Project Part 5: Final Report

Team: Walid Sharif, Yi Chen Kuo, Doug Falconieri
Title: Procurement System

Project Summary:

This project is to build a procurement system that manages the creation, approval, execution and tracking of purchase orders. The system will allow employees that need to purchase items for work to submit a purchase order describing what they need and why. The order will first be routed to the employee's manager to evaluate whether the order should be approved. If the order is approved, it is routed to a procurement officer who selects a vendor and purchases the item. The system collects data about the status of the purchase order including the cost of the items purchased, expected and actual delivery dates for the item and any defective items that are received. This data allows actors in the organization to track the status of orders, but also enables the aggregation of data across the whole organization such as overall spending and vendor performance that can be used to make the procurement process more efficient.

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

ID	Title	Description
US-01	Create Purchase Order	Employee can create a purchase order to acquire items needed to perform their job functions. If needed, they can save an order without submitting it and continue working on it at a later time.
US-02	Cancel Order	Employee can cancel the order if it no longer needed, Manager if the order is not approved, or Procurement Officer/Procurement Manager can cancel the order if is not correct.
US-03	Edit Purchase Order	An employee can edit information in existing

		purchase order that they created before.
US-04	Submit Purchase Order	The order has been stored in system.
US-06	Review Order	After employee submit their application form, Manager can determine whether the application is approved or not.
US-07	Select Vendor	A procurement officer selects the vendor for the purchase order based on the order’s items.
US-08	Add Vendor	Procurement officer adds a new vendor to the vendor database.
US-09	Execute Purchase	After ordering items associated with a purchase order, a procurement officer can update the order with the items’ final cost, tracking number and expected delivery date and move the order to “Ordered” status.

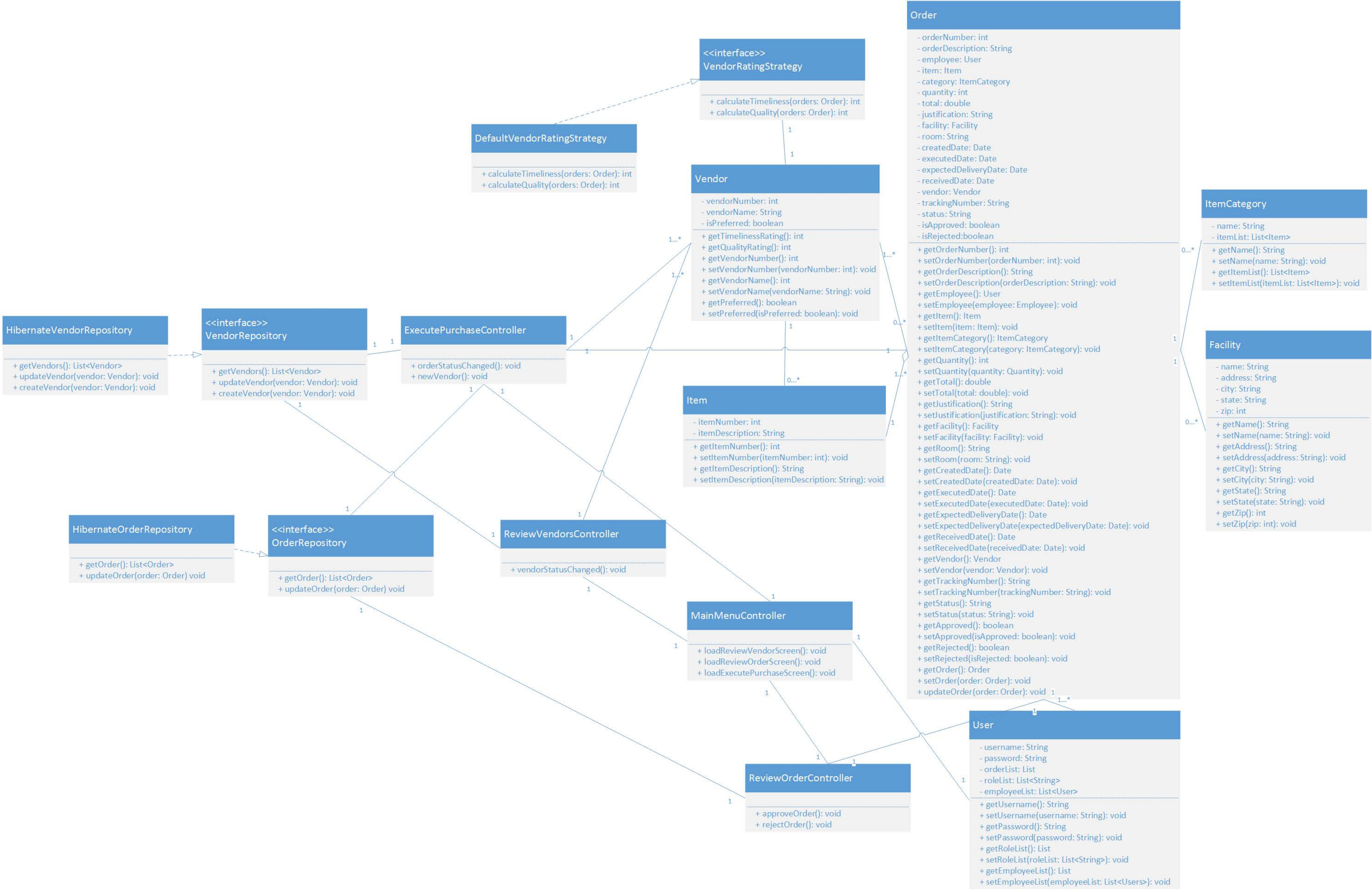
2. List the features were not implemented from Part 2 (table with ID and title).

ID	Title
US-05	Report Defect
US-10	Review Vendors
US-11	Log Item Receipt
US-12	Close Purchase Order

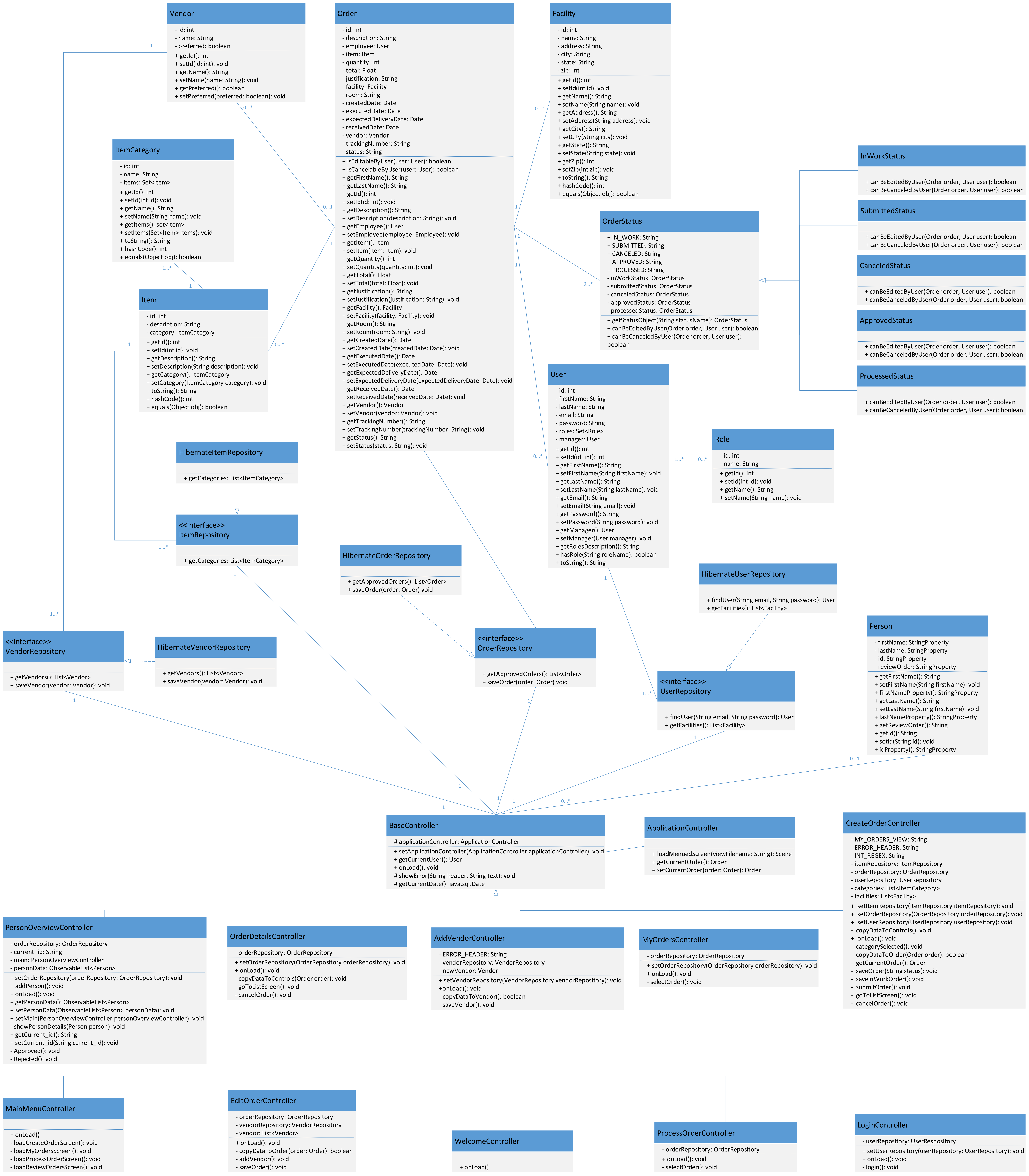
3. Show your Part 2 class diagram and your final class diagram. What changed? Why? If it did not change much, then discuss how doing the design up front helped in the development.

We made 4 main changes to the class diagram. First, we added the BaseController class in order to encapsulate the similar attributes and behavior of all controllers. Second, we added the ApplicationController class as a controller for the entire application. Some of its main responsibilities are to load the different views of the application as the user progresses in a particular workflow and get and set the order that the user is currently working on. Third, we added the Person class in order to list the Orders that need approval by the person that requested them. Finally, we added the OrderStatus class in order to change the ability of a user to edit or cancel their order depending on its state (canceled, submitted, approved, or processed). Our Part 2 class diagram and final class diagram are below.

Part 2 Class Diagram


Order
- orderNumber: int
- orderDescription: String
- employee: User
- item: Item
- category: ItemCategory
- quantity: int
- total: double
- justification: String
- facility: Facility
- room: String
- createdDate: Date
- executedDate: Date
- expectedDeliveryDate: Date
- receivedDate: Date
- vendor: Vendor
- trackingNumber: String
- status: String
- isApproved: boolean
- isRejected:boolean
+ getOrderNumber(): int
+ setOrderNumber(orderNumber: int): void
+ getOrderDescription(): String
+ setOrderDescription(orderDescription: String): void
+ getEmployee(): User
+ setEmployee(employee: Employee): void
+ getItem(): Item
+ setItem(item: Item): void
+ getItemCategory(): ItemCategory
+ setItemCategory(category: ItemCategory): void
+ getQuantity(): int
+ setQuantity(quantity: Quantity): void
+ getTotal(): double
+ setTotal(total: double): void
+ getJustification(): String
+ setJustification(justification: String): void
+ getFacility(): Facility
+ setFacility(facility: Facility): void
+ getRoom(): String
+ setRoom(room: String): void
+ getCreatedDate(): Date
+ setCreatedDate(createdDate: Date): void
+ getExecutedDate(): Date
+ setExecutedDate(executedDate: Date): void
+ getExpectedDeliveryDate(): Date
+ setExpectedDeliveryDate(expectedDeliveryDate: Date): void
+ getReceivedDate(): Date
+ setReceivedDate(receivedDate: Date): void
+ getVendor(): Vendor
+ setVendor(vendor: Vendor): void
+ getTrackingNumber(): String
+ setTrackingNumber(trackingNumber: String): void
+ getStatus(): String
+ setStatus(status: String): void
+ getApproved(): boolean
+ setApproved(isApproved: boolean): void
+ getRejected(): boolean
+ setRejected(isRejected: boolean): void
+ getOrder(): Order
+ setOrder(order: Order): void
+ updateOrder(order: Order): void
<<interface>>
VendorRatingStrategy
+ calculateTimeliness(orders: Order): int
+ calculateQuality(orders: Order): int
DefaultVendorRatingStrategy
+ calculateTimeliness(orders: Order): int
+ calculateQuality(orders: Order): int
Vendor
- vendorNumber: int
- vendorName: String
- isPreferred: boolean
+ getTimelinessRating(): int
+ getQualityRating(): int
+ getVendorNumber(): int
+ setVendorNumber(vendorNumber: int): void
+ getVendorName(): int
+ setVendorName(vendorName: String): void
+ getPreferred(): boolean
+ setPreferred(isPreferred: boolean): void
ItemCategory
- name: String
- itemList: List<Item>
+ getName(): String
+ setName(name: String): void
+ getItemList(): List<Item>
+ setItemList(itemList: List<Item>): void
Facility
- name: String
- address: String
- city: String
- state: String
- zip: int
+ getName(): String
+ setName(name: String): void
+ getAddress(): String
+ setAddress(address: String): void
+ getCity(): String
+ setCity(city: String): void
+ getState(): String
+ setState(state: String): void
+ getZip(): int
+ setZip(zip: int): void
HibernateVendorRepository
+ getVendors(): List<Vendor>
+ updateVendor(vendor: Vendor): void
+ createVendor(vendor: Vendor): void
<<interface>>
VendorRepository
+ getVendors(): List<Vendor>
+ updateVendor(vendor: Vendor): void
+ createVendor(vendor: Vendor): void
ExecutePurchaseController
+ orderStatusChanged(): void
+ newVendor(): void
Item
- itemNumber: int
- itemDescription: String
+ getItemNumber(): int
+ setItemNumber(itemNumber: int): void
+ getItemDescription(): String
+ setItemDescription(itemDescription: String): void
HibernateOrderRepository
+ getOrder(): List<Order>
+ updateOrder(order: Order) void
<<interface>>
OrderRepository
+ getOrder(): List<Order>
+ updateOrder(order: Order) void
ReviewVendorsController
+ vendorStatusChanged(): void
MainMenuController
+ loadReviewVendorScreen(): void
+ loadReviewOrderScreen(): void
+ loadExecutePurchaseScreen(): void
User
- username: String
- password: String
- orderList: List
- roleList: List<String>
- employeeList: List<User>
+ getUsername(): String
+ setUsername(username: String): void
+ getPassword(): String
+ setPassword(password: String): void
+ getRoleList(): List
+ setRoleList(roleList: List<String>): void
+ getEmployeeList(): List
+ setEmployeeList(employeeList: List<Users>): void
ReviewOrderController
+ approveOrder(): void
+ rejectOrder(): void
1
1
1...*
1...*
1...*
0...*
1
0...*
1
1...*
1
0...*
1
1
0...*
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1...*
1
1
1

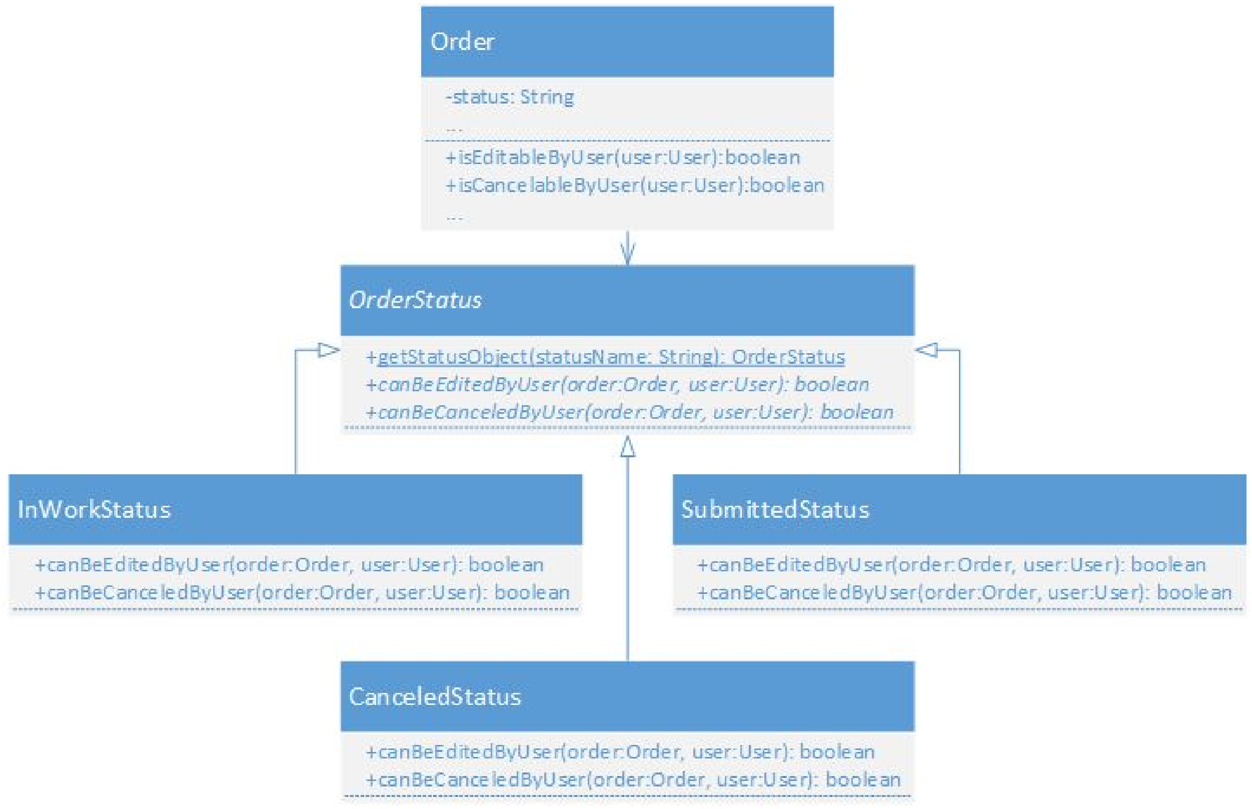
Final Class Diagram


Vendor
- id: int
- name: String
- preferred: boolean
+ getId(): int
+ setId(id: int): void
+ getName(): String
+ setName(name: String): void
+ getPreferred(): boolean
+ setPreferred(preferred: boolean): void
Order
- id: int
- description: String
- employee: User
- item: Item
- quantity: int
- total: Float
- justification: String
- facility: Facility
- room: String
- createdDate: Date
- executedDate: Date
- expectedDeliveryDate: Date
- receivedDate: Date
- vendor: Vendor
- trackingNumber: String
- status: String
+ isEditableByUser(user: User): boolean
+ isCancelableByUser(user: User): boolean
+ getFirstName(): String
+ getLastName(): String
+ getId(): int
+ setId(id: int): void
+ getDescription(): String
+ setDescription(description: String): void
+ getEmployee(): User
+ setEmployee(employee: Employee): void
+ getItem(): Item
+ setItem(item: Item): void
+ getQuantity(): int
+ setQuantity(quantity: int): void
+ getTotal(): Float
+ setTotal(total: Float): void
+ getJustification(): String
+ setJustification(justification: String): void
+ getFacility(): Facility
+ setFacility(facility: Facility): void
+ getRoom(): String
+ setRoom(room: String): void
+ getCreatedDate(): Date
+ setCreatedDate(createdDate: Date): void
+ getExecutedDate(): Date
+ setExecutedDate(executedDate: Date): void
+ getExpectedDeliveryDate(): Date
+ setExpectedDeliveryDate(expectedDeliveryDate: Date): void
+ getReceivedDate(): Date
+ setReceivedDate(receivedDate: Date): void
+ getVendor(): Vendor
+ setVendor(vendor: Vendor): void
+ getTrackingNumber(): String
+ setTrackingNumber(trackingNumber: String): void
+ getStatus(): String
+ setStatus(status: String): void
Facility
- id: int
- name: String
- address: String
- city: String
- state: String
- zip: int
+ getId(): int
+ setId(int id): void
+ getName(): String
+ setName(String name): void
+ getAddress(): String
+ setAddress(String address): void
+ getCity(): String
+ setCity(String city): void
+ getState(): String
+ setState(String state): void
+ getZip(): int
+ setZip(int zip): void
+ toString(): String
+ hashCode(): int
+ equals(Object obj): boolean
ItemCategory
- id: int
- name: String
- items: Set<Item>
+ getId(): int
+ setId(int id): void
+ getName(): String
+ setName(String name): void
+ getItems(): set<Item>
+ setItems(Set<Item> items): void
+ toString(): String
+ hashCode(): int
+ equals(Object obj): boolean
InWorkStatus
+ canBeEditedByUser(Order order, User user): boolean
+ canBeCanceledByUser(Order order, User user): boolean
SubmittedStatus
+ canBeEditedByUser(Order order, User user): boolean
+ canBeCanceledByUser(Order order, User user): boolean
OrderStatus
+ IN_WORK: String
+ SUBMITTED: String
+ CANCELED: String
+ APPROVED: String
+ PROCESSED: String
- inWorkStatus: OrderStatus
- submittedStatus: OrderStatus
- canceledStatus: OrderStatus
- approvedStatus: OrderStatus
- processedStatus: OrderStatus
+ getStatusObject(String statusName): OrderStatus
+ canBeEditedByUser(Order order, User user): boolean
+ canBeCanceledByUser(Order order, User user): boolean
CanceledStatus
+ canBeEditedByUser(Order order, User user): boolean
+ canBeCanceledByUser(Order order, User user): boolean
ApprovedStatus
+ canBeEditedByUser(Order order, User user): boolean
+ canBeCanceledByUser(Order order, User user): boolean
ProcessedStatus
+ canBeEditedByUser(Order order, User user): boolean
+ canBeCanceledByUser(Order order, User user): boolean
Item
- id: int
- description: String
- category: ItemCategory
+ getId(): int
+ setId(int id): void
+ getDescription(): String
+ setDescription(String description): void
+ getCategory(): ItemCategory
+ setCategory(ItemCategory category): void
+ toString(): String
+ hashCode(): int
+ equals(Object obj): boolean
User
- id: int
- firstName: String
- lastName: String
- email: String
- password: String
- roles: Set<Role>
- manager: User
+ getId(): int
+ setId(id: int): int
+ getFirstName(): String
+ setFirstName(String firstName): void
+ getLastName(): String
+ setLastName(String lastName): void
+ getEmail(): String
+ setEmail(String email): void
+ getPassword(): String
+ setPassword(String password): void
+ getManager(): User
+ setManager(User manager): void
+ getRolesDescription(): String
+ hasRole(String roleName): boolean
+ toString(): String
Role
- id: int
- name: String
+ getId(): int
+ setId(int id): void
+ getName(): String
+ setName(String name): void
HibernateItemRepository
+ getCategories(): List<ItemCategory>
<<interface>>
ItemRepository
+ getCategories(): List<ItemCategory>
HibernateOrderRepository
+ getApprovedOrders(): List<Order>
+ saveOrder(order: Order) void
HibernateUserRepository
+ findUser(String email, String password): User
+ getFacilities(): List<Facility>
<<interface>>
VendorRepository
+ getVendors(): List<Vendor>
+ saveVendor(vendor: Vendor): void
HibernateVendorRepository
+ getVendors(): List<Vendor>
+ saveVendor(vendor: Vendor): void
<<interface>>
OrderRepository
+ getApprovedOrders(): List<Order>
+ saveOrder(order: Order) void
<<interface>>
UserRepository
+ findUser(String email, String password): User
+ getFacilities(): List<Facility>
Person
- firstName: StringProperty
- lastName: StringProperty
- id: StringProperty
- reviewOrder: StringProperty
+ getFirstName(): String
+ setFirstName(String firstName): void
+ firstNameProperty(): StringProperty
+ getLastName(): String
+ setLastName(String firstName): void
+ lastNameProperty(): StringProperty
+ getReviewOrder(): String
+ getid(): String
+ setid(String id): void
+ idProperty(): StringProperty
BaseController
# applicationController: ApplicationController
+ setApplicationController(ApplicationController applicationController): void
+ getCurrentUser(): User
+ onLoad(): void
# showError(String header, String text): void
# getCurrentDate(): java.sql.Date
ApplicationController
+ loadMenuedScreen(viewFilename: String): Scene
+ getCurrentOrder(): Order
+ setCurrentOrder(order: Order): Order
CreateOrderController
- MY_ORDERS_VIEW: String
- ERROR_HEADER: String
- INT_REGEX: String
- itemRepository: ItemRepository
- orderRepository: OrderRepository
- userRepository: UserRepository
- categories: List<ItemCategory>
- facilities: List<Facility>
+ setItemRepository(ItemRepository itemRepository): void
+ setOrderRepository(OrderRepository orderRepository): void
+ setUserRepository(UserRepository userRepository): void
- copyDataToControls(): void
+ onLoad(): void
- categorySelected(): void
- copyDataToOrder(Order order): boolean
- getCurrentOrder(): Order
- saveOrder(String status): void
- saveInWorkOrder(): void
- submitOrder(): void
- goToListScreen(): void
- cancelOrder(): void
PersonOverviewController
- orderRepository: OrderRepository
- current_id: String
- main: PersonOverviewController
- personData: ObservableList<Person>
+ setOrderRepository(orderRepository: OrderRepository): void
+ addPerson(): void
+ onLoad(): void
+ getPersonData(): ObservableList<Person>
+ setPersonData(ObservableList<Person> personData): void
+ setMain(PersonOverviewController personOverviewController): void
- showPersonDetails(Person person): void
+ getCurrent_id(): String
+ setCurrent_id(String current_id): void
- Approved(): void
- Rejected(): void
OrderDetailsController
- orderRepository: OrderRepository
+ setOrderRepository(OrderRepository orderRepository): void
+ onLoad(): void
- copyDataToControls(Order order): void
- goToListScreen(): void
- cancelOrder(): void
AddVendorController
- ERROR_HEADER: String
- vendorRepository: VendorRepository
- newVendor: Vendor
+ setVendorRepository(VendorRepository vendorRepository): void
+onLoad(): void
- copyDataToVendor(): boolean
- saveVendor(): void
MyOrdersController
- orderRepository: OrderRepository
+ setOrderRepository(OrderRepository orderRepository): void
+ onLoad(): void
- selectOrder(): void
MainMenuController
+ onLoad()
- loadCreateOrderScreen(): void
- loadMyOrdersScreen(): void
- loadProcessOrderScreen(): void
- loadReviewOrdersScreen(): void
EditOrderController
- orderRepository: OrderRepository
- vendorRepository: VendorRepository
- vendor: List<Vendor>
+ onLoad(): void
- copyDataToOrder(order: Order): boolean
- addVendor(): void
- saveOrder(): void
WelcomeController
+ onLoad()
ProcessOrderController
- orderRepository: OrderRepository
+ onLoad(): void
- selectOrder(): void
LoginController
- userRepository: UserRespository
+ setUserRepository(userRepository: UserRepository): void
+ onLoad(): void
- login(): void
1
0...*
0...1
1
0...*
1
1
1
0...*
1...*
1
1
0...*
1...*
0...*
1...*
1
1...*
1
1
1
1
1
1
1...*
1
0...1
0...*
1
1

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.

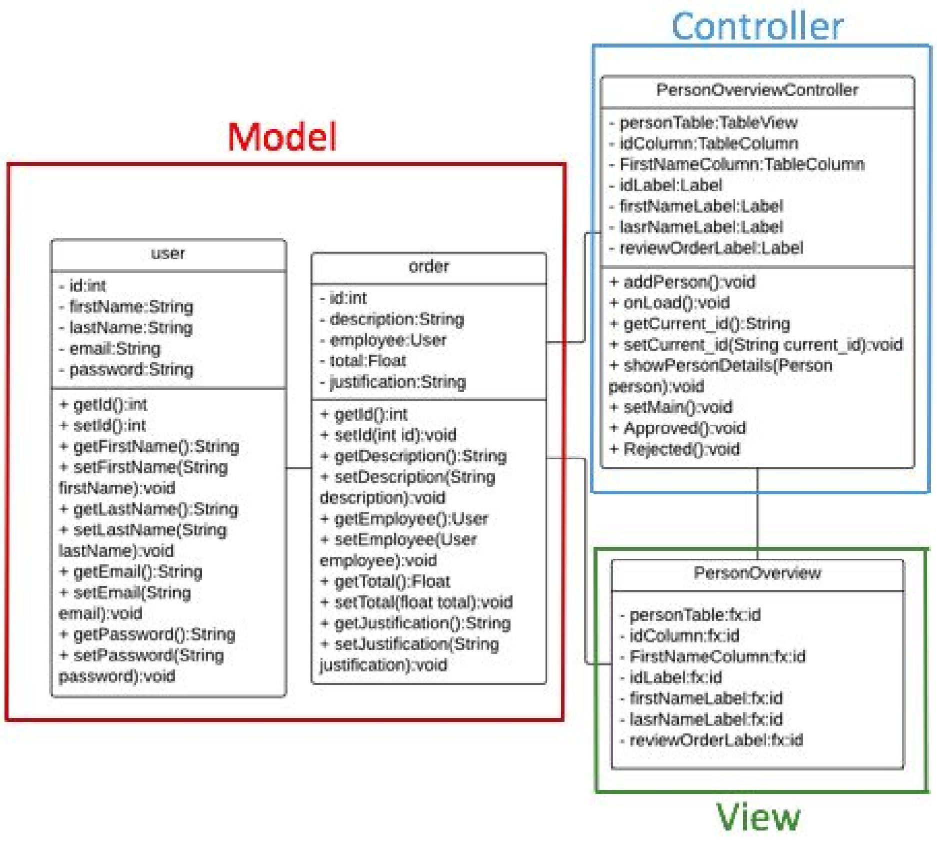
Our project made use of several design and architectural patterns that we covered in class this semester. Perhaps the most important pattern for our project is the State pattern. Our application is essentially a large multi-user workflow for creating, approving and executing purchase orders. At every point in the workflow, the application must decide which modifications the current user is allowed to make to the current purchase order. This decision is based on a number of variables including the current state of the order, the roles that the current user has, whether the current user created the order and whether the current user is the manager of the current user. A naive way of implementing this logic would be to use a complex set of nested if and else statements in each place in the application where an access decision needs to be made. However, that would be an example of the Arrowhead design anti-pattern, a complex nest of conditional logic that is difficult to understand or modify. In addition, several screens in the application need to make similar access control decisions. If the same conditional logic is copied to each screen's controller class, it could lead to the *duplicate code* code smell unless the shared logic is refactored out to a separate class where it can be shared between multiple controllers.

Introducing the State pattern to our application solved both of these problems. We implemented the pattern by creating a set of classes to represent all of the states that a purchase order can be in during our workflow such as *InWorkStatus* and *SubmittedStatus*. Each of these state classes extends from an abstract base class called *OrderStatus* which defines several abstract methods like *canBeEditedByUser(Order order, User user)* and *canBeCanceledByUser(Order order, User user)* that each subclass must implement to make access decisions. These decision methods return a boolean indicating whether or not the current user can perform the specified action to the current order. The logic in each subclass' implementations is much simpler than the original nested if-statements because it only needs to handle a single purchase order state so it can focus on the other variables like the current user's roles. Each decision method takes a *User* object representing the current user as a parameter so it can check the current user's roles and relationship to the current order. In order to hide complexity, controllers in the application that need to enforce access control decisions do not access the state objects directly. Instead, they call simple methods on the current *Order* class like *isEditableByUser(User user)*. Behind the scenes, the *Order* class implements these methods by calling a factory method on the *OrderStatus* class called *getStatusObject(String statusName)* which returns the correct subclass of *OrderStatus* for the order's current state. This factory method is one reason that *OrderStatus* is an abstract class and not an interface. Because the state objects contain decision logic, but no nested state of their own, they are implemented as singleton objects. The *OrderStatus* class keeps a reference to one instance of each state object that is reused over and over again to prevent duplicate object creation. Once the *Order* class gets the appropriate *OrderStatus* object, it delegates its decision by calling the appropriate decision method on the status object and returning the result. Because the decision methods of the *Order* class can be called by any controller, the issue of duplication code is also prevented.


Order
-status: String
...
+isEditableByUser(user:User):boolean
+isCancelableByUser(user:User):boolean
...
OrderStatus
+getStatusObject(statusName: String): OrderStatus
+canBeEditedByUser(order:Order, user:User): boolean
+canBeCanceledByUser(order:Order, user:User): boolean
InWorkStatus
+canBeEditedByUser(order:Order, user:User): boolean
+canBeCanceledByUser(order:Order, user:User): boolean
SubmittedStatus
+canBeEditedByUser(order:Order, user:User): boolean
+canBeCanceledByUser(order:Order, user:User): boolean
CanceledStatus
+canBeEditedByUser(order:Order, user:User): boolean
+canBeCanceledByUser(order:Order, user:User): boolean

Our application also makes use of three higher-level architectural patterns: Model-View-Controller (MVC), Object-Relational Mapping (ORM) and Dependency Injection. The views in our application are implemented using the JavaFX framework. JavaFX allows Java client user interfaces to be defined using XML files instead of Java code which is more concise. The XML files can specify a controller class to manage the view. When the view is loaded JavaFX will instantiate an instance of the controller class and create linkages so that the controller can read and modify the view's controls. Unlike some MVC implementations, our version does not use the Observer pattern to update views in real-time in response to changes in the data. Instead, the controller updates the view when the view is initially loaded and also in response to button presses by the user. This works well for this application because the workflow rules generally only allow one person to modify an order during each workflow state. The controllers do not directly access the application database, but instead do so indirectly using the model part of our application. Our model consists of two groups of classes. The classes in the *models* package represent business objects like purchase orders, users, items, facilities and vendors. The classes in the *repositories* package contain logic for saving and loading the model objects to and from the database. The repositories are an example of the Data Access Object (DAO) pattern. Internally, they are implemented using the Hibernate ORM framework. There is one repository for each major portion of the database such as purchase orders and users. Each repository consists of an interface defining what operations are available and an implementation that uses Hibernate to implement the operations. The controllers only access the repositories through their interfaces and have no knowledge of the implementation classes. The view, controller and model layers are all linked together in a decoupled way using Dependency Injection with the Spring framework. We override JavaFX's view loading system to load each view's controller from our Spring application context instead of JavaFX's default behavior of creating the controller object by calling its constructor. We also configure Spring to inject any needed repositories into each controller. The repositories in turn are injected with a Hibernate session factory which is also configured in the Spring configuration file.

These larger architectural patterns are made up of several design patterns. Spring is basically a more full-featured version of the Factory pattern since it allows object classes in the application to obtain instances of the objects they depend on in a decoupled way and access them only through their interfaces. The fact that the controller classes only access the repositories through a set of interfaces can also be seen as an example of the Facade pattern because it hides the complexity of using Hibernate from the controllers. The same design can also be seen as an example of the Strategy pattern because it allows different persistence strategies to be swapped out. As long as the interfaces do not change, the implementation classes could be replaced with alternate implementations that use JDBC code instead of Hibernate or even use a completely different data storage technology like a No SQL database. They could also be swapped out with mock implementations to facilitate unit testing without using the actual database.



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?

Walid

I now understand why it matters that our code is maintainable, especially if it is to be used in a large enterprise. Developers tend to spend a majority of time maintaining existing code so if an investment is not made up front to make code maintainable, then developers will end up spending much more time maintaining it in the future. For example, in our project, we created repository interfaces and then created classes that inherited from those interfaces and used Hibernate to retrieve data from the database. These classes can easily be replaced by other frameworks like Torque and Castor in the future if need be. There may be pressure on developers to spend more time on developing new features instead of ensuring that their existing code is maintainable because new features are generally what sells software. However, in many cases neglecting to dedicate time up front to making existing code maintainable may end up costing more in lost time than any new revenue a new feature can generate. Of course, this may not always be the case, especially for small projects or enterprises, but I now understand the benefits of Object-Oriented Design and Analysis in making code more maintainable.

Yi-Chen

Developing a system is more complicated than I expected. We have to spend lots of time to discuss what goal of the system we want to achieve. Then, establishing the basic concept of this system step by step to create it. Using various diagrams such as use case diagram, class diagram to specify the function of system and have clear understanding what's the goal of the system and how to design it.

Besides, how to implement the system is another issue. In the process of coding, we have to discuss and modify the code to make it work in the group application. Furthermore, learn how the frameworks works also new for me, but it is a good tool to make the implementation works in easier way. But it takes time to get used to this method. However, It makes communication between class and database becomes more systematic. And design pattern also plays an important role in this work. It organizes the classes that we set before using it. For example, at the beginning of discussion about this project, we focused on how to make the functions works. As a result, the class diagram in Part 2 looked complex. But, when we apply MVC pattern in project, it distinguishes the classes and make it organized.

In general, this is a good experience about how to set up a system and cooperates with team members. And I have better understanding of Object-Oriented Design and Analysis concepts and implementation.

Doug

One of the more challenging aspects of this project was the need to use three large, complex Java frameworks: JavaFX, Spring and Hibernate. Not only did we need to learn enough of the nuances of these frameworks to satisfy our requirements, but we also needed to integrate them all together in an elegant and natural way. The process of getting this all working involved trying many different things and looking at sample code from many different source. It would have been very easy for this process to result in spaghetti code. However, I believe that the work that we did ahead of time in carefully defining the use cases for the app and developing a clear design and architecture helped to prevent that from happening. The detailed use cases helped us to keep our focus on what the application's users would need and not just on what was easy to implement using our technology stack. Similarly, in planning our app, we decided to use an MVC architecture with loose coupling between the user interface code and the data access code. During the implementation of our app, this design work drove us to find elegant ways to integrate the frameworks in a way that fit our architecture instead just settling for the first thing we could get to work.

However, while the challenges of this project clearly showed the importance of upfront planning and design, it also demonstrated the value of another technique we discussed in class: refactoring. Even though our advanced planning and design work helped us avoid many problems, there were some complexities that we did not foresee that led to some less elegant code. The refactoring techniques we learned were invaluable in fixing these issues and getting our application back to a clean design. One example of this was when we realized that our controller classes all needed to perform several common tasks like retrieving the current logged in user or sending the user to another screen. We were able to avoid this leading to duplicate code by creating a base class for our controllers to extend from and moving the shared code into it. Another example occurred when we realized how complicated the logic was to keep track of which users could perform which actions at each step of the workflow. We were able to simplify this decision logic by introducing the State pattern into our application. My biggest takeaway from the project is that the combination of careful upfront planning to develop a good architecture and a commitment to continuously refactoring to deal with unforeseen complications that arise during development is a powerful way to write elegant, maintainable code.