Name: Ian Smith

Project Summary: An application to manage relationships with friends/family, where a user can track how often they reach out to/check in (via call/text) with the people that are most important to them. The goal of the project was to create a phonebook-like system that gives user a method to track their communication with people that matter to them.

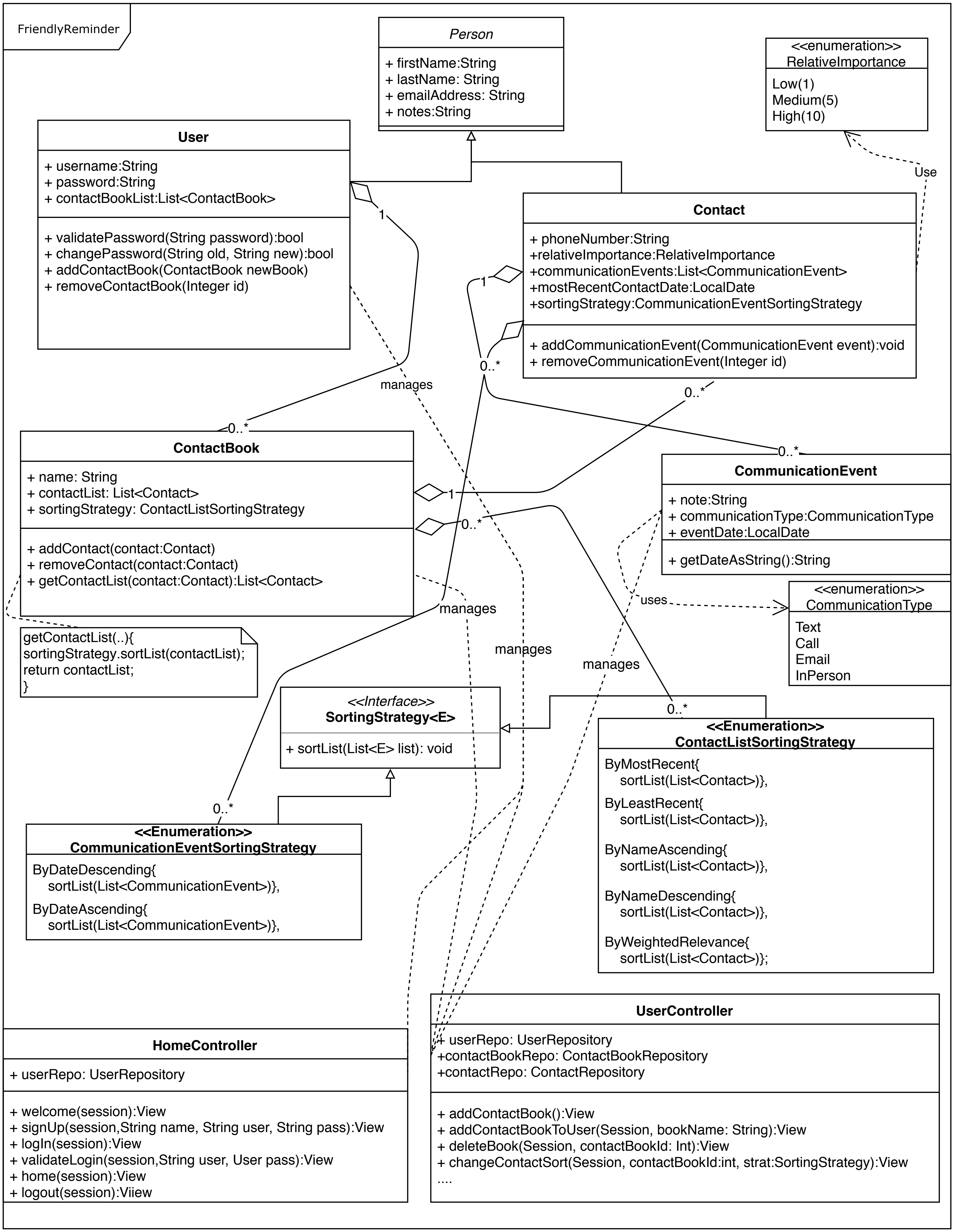
The system allows users to manage contacts, log communication events with those contacts and then generate a list of contacts to reach out to (sorted by importance of contact and time since last communication event).

List of Features Implemented:

ID	*User Requirement Description*
UR-001	User can sign up
UR-002	User can log in
	(b) User can log out from any screen
UR-003	User can create new contact book
	(b) User can delete existing contact book
UR-005	User can open contact book
	(b) User can view list of contacts in open contact book
	(c) User can sort list of contacts by name, importance, or date of last contact
	(NOTE: importance sort means at the top of the list should be contacts we haven't reached out to for a while, with preference given to contacts with higher importance rank)
UR-006	User can add new contact to a contact book
	(b) User can delete a contact from a contact book
	(c) User can update information for a contact in a contact book
UR-008	User can add a new communication event for a contact (information about a specific reaching out to that contact)
	(b) User can delete a communication event

List of Features NOT Implemented:
NONE. All of the features laid out in part 2 were implemented.

Final Class Diagram (next page):


FriendlyReminder
Person
+ firstName:String
+ lastName: String
+ emailAddress: String
+ notes:String
<<enumeration>>
RelativeImportance
Low(1)
Medium(5)
High(10)
Use
User
+ username:String
+ password:String
+ contactBookList:List<ContactBook>
+ validatePassword(String password):bool
+ changePassword(String old, String new):bool
+ addContactBook(ContactBook newBook)
+ removeContactBook(Integer id)
1
Contact
+ phoneNumber:String
+relativeImportance:RelativeImportance
+communicationEvents:List<CommunicationEvent>
+mostRecentContactDate:LocalDate
+sortingStrategy:CommunicationEventSortingStrategy
+ addCommunicationEvent(CommunicationEvent event):void
+ removeCommunicationEvent(Integer id)
1
0..*
0..*
manages
0..*
ContactBook
+ name: String
+ contactList: List<Contact>
+ sortingStrategy: ContactListSortingStrategy
+ addContact(contact:Contact)
+ removeContact(contact:Contact)
+ getContactList(contact:Contact):List<Contact>
1
0..*
0..*
CommunicationEvent
+ note:String
+ communicationType:CommunicationType
+ eventDate:LocalDate
+ getDateAsString():String
uses
<<enumeration>>
CommunicationType
Text
Call
Email
InPerson
manages
getContactList(..){
sortingStrategy.sortList(contactList);
return contactList;
}
manages
manages
<<Interface>>
SortingStrategy<E>
+ sortList(List<E> list): void
0..*
<<Enumeration>>
ContactListSortingStrategy
ByMostRecent{
sortList(List<Contact>)},
ByLeastRecent{
sortList(List<Contact>)},
ByNameAscending{
sortList(List<Contact>)},
ByNameDescending{
sortList(List<Contact>)},
ByWeightedRelevance{
sortList(List<Contact>)};
0..*
<<Enumeration>>
CommunicationEventSortingStrategy
ByDateDescending{
sortList(List<CommunicationEvent>)},
ByDateAscending{
sortList(List<CommunicationEvent>)},
UserController
+ userRepo: UserRepository
+contactBookRepo: ContactBookRepository
+contactRepo: ContactRepository
+ addContactBook():View
+ addContactBookToUser(Session, bookName: String):View
+ deleteBook(Session, contactBookId: Int):View
+ changeContactSort(Session, contactBookId:int, strat:SortingStrategy):View
....
HomeController
+ userRepo: UserRepository
+ welcome(session):View
+ signUp(session,String name, String user, String pass):View
+ logIn(session):View
+ validateLogin(session,String user, User pass):View
+ home(session):View
+ logout(session):Viiew

Final Class Diagram Continued:
What changed?

My final class diagram includes two additional controller classes, an additional enumeration class and an additional interface that is implemented by two `enum` types.

In my final class diagram, I no longer needed a `LoginSystem` class (as this is handled by the controllers) and I no longer needed a `DateTime` class as I opted for Java's `LocalDate` class.

Not included in the diagram are the 3 repository classes which are used in classes in my diagram, but which are implemented by the Spring framework. Further, my driver class Application with method `main()` is not included as this class is also implemented by the Spring framework.

These two controller classes were needed in the final class diagram in order to give the application real functionality. Before, the classes were sort of floating and no tasks were delegated: nothing was handling user input/actions. The other classes did not change much from the initial class diagram. Doing this work upfront made implementing these classes very easy and straightforward and saved lots of time in the IDE.

Design Pattern Implemented:
I chose to implement the Strategy design pattern for my project. The Strategy design pattern is useful when you some sort of Algorithm that changes depending on context. It enables selecting an algorithm at runtime as opposed to compile time.

In my project, I had the need to allow users to choose how they sort their contact lists and the communication events for each contact. Users could choose to sort events by increasing/decreasing date, and they could choose to sort the Contact lists within contact books themselves. They could sort contact lists by name and date, both ascending and descending, and they could also use a weighted relevance ranking that would sort contacts in order of which contact should be contacted next (taking into account the user defined importance for that specific contact and the time since that contact was last contacted). It is already easy to see how this could involve creating many different sorters that would have to be carefully managed. However, applying the Strategy pattern in this instance is perfect as the user (client) can choose which sorting algorithms they would like to use at run time. Further, they can sort different contact lists by different criteria! We see the design pattern followed by the pattern in my class diagram:

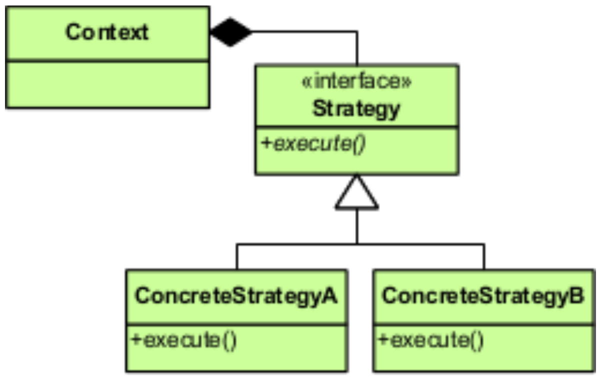
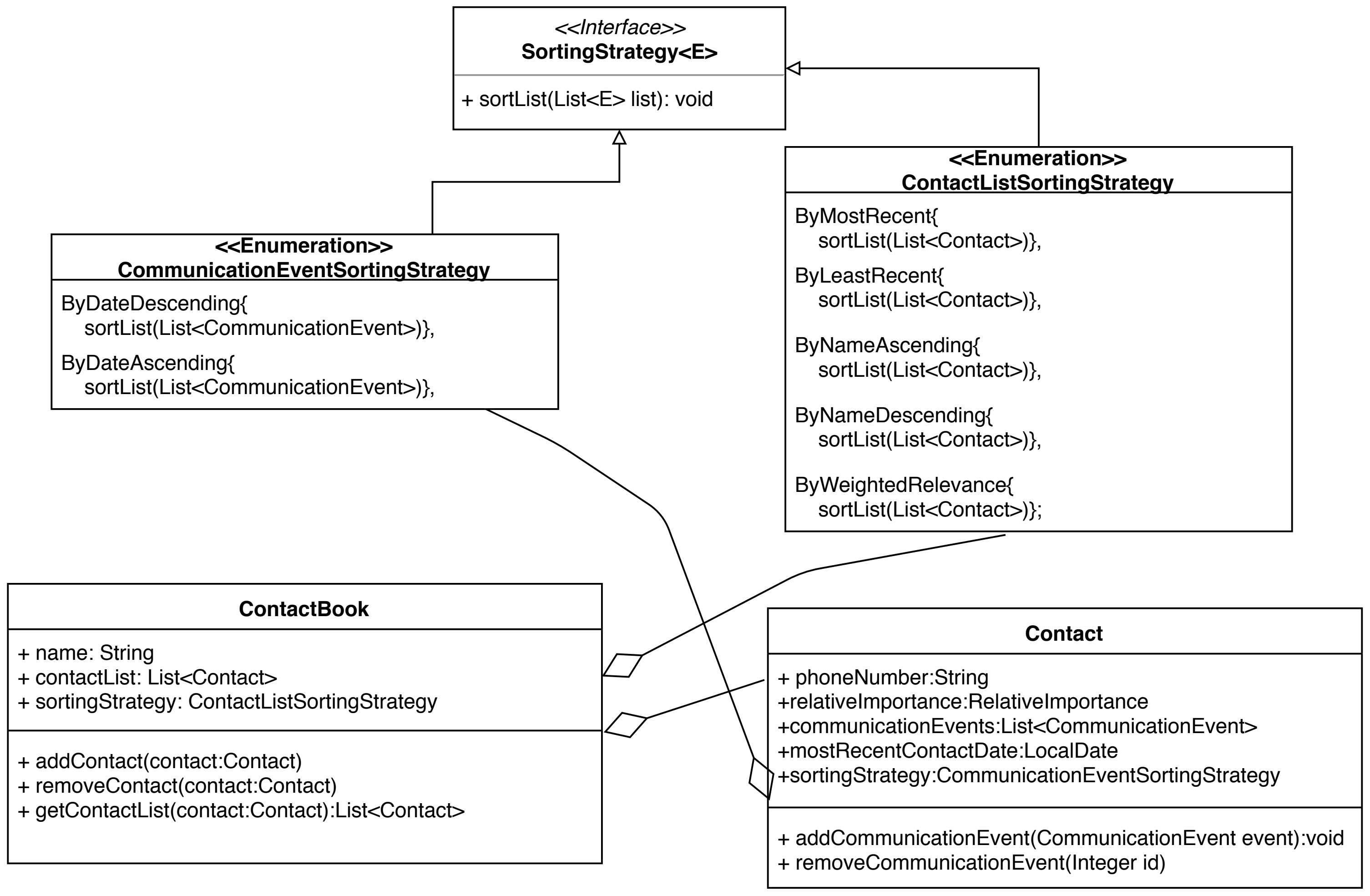
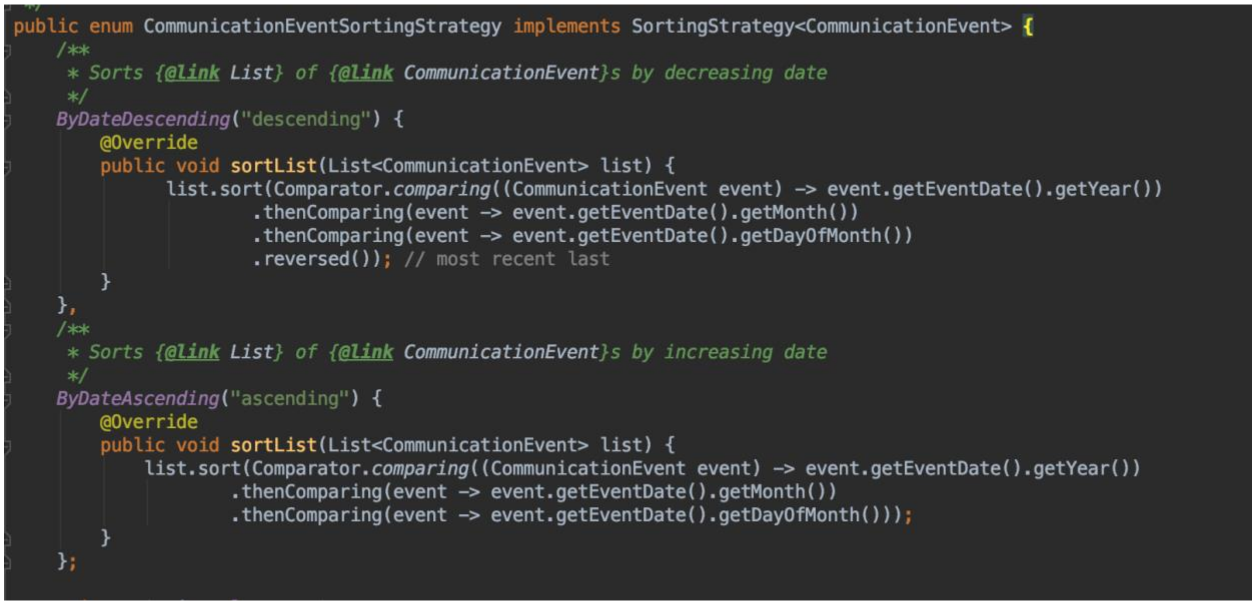


Figure 1: Strategy Design Pattern from Wikipedia


<<Interface>>
SortingStrategy<E>
+ sortList(List<E> list): void
<<Enumeration>>
ContactListSortingStrategy
ByMostRecent{
sortList(List<Contact>)},
ByLeastRecent{
sortList(List<Contact>)},
ByNameAscending{
sortList(List<Contact>)},
ByNameDescending{
sortList(List<Contact>)},
ByWeightedRelevance{
sortList(List<Contact>)};
<<Enumeration>>
CommunicationEventSortingStrategy
ByDateDescending{
sortList(List<CommunicationEvent>)},
ByDateAscending{
sortList(List<CommunicationEvent>)},
ContactBook
+ name: String
+ contactList: List<Contact>
+ sortingStrategy: ContactListSortingStrategy
+ addContact(contact:Contact)
+ removeContact(contact:Contact)
+ getContactList(contact:Contact):List<Contact>
Contact
+ phoneNumber:String
+relativeImportance:RelativeImportance
+communicationEvents:List<CommunicationEvent>
+mostRecentContactDate:LocalDate
+sortingStrategy:CommunicationEventSortingStrategy
+ addCommunicationEvent(CommunicationEvent event):void
+ removeCommunicationEvent(Integer id)

I implemented this design pattern with an Interface to define the method/algorithm that my Strategies should be required to implement (as the interface is just a contract). My `SortingStrategy` interface was generic and required a generic method, `sortList(List<E> list)`, and each class that implements this interface had to specify what type of List it would be sorting. Each strategy needed to take a list and sort it (it required no other information) and could thus be a static method. Further, I wanted to persist a User's choice of strategy but still ensure that I was not instantiating a new Strategy object for every Users' ContactBooks and every ContactBook's Contacts, thus I decided on making the method static. For each type of Sorter (`List<ContactBook>, List<Contact>`) it seemed helpful to simply implement the interface with two `enum` classes. Within each `enum` class, the `enum` constants would implement the abstract `sortList(List<E> list)` method in a different way. We could then choose concrete sort strategies by simply specifying the `enum` type corresponding to the desired concrete sort strategy and calling that method. Because `enum` constants have only one instance (the one defined in the `enum` class), these concrete strategies are essentially static and can be shared across all `Contact` and `ContactBook` classes. Setting a new strategy simply requires changes which `enum` type the class member `sortingStrategy` points to. A screenshot of one of these `enum` classes is included:



```java
public enum CommunicationEventSortingStrategy implements SortingStrategy<CommunicationEvent> {
    /**
     * Sorts {@link List} of {@link CommunicationEvent}s by decreasing date
     */
    ByDateDescending("descending") {
        @Override
        public void sortList(List<CommunicationEvent> list) {
            list.sort(Comparator.comparing((CommunicationEvent event) -> event.getEventDate().getYear())
                    .thenComparing(event -> event.getEventDate().getMonth())
                    .thenComparing(event -> event.getEventDate().getDayOfMonth())
                    .reversed()); // most recent last
        }
    },
    /**
     * Sorts {@link List} of {@link CommunicationEvent}s by increasing date
     */
    ByDateAscending("ascending") {
        @Override
        public void sortList(List<CommunicationEvent> list) {
            list.sort(Comparator.comparing((CommunicationEvent event) -> event.getEventDate().getYear())
                    .thenComparing(event -> event.getEventDate().getMonth())
                    .thenComparing(event -> event.getEventDate().getDayOfMonth()));
        }
    };
```

What I Learned:
From this process I learned that designing an OO-system that is easily extendable requires a lot of planning, even for a fairly simple project. I learned that planning out relationships between classes and how they will interact before you even start coding makes the coding process much clearer. Further, from this process I learned that while Design Patterns are well-documented and easy too understand in theory, it can actually take a bit of work to get the pattern to fit your framework and project properly. For example, my use of Hibernate made it difficult to persist the subclass type with a class that made use of Object composition. Further, I didn't want to have to create DB tables

filled with rows on rows of which ContactBook and which Contact used which type of sort strategy. Instead, researching this enum method solution taught me a lot about Java and a lot about the usefulness of being flexible with design pattern implementations. Overall, the project taught me a lot about planning and a lot about being flexible within your tools, frameworks and design pattern implementations because even if a pattern is a good fit for your application, the standard way of implementing it may not be!