CampusRecycle-36

Team members:

Si Shen
Yang Jiang
Albert Dayn

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

Implemented Features

Business Requirements:

ID	Description	Priority
BR-01	user must register with @colorado.edu email	Medium
BR-02	user must login before publish/view the products	High
BR-03	Admin can delete some product for (business reason e.g. illegal product)	Medium
BR-04	Admin can add / delete* categories (for business reason)	High
BR-05	Admin can delete a seller if they have bad reviews (for business reason)	Low

*: we did implement “Admin can add categories”, but we deleted Admin can delete categories since it’s not reasonable to delete the categories that already has products.

User Requirements:

ID	Description	Priority
US-01	User can view product list	High
US-02	User can view category list	High
US-03	User can view product detail page	High
US-04	User can sort product list	Medium

US-05	User can sort product list by date	Medium
US-06	User can sort product list by price	Medium
US-10	User can publish his/her own product (but need approval by admin before publish)	High
US-11	User can fill information (title, description,picture,category,price) about his/her own product	High
US-12	User can update his/her published product (but need approval by admin before publish)	Medium
US-13	User can delete his/her published product	High
US-14	User can mark his/her product as sold	High
US-15	User can send message to each other (via email)	High
US-16	Users can rate seller and buyer*	Low
US-17	Admin can view unprocessed product list	High
US-18	Admin can view unprocessed product detail	Medium
US-19	Admin can verify (approve/deny) product publishment	High
US-20	Admin can add category	High
US-22	Admin can delete user	Low

Functional Requirements:

ID	Description	Priority
FR-1	When a user registers, the system must check that the domain of the email is “colorado.edu”	High
FR-3	The system should store image, price, description, published date, category, sold flag of each item	High

Non Functional

ID	Description	Priority
NFR-01	Easy and intuitive to use	High
NFR-02	Easy to extend categories for admin	Medium
NFR-03	No operation should take more than 5 seconds	Medium

3

Not Implemented Features

Business Requirements:

ID	Description	Priority

User Requirements:

ID	Description	Priority
US-08	User can find product by keyword search	Medium
US-21	Admin can remove category	Medium

Functional Requirements:

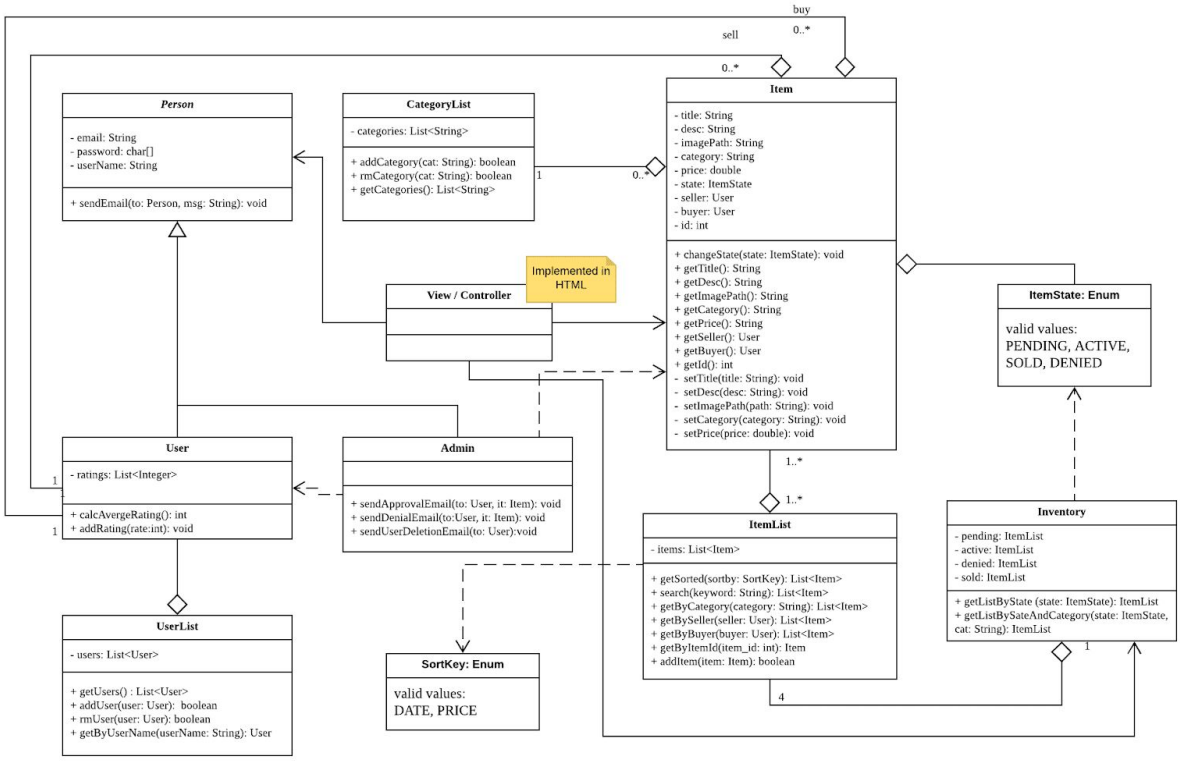
ID	Description	Priority
FR-2	The system must send a confirmation email to verify the user has that email address upon registration	High
FR-4	The system should send an email to the recipient whenever a message is sent on the website	High

Non Functional

ID	Description	Priority

4

Part 2 Class Diagram:



We had quite a few changes in our class diagram because our initial attempt was not quite as thought out as it could have been.

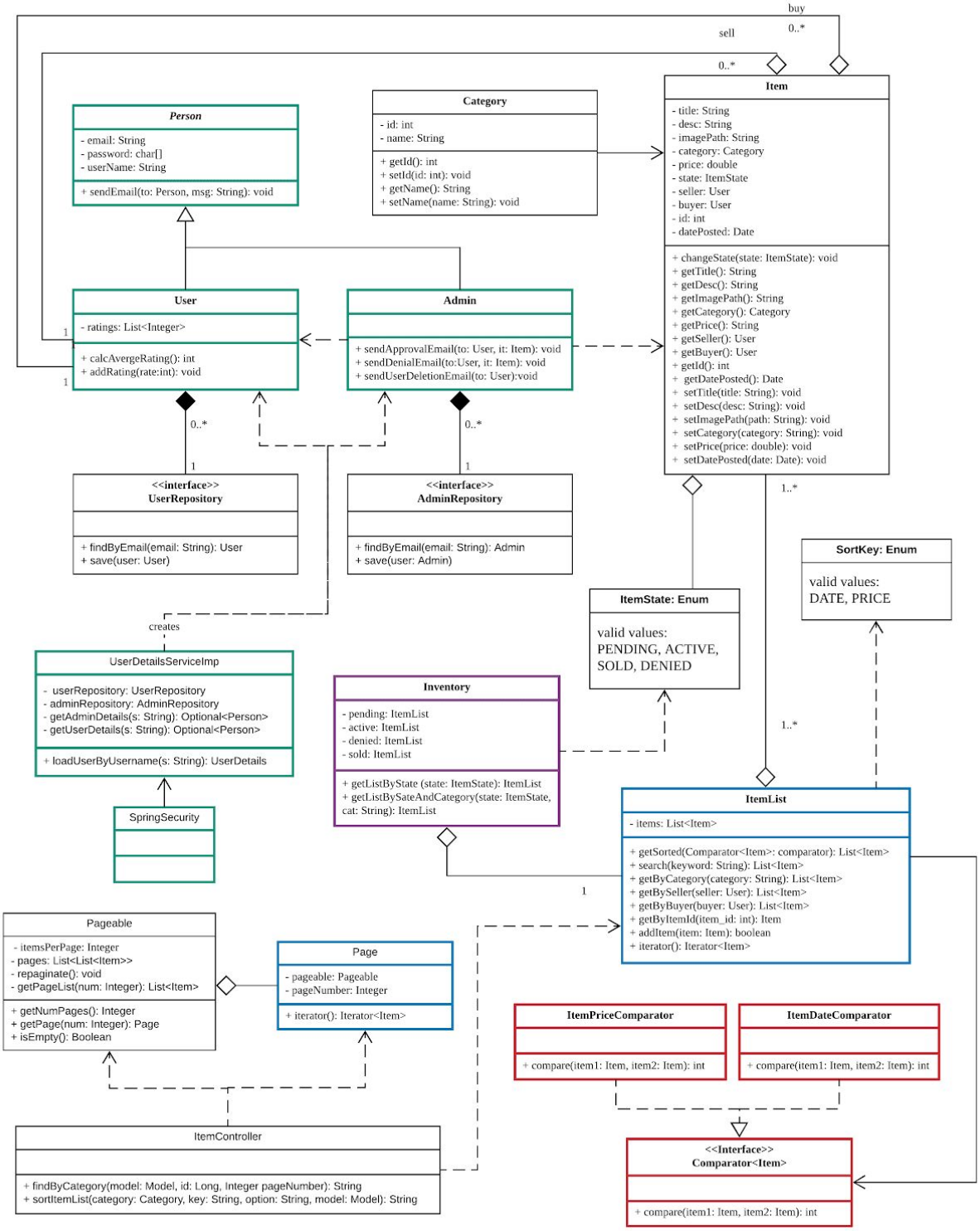
First, we did not consider that we would have to design the classes around the framework we were using. Spring Boot has requirements on how controllers, singletons, and security need to be implemented, and Hibernate affects the choice of which data goes where, as well as how it is accessed. To work with hibernate, we removed the UserList and instead use a UserRepository and AdminRepository. Next, to use Spring security, we made a UserDetailsService, which looks up users and admins by username. Overall, we enjoyed how much of the original architecture spring let us keep untouched, despite these required changes.

We also did not plan for efficiency on the front end. To avoid returning an endless list of posts when users browse items, we needed to implement a paging system. For this we introduced the Pageable class (to represent collections that can be split into pages), and the Page class (which represents a single page of this collection).

Through designing the system, we found that the factory design pattern would be useful in the UserDetailsService, and that pages should be iterable to be used in the views. Overall,

the new design patterns we learned helped keep changes encapsulated to their related classes, and kept us moving forward at a quick pace.

Final Class Diagram:

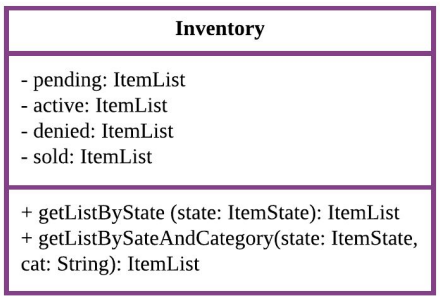


5

Design-Pattern-Implemented Class Diagram:

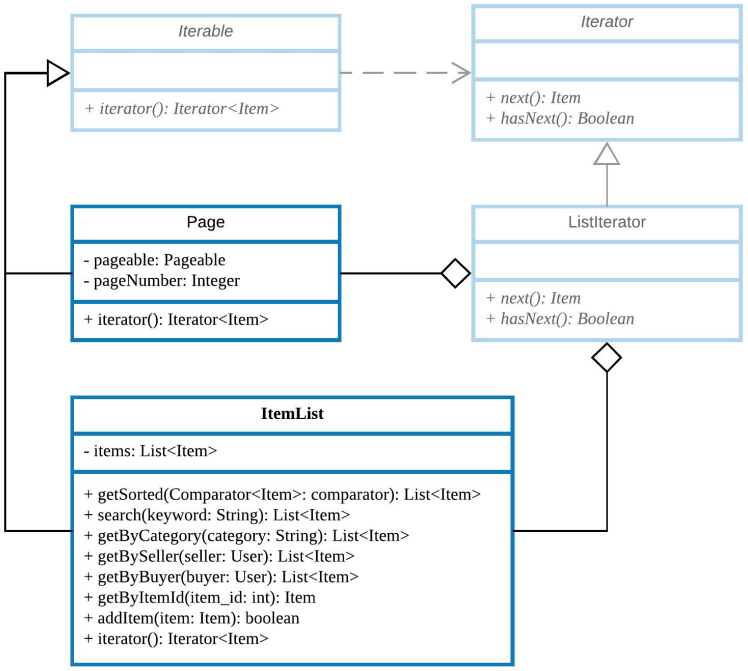
Singleton:

We decided to make the inventory a singleton object, since it would not make sense for item additions, deletions and searches to happen on multiple sets of items. Unlike most singleton UML diagrams, ours does not include a static getter method, as Spring's dependency inversion provides the singleton where it is needed.

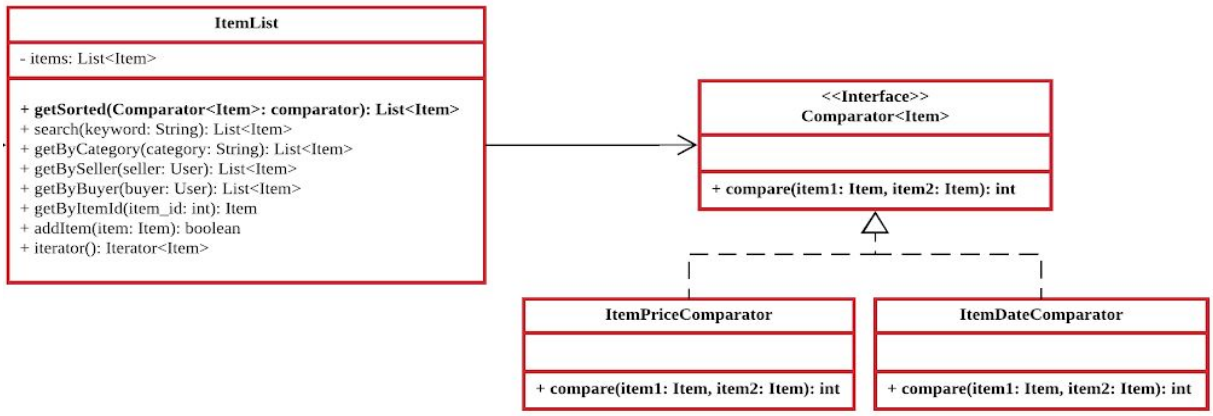


Iterator:

Since we were creating our own containers and needed to use them in our views (using Thymeleaf), we implemented the iterator interface for them. In the class diagram to the right, the faded classes are given by Java, so they are not seen in the Class diagram showing our entire system, but here they make clear that we were in fact using this design pattern. It helped us traverse our custom collections in a way that looked like we were using one of the built in collections.

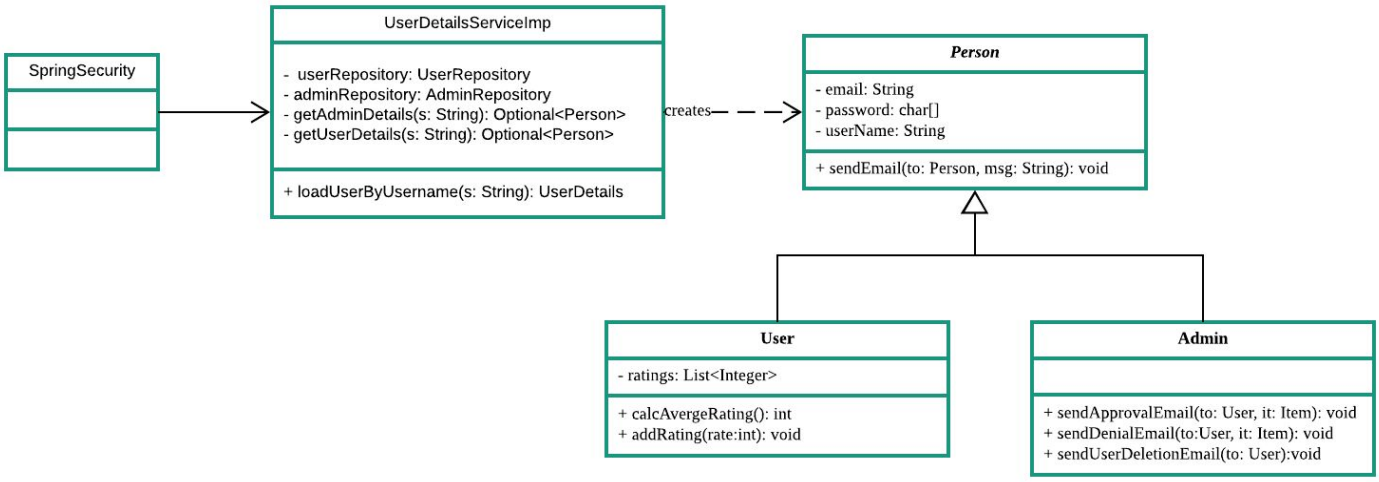


Strategy:



We used the strategy design pattern to enable selecting between sorting comparators at run time. Since the user can select either sort by price or date, we needed to have a way to change the functionality of sorting. We accomplished this by implementing the ItemPriceComparator and ItemDateComparator classes. Each of these implement the comparator interface for items, which is used by the ItemList to sort itself. The inventory then returns this sorted list.

Factory:



Once we started using Spring Security, we realized we needed to log users and admins in with the same function. To do this, we made the user construction process go through a factory design pattern. Based on the username given to the loadUserByUsername method, we either create an Admin or User object. If the security library created the user itself, it would have needed to know about the subclasses implementing the Person class. Instead, we use a factory to abstract the user creation.

6

What we have learned:

Creating good, bug free, and refactorable enterprise grade software is impossible without having a firm understanding of design patterns. In this small class project alone, we had to refactor our initial solution many times, and change our approach to problems to implement them in a way that follows a common pattern or practice to get around many issues.

We did not put enough thought into the technology stack we were using or the connections and requirements of system components that did not just include the server. For example, we did not look at the design of the Spring Boot framework before we picked it and designed our initial class diagram. Thus, when it came time to interface with features provided by the framework, such as the MVC architectural pattern or the security component, our design did not fit the pattern Spring was looking for. This required us to refactor not just implementations, but the entire API of multiple classes. If we hadn't followed the principle of encapsulation, this would have required a much greater amount of work than it ended up being.

We also did not consider that the front end view technology would need to be responsive even when many items were posted. We had to refactor our implementation of certain server API calls to implement paging so the user did not have to go through an absurdly long list of products.

This class has given us a background in the many object oriented concepts and software engineering principles that are needed to work as an enterprise level software developer.