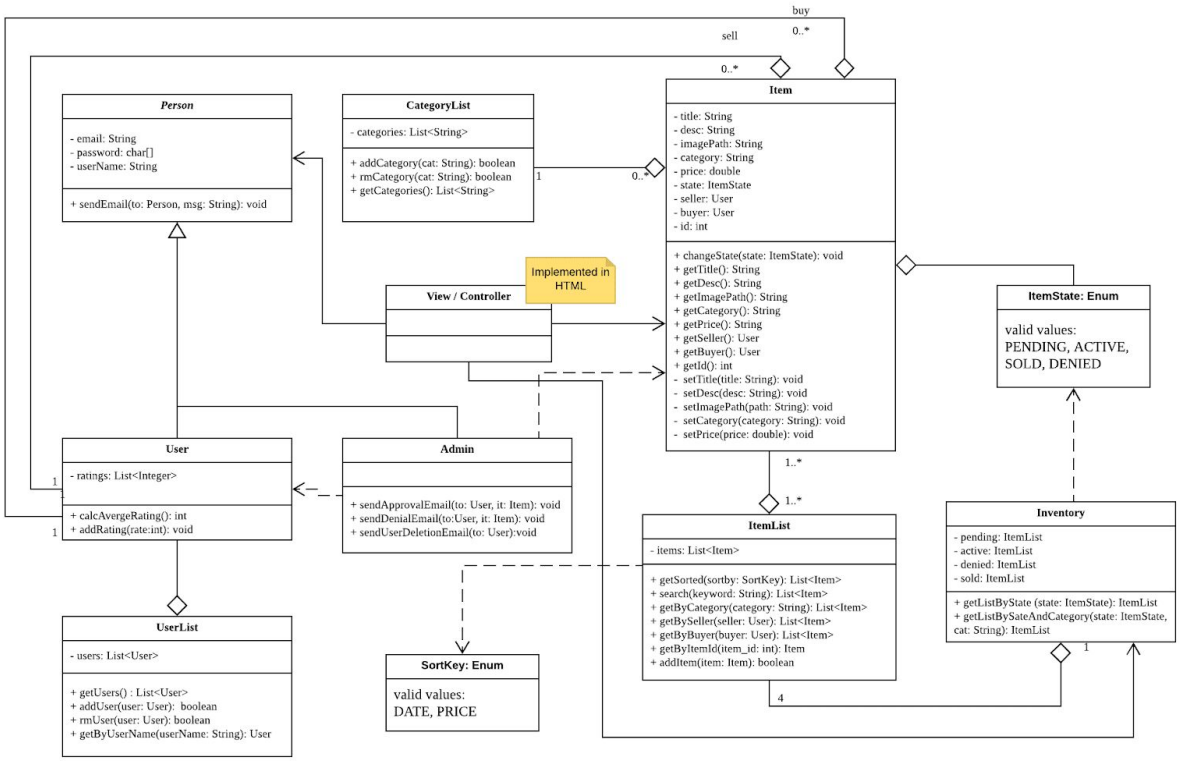
Vision:

A platform for CU students to post products they want to sell and organize meetups with potential buyers

Description

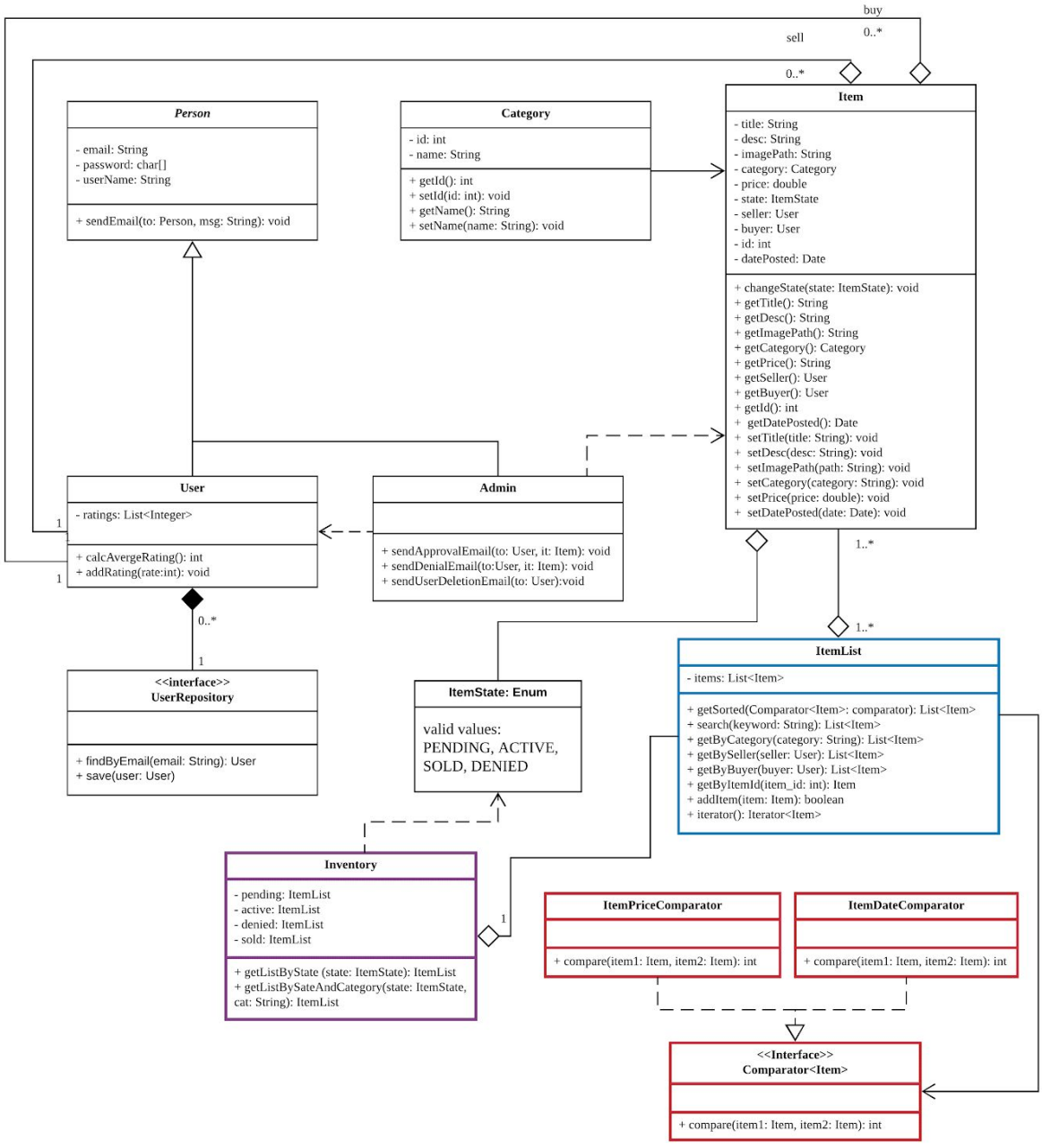
The main goal for this project is to make the useless resources (like old bikes or other products) recyclable and available among students and staff. This will be a platform for CU students to post products they want to sell and organize meetups with potential buyers.

2. Previous Class Diagram



3. Complete Diagram

Completed Class Diagram



4. Summary

In the last two weeks, we have implemented almost all of the original class diagram, set up a login page (including the HTML front end and a mock user), and refactored our class diagram to incorporate 3 design patterns.

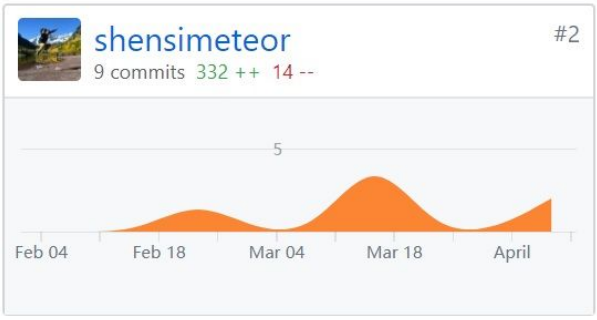
5. Breakdown

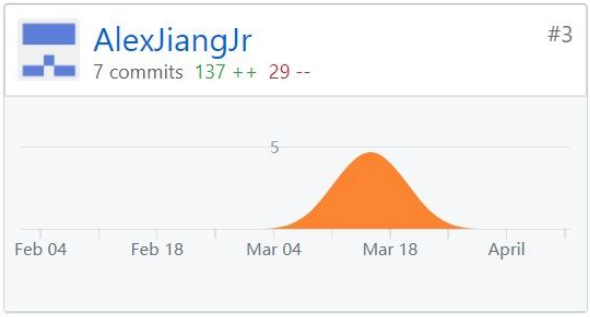
Si Shen: Set up comparator interface and wrote ItemPriceComparator. Created UserRepository

Yang Jiang: datecomparator class and part of iterator design pattern ..

Albert Dayn: Added login HTML page and secure login functionality to backend. Added iterator design pattern to Item list.

6. Github Graph





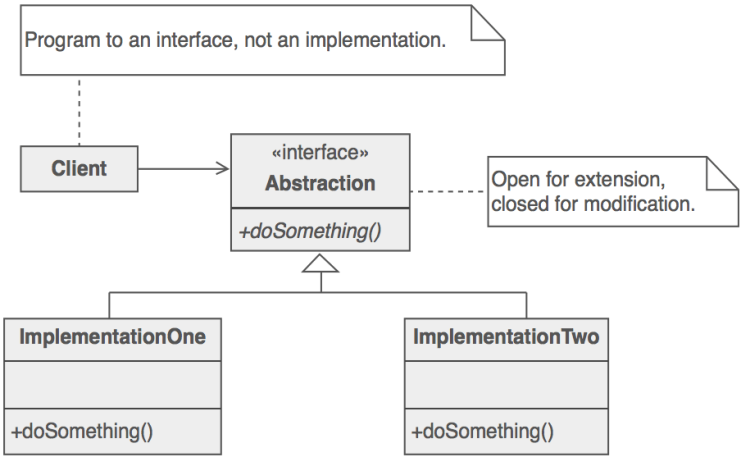
7. Estimate Remaining Effort

We have been mostly focused on the backend, so much of the front end has not yet been developed. We still need to add the capability to send emails from the backend, the ability to save securely hashed passwords for login, and implement many of the controllers.

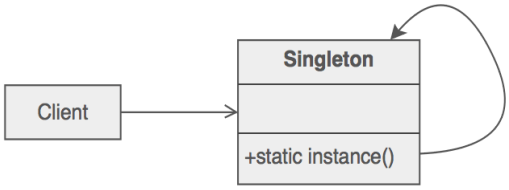
We also have yet to implement most of the HTML pages of the application and add CSS styling, so our web page still looks very bare, despite everything we've completed.

8. Design Patterns

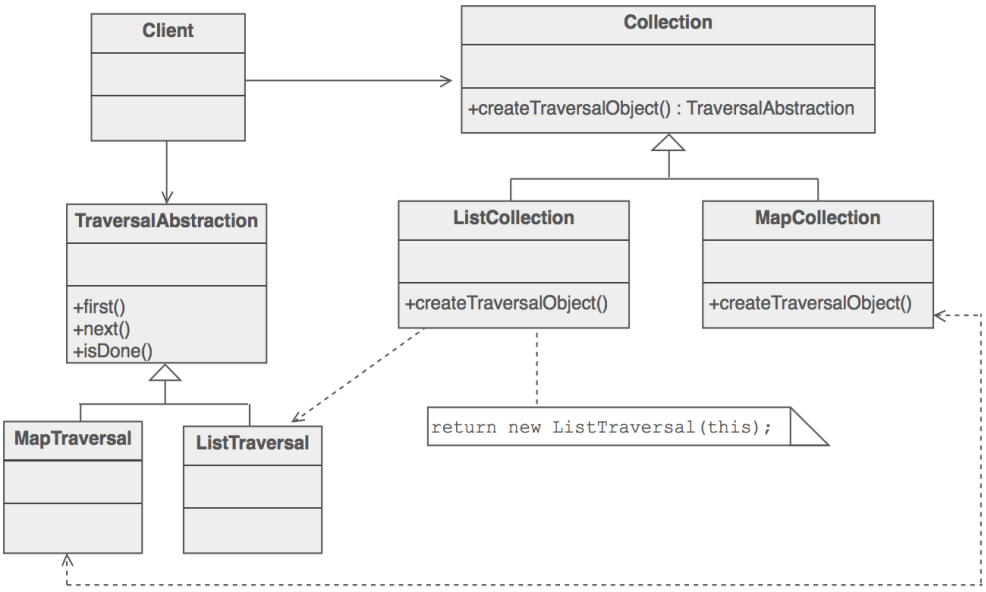
Strategy Design Pattern. The following figure demonstrates how this is routinely achieved - encapsulate interface details in a base class, and bury implementation details in derived classes. Clients can then couple themselves to an interface, and not have to experience the upheaval associated with change: no impact when the number of derived classes changes, and no impact when the implementation of a derived class changes. This design pattern is helpful for implementing ItemList.getSorted(..) method because we may have different sort key but we don't want if statements inside this method.



Singleton Design Pattern. Singleton design pattern means only one instance for this class. Singleton design pattern is used for lazy initialization, access protect and global access. We use this design pattern because our Inventory is unique so we want a global access to it. It is also helps for lazy initialization because it needs access the database which costs time. So lazy initialization helps.

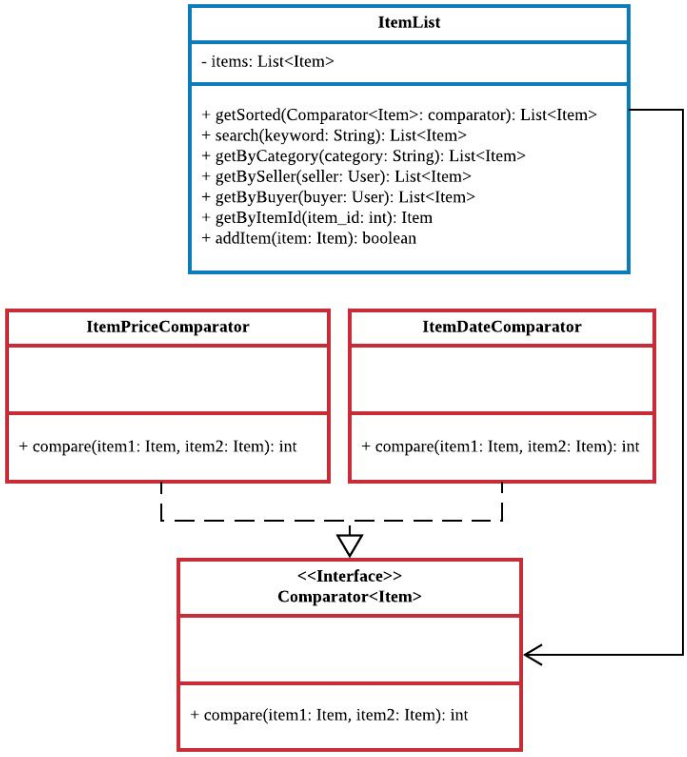


Iterator: The iterator design pattern allows us to abstract the concept of traversing the internal list, and give a generic method with which to access every element in the list. Although we do not have multiple data structures storing collections of items, we included this design pattern so an ItemList behaved as closely as possible to a List<Item>.



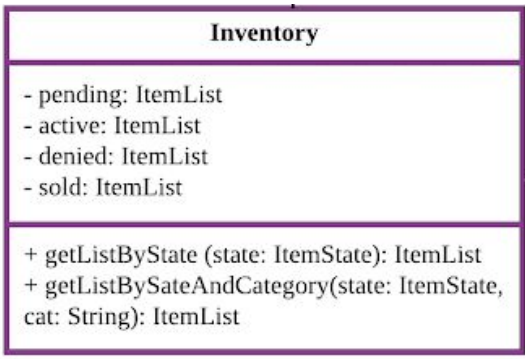
9. Portion of Design Pattern

Our portion class diagram for “Strategy Design Pattern”:



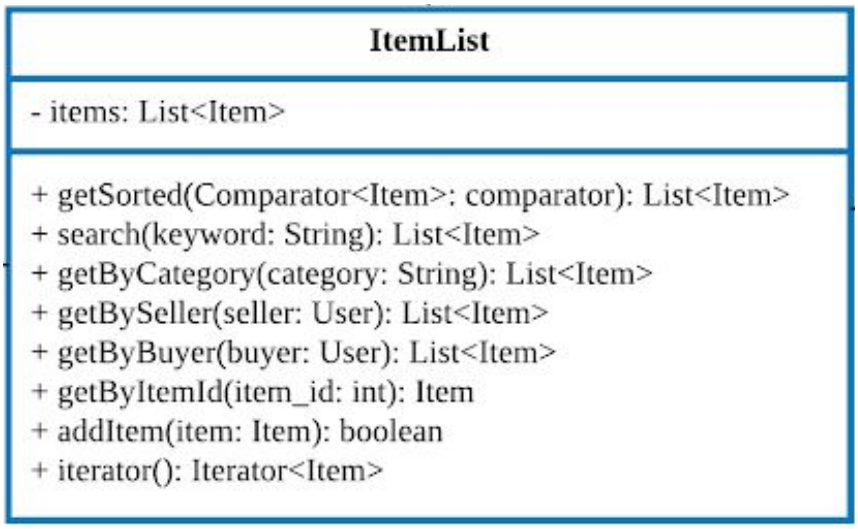
In this design pattern, Client is “ItemList”. It will use “Comparator<Item>” to do the “getSorted” method. Interface is the “Comparator<Item>”. Two implementations of the interface are “ItemPriceComparator” and “ItemDateComparator”. This design pattern helps us remove the “if statements” inside ItemList.getSorted(..) method. So if we want to add more different sortkey later, we don’t need to modify code inside ItemList.getSorted(..). We just need to pass a new Comparator instance to the method.

Singleton:



We always have exactly one object of the “Inventory” class, so we chose to use the singleton design pattern to represent it. We did not directly follow the class diagram shown above with the factory and a single static method, instead, we are utilizing the Spring Boot framework’s dependency injection to always give the single inventory object when it is autowired.

Iterator:



Again, we are leveraging Java’s internal iterator for lists to implement this design pattern, so some of the components in the class diagram for this design pattern above are done by Java. Our client will be the controllers, our traversalAbstraction is Iterator<Item>, our collection is ItemList, and the createTraversalObject() method is called iterator() in ItemList.

10. Final Iteration

We still need to finish our Spring MVC and hibernate. Also we are going to wrap up with each class functionality. To do it, we may finish controller classes. And we also maybe do refactoring for code. Overall, the bulk of the effort will go into tying everything together with controllers, and creating a view with HTML to interact with them. We feel confident that we will reach our target by the deadline.